Learning to Build the Environment: Self-Evolving Reasoning RL via Verifiable Environment Synthesis

Yucheng Shi[1], **Zhenwen Liang**[1], **Kishan Panaganti**[1], **Dian Yu**[1], **Wenhao Yu**[1], **Haitao Mi**[1]

[1]Tencent HY LLM

mailofsyc@gmail.com

We pursue a vision for self-improving language models in which the model does not merely generate problems or traces to imitate, but constructs the environments that train it. In zero-data reasoning RL, this reframes self-improvement from a data-generation loop into an environment-construction loop, where each artifact is a reusable executable object that samples instances, computes references, and scores responses. Whether this vision sustains improvement hinges on a single property: the environments must exhibit stable solve–verify asymmetry: the model must be able to write an oracle once that it cannot reliably execute in natural language on fresh instances. This asymmetry takes two complementary forms. Some tasks are algorithmically hard to reason through but trivial as code: a dynamic program or graph traversal, compiled once, yields unboundedly many calibrated instances. Others are intrinsically hard to solve but easy to verify, like planted subset-sum or constraint satisfaction. Both create a durable gap between proposing and solving that the policy cannot close by gaming the verifier, and it is this gap that keeps reward informative as the learner improves. We instantiate this view in EvoEnv, a single-policy generator-solver method that synthesizes Python environments from ten seeds and admits them only after staged validation, semantic self-review, solver-relative difficulty calibration, and novelty checks. The strongest evidence comes from the already-strong regime: on Qwen3-4B-Thinking, fixed public-data RLVR and fixed hand-crafted environment RLVR reduce the average, while EvoEnv improves it from 72.4 to 74.8, a relative gain of 3.3%. Stable self-improvement, we suggest, depends not on producing more synthetic data, but on models learning to construct worlds whose difficulty stays structurally beyond their own reach.

Date: May 13, 2026

1 Introduction

Language-model self-improvement is often framed as a data-generation problem: the model produces more questions, traces, solutions, or hard examples near its current frontier [13, 18, 19]. This framing is useful but incomplete. Capable LLMs do not improve only by imitating additional examples; they improve by interacting with environments that generate situations, impose constraints, and return feedback through execution, tools, tests, or state changes [32, 45]. We ask whether a language model can learn not only from self-generated examples, but from self-constructed training environments. We study this question in a deliberately controlled setting: zero-data reinforcement learning with verifiable rewards (RLVR) for reasoning. A reasoning environment is a reusable executable artifact with four routines: a sampler that generates latent task instances, an oracle that computes reference answers, a renderer that turns instances into natural-language prompts, and a scorer that evaluates solver responses. The model may author this artifact, but once the artifact is validated and admitted to the pool, its rewards are determined by execution rather than by the model's current sampled answers.

This distinction addresses two limitations of existing self-training recipes. Standard RLVR obtains reliable supervision from fixed datasets, answer-equivalence rules, unit tests, or executable checkers [10, 21, 42]. However, the verified distribution is fixed. As the policy improves, many prompts become almost always

solved, others remain almost always failed, and group-relative methods lose useful reward variation. Continued optimization on a saturated pool can then narrow behavior or cause forgetting rather than expand capability [11, 29, 38]. Stable verifiers are therefore not sufficient; the verified distribution must also stay near the solver's changing frontier.

Per-instance self-play provides adaptivity but can compromise label stability. If new prompts are labeled by self-consistency, majority vote, semantic clustering, or another signal derived from the same policy being optimized, the reward function moves with the learner [13, 18, 19]. If the model instead generates a one-off executable verifier for a single problem, correctness is better grounded, but the artifact is consumed after one rollout [51]. The resulting system still lacks a durable object that can be validated once, sampled repeatedly, calibrated against the current solver, retired when it saturates, and later reused as seed material.

Our thesis is that the unit of self-synthesis should be the environment rather than the individual problem. A generated problem gives one prompt and one label; a generated environment gives a distribution of prompts and a reusable executable reward source. The reason this environment-level unit is tractable is not merely amortization across rollouts, but a structural property we call *stable solve–verify asymmetry*: across a wide class of reasoning tasks, authoring or checking an executable procedure is easier than carrying out the corresponding natural-language reasoning process on fresh instances.

This asymmetry appears in two complementary forms. In *algorithmic* tasks, such as dynamic programming, graph traversal, modular recurrence, sorting, and sequence computation, the model may be able to write a compact oracle even when it cannot reliably execute that algorithm in natural language on arbitrary rendered instances. In *verification* tasks, such as planted subset-sum, feasibility checking, and constraint satisfaction, producing a valid answer can be hard, while checking a proposed answer is simple. Both forms create a durable gap between proposing and solving, a gap that the policy cannot close by gaming the verifier, because the verifier is frozen code. It is this gap that keeps reward informative as the learner improves, and it is what distinguishes self-*built environments* from self-*generated problems*.

We instantiate this view in EvoEnv, a single-policy dual-role trainer. The same policy alternates between a *generator* role, which proposes Python environments from a small seed pool, and a *solver* role, which answers fresh prompts sampled from the accepted environment pool. Candidate environments enter the pool only when they satisfy four contracts: they execute under a strict interface; their oracle and scorer match the advertised task under conservative semantic self-review; sampled instances are hard-but-solvable for the current policy; and the pool remains broad enough to avoid template collapse. We enforce these contracts through L1–L5 validation, semantic self-review, novelty gating, in-batch deduplication, and pool rotation.

The strongest evidence for this framing comes from the already-strong model regime. On Qwen3-4B-Thinking, fixed public-data RLVR and fixed hand-crafted environment RLVR both reduce the average score in Table 1, while EvoEnv improves it. This result suggests that self-evolving environments are not merely a way to obtain more synthetic data for weak models. They are a way to keep reward stable and frontier-calibrated when static distributions have already saturated or become misaligned with the learner. Our contributions are:

1. We formulate verifiable environment synthesis for zero-data reasoning RL, identifying *stable solve–verify asymmetry*, in both algorithmic and verification forms, as the structural property that allows self-built environments to serve as durable reward sources rather than policy-coupled pseudo-labels.

2. We introduce EvoEnv, a single-policy proposer–solver algorithm that curates a self-generated environment pool through staged validation, semantic self-review, solver-relative difficulty calibration, novelty control, and pool rotation.

3. We show that EvoEnv improves three model families and, in particular, improves an already-strong thinking-mode checkpoint where both fixed public-data RLVR and fixed hand-crafted environment RLVR reduce average performance.

2 Related Work

We position EvoEnv by asking two questions: what object is reused during training, and where does its reward come from? Prior work provides stable verifiers, adaptive self-generated curricula, executable task checks, or environment-level training. EvoEnv combines these ingredients in a specific setting: zero-data reasoning RL, where the learner itself authors reusable executable environments, and each environment is admitted only after validation, solver-relative calibration, and novelty filtering.

Fixed-distribution RLVR. Verifier-backed RL has been highly effective for reasoning because outcome rewards can be computed without a learned preference model [10, 21, 42]. Most RLVR systems, however, train on a fixed set of prompts and checkers. This is reliable but not adaptive: once examples become almost always solved or almost always failed, group-relative rewards lose useful variation, and continued optimization can lead to saturation, over-specialization, or forgetting [11, 29, 38]. EvoEnv keeps the key benefit of RLVR—execution-grounded reward—but replaces the static prompt pool with a changing pool of validated executable environments.

Self-generated curricula. A broad line of self-improvement methods trains on model-generated rationales, questions, preferences, or judgments [7, 12, 30, 43, 44]. Recent zero-data methods go further by letting the model propose its own tasks and estimate correctness through majority vote, self-consistency, internal feedback, or co-evolving agents [4, 5, 9, 13, 16, 24, 35, 48, 49, 52, 56]. These approaches are adaptive, but their reward signals are often policy-coupled: the same model family that learns also helps decide what is correct. Novelty-based variants such as EVOL-RL mitigate collapse in these loops [54]. EvoEnv shares the goal of adaptive self-improvement, but it does not trust the model's sampled answers as labels; it trusts only executable artifacts that pass admission and are then frozen for scoring.

Executable grounding. Several methods ground self-generated tasks through code or tests. Absolute Zero Reasoner verifies program/input/output tasks by execution [51]; Self-Challenging Agents generate Code-as-Task verifiers [53]; SPC trains adversarial critics for reasoning errors [3]; and agentic systems extend similar ideas to tool use and software engineering [37, 40, 55]. These works are closer to ours because correctness is no longer purely self-believed. The main difference is the reuse unit: they typically generate tasks, tests, or episodes, whereas EvoEnv generates an environment-level object that can sample many fresh instances, be calibrated against the current solver, retired when saturated, and reused as seed material.

Environment-level training. The closest neighbors also treat environments as training objects. Some environments are externally given, such as fixed-rule games, language games, document corpora, or hand-authored verifiable Python suites [15, 18–20, 22, 46]. Others synthesize tool, web, UI, or embodied environments through offline pipelines [2, 17, 25, 31, 39, 50]. Learned-simulator approaches instead use an LLM world model, so reward depends on the simulator's beliefs [6, 8, 36]. EvoEnv studies a more controlled case: the same policy that solves reasoning tasks also authors deterministic Python environments, while admission is on-policy, solver-calibrated, and execution-grounded rather than human-authored, offline-pipeline-generated, or simulator-defined. Appendix A gives a fuller comparison.

3 Method

EvoEnv trains a single policy to do two related things: solve verifiable reasoning tasks, and construct the executable environments from which such tasks are sampled. The central object is therefore not a generated problem, but a reusable environment. A candidate environment is allowed to influence solver training only after it passes mechanical validation, conservative semantic review, solver-relative difficulty calibration, and novelty filtering. The design goal is simple: teach the model to construct new training worlds, while ensuring that rewards used for solver updates come from frozen execution rather than from the model's current sampled answers. A complete workflow is presented in Figure 1 and Algorithm 1.

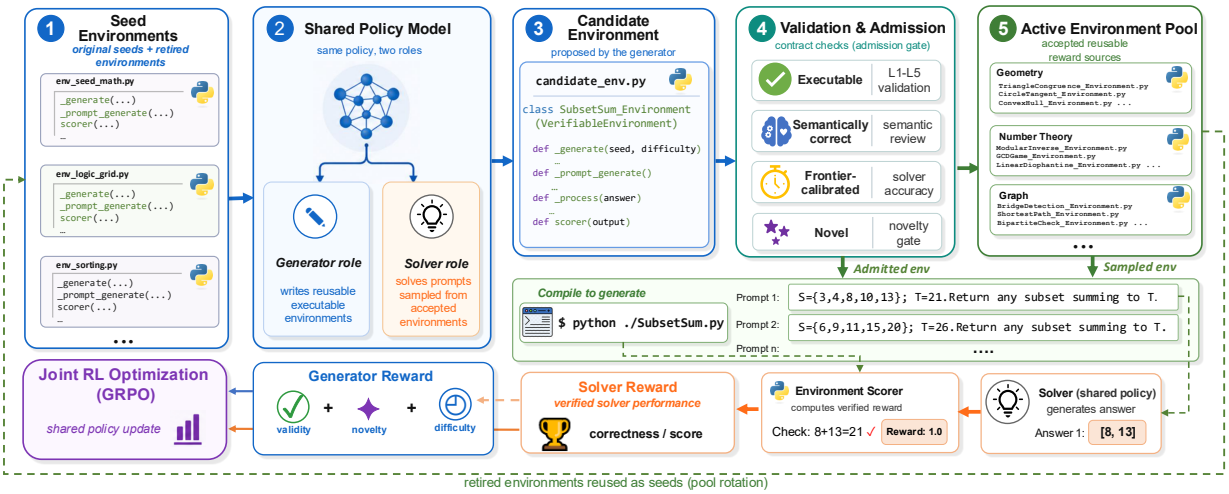

Figure 1 Overview of EvoEnv. A single policy generates reusable executable environments and solves fresh instances sampled from the accepted pool. Candidate environments enter the pool only after validation, semantic review, solver-relative difficulty calibration, and novelty filtering.

3.1 Environment interface

A verifiable environment is a reusable executable object, not a single labeled problem. We write it as $e = (G_e, \Pi_e, S_e)$. Given a seed σ and difficulty parameter δ, the generator–oracle routine produces a latent instance and reference answer, $(z, a) = G_e(\sigma, \delta)$. The solver observes only the rendered prompt $x = \Pi_e(z)$ and receives reward $R_e(y; \sigma, \delta) = S_e(y, z, a)$ for its response y. Thus, once an environment is admitted, the reward source is a frozen executable path: for fixed e, σ, δ, the current policy can change only the sampled response, not the reference answer or scoring rule.

```python
import random

class SortingEnv(VerifiableEnvironment):

    def _generate(self, seed, difficulty):   # G_e
        rng = random.Random(seed)
        z = [rng.randint(0, 99)
             for _ in range(difficulty)]
        return z, sorted(z)                  # (z, a)

    def _prompt_generate(self, z):           # Pi_e
        arr = " ".join(map(str, z))
        return "Sort: " + arr                # x

    def _process(self, y):
        try:
            return list(map(int, y.split()))
        except ValueError:
            return None

    def scorer(self, y, z, a):               # S_e
        return float(self._process(y) == a)
```

Figure 2 A minimal sorting environment. The method `_generate` implements G_e, `_prompt_generate` implements Π_e, and `_process` with `scorer` implements S_e.

Figure 2 instantiates this interface with a minimal sorting environment. The environment samples an array, computes the executable reference answer with `sorted`, renders the array as a model-facing prompt, and scores the parsed response by exact comparison. This illustrates the operational solve–verify asymmetry used throughout EvoEnv: writing and running this code is relatively easy, but solving many fresh rendered instances still supplies nontrivial training signal for the language model.

In implementation, each candidate is emitted as a Python subclass of `VerifiableEnvironment`. The method `_generate(seed, difficulty)` implements G_e by constructing z and computing a; `_prompt_generate` implements Π_e; and `_process` together with `scorer` implements S_e. Candidates are restricted to an approved standard-library subset and executed in sandboxed subprocesses with wall-clock timeouts.

This interface covers both equality-check oracles, such as sorting, dynamic programming, graph traversal, modular recurrence, and sequence computation, and feasibility-check oracles, such as planted subset-sum and constraint satisfaction. Feasibility tasks require stronger scorer probes because multiple answers may be valid. Appendix B gives the full data-flow diagram, and Appendix D provides a complete planted subset-sum example.

3.2 Validation and semantic review

Our method is equipped with multi-layer validation. Let $\ell(e)$ denote the highest validation layer reached by candidate e. L1 extracts parseable Python and checks that the expected class and methods exist. L2 instantiates the class and runs generation, prompt rendering, parsing, and scoring on several seeds and difficulty settings. L3 checks determinism by repeating generation under identical seeds and comparing the latent instance, prompt, and reference object. L4 checks non-triviality by requiring variation across seeds and difficulty values. L5 checks the local scorer contract: the stored reference object must score positively; injected perturbations, malformed answers, and type-mismatched answers must not; and parsing must not leak the hidden reference object. Only L5 candidates proceed to semantic review and solver-relative calibration.

Mechanical execution alone cannot prove that the generated code implements the task described in the prompt. A candidate can be deterministic and non-trivial while computing the wrong recurrence, rewarding the wrong target, or accepting malformed answers. We therefore add a conservative semantic-review filter. The reviewer receives the candidate source code, one or more concrete generated instances, the reference object, the rendered prompt, and scorer probes. It is asked to perform a code-review task: trace the advertised task, check that the generator computes the advertised quantity, and search for domain relabeling, hidden answer leakage, and overly permissive parsing.

The reviewer is the same policy used elsewhere in training, but its verdict is not used as generator reward. This distinction is important. The generator's scalar reward is computed from mechanical validation, solver-relative difficulty, and novelty; the semantic reviewer only decides whether an otherwise valid candidate may enter the active solver-training pool. A rejected candidate may still contribute its mechanically computed generator rollout reward, but it cannot become a reward source for solver training. This removes the direct channel by which the generator could be optimized to satisfy the reviewer's natural-language preferences.

We run $K_{\text{rev}} = 3$ independent reviews and use an any-reject rule: if any review identifies a likely semantic bug, the candidate is rejected from pool admission. The review task is substantially more local than solving benchmark problems from scratch: the reviewer sees the source, hidden state, reference object, and scorer behavior, and only needs to check consistency among them. As an additional sanity check, we audit this same-policy review against a stronger external reviewer and find high agreement; details are reported in Appendix E.

3.3 Difficulty and novelty rewards

For each L5 candidate, we estimate whether its sampled instances produce useful outcome variation for the current solver. We sample $m = 8$ calibration instances by running G_e, draw a single solver response per instance, and average:

$$\hat{a}_m(e;\pi_\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{I}[S_e(y_i, z_i, a_i) > 0], \qquad (z_i, a_i) = G_e(\sigma_i, \delta), \quad y_i \sim \pi_\theta(\cdot \mid \Pi_e(z_i)). \tag{1}$$

We use one response per instance to estimate the pass rate without inflating compute. Candidates with $\hat{a}_m(e;\pi_\theta) = 0$ are too hard, underspecified, or overly strict under the current solver; candidates with $\hat{a}_m(e;\pi_\theta) = 1$ are saturated or overly permissive. We therefore require $0 < \hat{a}_m(e;\pi_\theta) < 1$ for admission. The generator's difficulty reward grades each L5 candidate by its solver-relative pass rate using a piecewise schedule:

$$Q_{\text{unc}}(e;\theta) = \exp\left(-\frac{(\hat{a}_m(e;\pi_\theta) - a^\star)^2}{2\sigma_a^2}\right), \qquad a^\star = 0.3. \tag{2}$$

The target $a^\star = 0.3$ biases generation toward environments that are solvable but still challenging: the current solver succeeds often enough to produce positive examples, but fails often enough to preserve useful reward variation. We choose a target below 0.5 because near-half accuracy candidates can become saturated quickly as the solver improves.

Novelty prevents the generator from repeatedly producing the first template that passes validation. We embed each environment with a frozen external embedding model, `all-MiniLM-L6-v2` [26], rather than with the training policy itself. Each candidate has two embeddings: a prompt embedding u_e from its `prompt_template` and a code embedding v_e from its cleaned `_generate` body. Let $\mathcal{C}_p$ and $\mathcal{C}_c$ be caches of prompt and code embeddings for previously admitted environments. We compute

$$\mathrm{sim}_t(e) = \lambda \max_{u' \in \mathcal{C}_p} \cos(u_e, u') + (1-\lambda) \max_{v' \in \mathcal{C}_c} \cos(v_e, v'), \qquad \lambda = 0.5, \tag{3}$$

and define $N_t(e) = 1 - \mathrm{sim}_t(e)$. The two-view novelty score is a guardrail to prevent surface-level duplicates. Prompt embeddings alone can miss code clones with different story wrappers, and code embeddings alone can miss the same task written in different language. Combining prompt and code views makes exact or near-exact duplication less attractive. At the same time, we also acknowledge that surface variants can still be useful training environments when they are semantically valid and present the solver with different natural-language contexts. The novelty weight adapts to the repetitiveness of the accepted stream. Let $\bar{s}_t$ be an exponential moving average of within-batch maximum similarity. We set

$$\gamma_t = \gamma_{\min} + (\gamma_{\max} - \gamma_{\min}) \cdot \mathrm{clip}\left(\frac{\bar{s}_t - \tau_{\text{low}}}{\tau_{\text{high}} - \tau_{\text{low}}}, 0, 1 \right). \tag{4}$$

Exploration pressure therefore rises when the accepted pool becomes repetitive and relaxes when new environments are already diverse. The full generator reward combines layered validation with a novelty bonus:

$$R_{\text{gen}}(e; \theta, t) = Q_{\text{val}}(\ell(e), \hat{a}_m(e; \pi_\theta)) + \mathbb{I}[\ell(e) \geq 2]\, \gamma_t\, N_t(e). \tag{5}$$

Here the validation term is

$$Q_{\text{val}}(\ell, \hat{a}) = -\mathbb{I}[\ell < 1] - \frac{1}{2}\mathbb{I}[\ell = 1] - \frac{1}{4}\mathbb{I}[\ell = 2] + \mathbb{I}[\ell = 5]\, Q_{\text{unc}}(\hat{a}). \tag{6}$$

The validation term penalizes mechanically invalid candidates, assigns zero reward to candidates that pass execution-level checks but fail top-layer validation, and gives the solver-relative uncertainty reward Q_{unc} only to L5 candidates. The novelty bonus is gated by $\ell(e) \geq 2$, so unparseable or syntactically broken candidates cannot recover through novelty alone. Crucially, the semantic-review verdict is not a term in R_{gen}; it only affects the pool-admission decision.

3.4 Pool admission and joint optimization

A candidate is inserted into the active solver-training pool only if it satisfies all four admission requirements:

$$A_t(e) = \mathbb{I}[\ell(e) = 5] \cdot \mathbb{I}[\text{review}(e) = 1] \cdot \mathbb{I}[0 < \hat{a}_m(e; \pi_\theta) < 1] \cdot \mathbb{I}[\mathrm{sim}_t(e) < \tau_{\text{gate}}]. \tag{7}$$

Accepted environments enter $\mathcal{P}_t$ and may later be used both for solver training and as examples in $\mathcal{S}_t$ for future generator prompts. At each training step, we form solver and generator rollout groups.

A solver group samples an environment $e \sim \mathcal{P}_t$, generates an instance and prompt using G_e and Π_e, samples multiple responses from π_θ, and scores each response with the frozen environment scorer S_e. Thus the solver receives ordinary verifier-backed RL feedback: it is rewarded only for producing an answer accepted by the environment.

A generator group samples multiple candidate environments from the same few-shot generator prompt built from $\mathcal{S}_t$. Each candidate receives the reward in Eq. equation 5; candidates satisfying Eq. equation 7 are additionally admitted into the active pool. We normalize advantages within solver groups and generator groups separately, then optimize the shared policy with a role-conditioned GRPO objective:

$$\mathcal{L}(\theta) = \mathcal{L}^{\text{GRPO}}_{\text{solver}}(\theta) + w_{\text{gen}} \mathcal{L}^{\text{GRPO}}_{\text{gen}}(\theta) + \beta\, \mathrm{KL}(\pi_\theta \,\|\, \pi_{\text{ref}}), \qquad w_{\text{gen}} = 0.3. \tag{8}$$

Algorithm 1 One self-evolution iteration of EvoEnv.

Require: policy π_{θ_t}; active pool $\mathcal{P}_t$; seed pool $\mathcal{S}_t$; novelty cache $\mathcal{C}_t$
Ensure: updated policy $\pi_{\theta_{t+1}}$ and pools $(\mathcal{P}_{t+1}, \mathcal{S}_{t+1})$

Initialize generator data $\mathcal{D}_t^g \leftarrow \emptyset$, solver data $\mathcal{D}_t^s \leftarrow \emptyset$, accepted set $\mathcal{A}_t \leftarrow \emptyset$
Sample candidate environments $\{e_i, \tau_i^g\}_{i=1}^M \sim \pi_{\theta_t}(\cdot \mid \mathsf{Prompt}_{\text{gen}}(\mathcal{S}_t))$
for $i = 1, \ldots, M$ **do**
    $(r_i^g, A_i) \leftarrow \textsc{EvalEnv}(e_i; \pi_{\theta_t}, \mathcal{C}_t)$
    $\mathcal{D}_t^g \leftarrow \mathcal{D}_t^g \cup \{(\tau_i^g, r_i^g)\}$; if $A_i = 1$, add e_i to $\mathcal{A}_t$
end for
for $j = 1, \ldots, B$ **do**
    Sample $e \sim \mathcal{P}_t \cup \mathcal{A}_t$, $(z, a) \sim G_e$, $x = \Pi_e(z)$, and responses $\{y_{jm}\}_{m=1}^K \sim \pi_{\theta_t}(\cdot \mid x)$
    $\mathcal{D}_t^s \leftarrow \mathcal{D}_t^s \cup \{(\tau_{jm}^s, S_e(y_{jm}, z, a))\}_{m=1}^K$
end for
$\theta_{t+1} \leftarrow \text{GRPO}(\theta_t; \mathcal{D}_t^g, \mathcal{D}_t^s)$
$(\mathcal{P}_{t+1}, \mathcal{S}_{t+1}, \mathcal{C}_{t+1}) \leftarrow \textsc{UpdatePool}(\mathcal{P}_t, \mathcal{S}_t, \mathcal{C}_t, \mathcal{A}_t)$

The two roles use distinct prompts but share parameters. The single-policy design is chosen for efficiency: the same model that benefits from executable reasoning environments also learns the code patterns needed to construct them. In practice, separate group normalization and the generator weight w_{gen} prevent long, high-variance generator trajectories from dominating solver updates.

Pool rotation controls the long-horizon curriculum. After an environment has been used for a fixed number of solver epochs, it is retired from $\mathcal{P}_t$ and is no longer sampled for direct solver training. Retired environments can instead enter $\mathcal{S}_t$ as few-shot examples for future generator prompts. Original seed environments are protected by a floor $\rho_{\min} = 0.2$: an original seed is not retired if doing so would reduce the share of originals in the active pool below the floor. This keeps the generator context anchored while allowing the solver-training pool to move toward newly synthesized environments.

4 Experiments

Our experiments test three questions. (Q1) Does EvoEnv improve downstream reasoning over the untrained base and strong RLVR baselines? (Q2) Does environment synthesis sustain a frontier signal over training, or does it saturate like a fixed pool? (Q3) Which components are load-bearing?

4.1 Setup

Models. We evaluate on three diverse base models: **Qwen3-4B-Instruct-2507** [41], a standard instruction-tuned 4B model; **Qwen3-4B-Thinking-2507** [41], an already-strong thinking-mode checkpoint; and **Nemotron-Cascade-8B-IFRL** [33], an 8B thinking model from a different family.

Seeds and training. All runs start from the same fixed set of ten seed environments spanning sorting, dynamic programming, graph, and number-theoretic templates (Appendix F). No external problem-answer data is used. We train with single-policy GRPO, generator advantage scale $w_{\text{gen}} = 0.3$, and the hyperparameters in Appendix I. All models are trained for 100 steps.

Evaluation. We report pass@1 [%] across eight benchmarks using the Nemo-Skills toolkit [23]. For the small-problem-set competition math benchmarks, AIME 2024, AIME 2025, HMMT Feb 2025, Beyond-AIME [28], and Brumo 2025 [1], we average over 16 runs to reduce variance; AMC 2023 [1], GPQA Diamond [27], and LiveCodeBench v6, 2024-08–2025-05 (LCB) [14] are reported as single-run pass@1. Unless stated otherwise, evaluation uses up to 128k context, symbolic-equivalence checking, temperature 0.6, and top-p 0.95.

Baselines. We compare against: **Untrained**, the base checkpoint under the same decoding protocol; **R-Zero** [13], a zero-data proposer–solver RLVR method; **DAPO** [42], GRPO trained on DAPO-Math-17k at

Table 1 Main results across three base models and eight benchmarks. "Avg" is the mean result across the reported cells in that row. Best result per model block is **bolded**.

Base Model	Method	AIME'24	AIME'25	AMC'23	HMMT	B-AIME	Brumo	GPQA	LCB	Avg
Qwen3-4B Instruct-2507	Untrained	58.8	44.0	87.5	30.0	32.0	56.5	52.5	32.6	49.2
	R-Zero	55.0	43.3	87.5	26.5	27.0	55.2	54.0	33.5	47.8
	DAPO	50.0	44.8	92.5	19.8	22.2	47.1	56.1	33.5	45.8
	RLVE	57.1	44.3	92.5	27.7	32.8	57.9	**57.1**	32.6	50.3
	EvoEnv	**61.9**	**52.5**	**95.0**	**30.0**	**33.6**	**61.2**	55.0	**35.5**	**53.1**
Qwen3-4B Thinking-2507	Untrained	**86.7**	78.8	97.5	56.9	53.1	**81.9**	65.2	59.0	72.4
	R-Zero	84.2	80.6	97.5	53.5	52.9	79.6	67.2	59.0	71.8
	DAPO	74.0	65.4	97.5	48.8	45.8	68.3	64.6	54.0	64.8
	RLVE	81.9	75.0	95.0	51.5	49.6	75.8	67.7	57.3	69.2
	EvoEnv	86.2	**83.0**	**100.0**	**60.0**	**53.6**	**81.9**	**70.2**	**63.2**	**74.8**
Nemotron Cascade-8B	Untrained	83.3	71.3	92.5	61.5	55.8	73.8	65.2	**64.3**	71.0
	R-Zero	83.3	71.5	97.5	55.8	50.1	73.3	58.1	62.1	69.0
	DAPO	78.3	63.5	**100.0**	40.0	40.6	65.8	61.6	60.6	63.8
	RLVE	79.8	69.4	**100.0**	55.4	54.3	74.2	**66.2**	56.6	69.5
	EvoEnv	**84.8**	**72.3**	**100.0**	**64.2**	**57.2**	**78.8**	65.7	62.8	**73.2**

matched compute; and **RLVE** [46], fixed-environment RLVR with hand-curated environments and matched training steps.

4.2 Main results: adaptive environments remain useful where fixed distributions fail

Table 1 presents the main results. The rightmost column reports the mean pass@1 across the benchmarks in each row. Three patterns stand out. First, EvoEnv is the only method that improves all three model families. It raises the average from 49.2 to 53.1 on Qwen3-4B-Instruct, from 72.4 to 74.8 on the already-strong Qwen3-4B-Thinking checkpoint, and from 71.0 to 73.2 on Nemotron-Cascade-8B, a thinking model from a different family. These correspond to relative gains of 7.9%, 3.3%, and 3.1%, respectively. Second, the strong-model regime highlights the weakness of fixed curricula. On Qwen3-4B-Thinking, DAPO and RLVE reduce the average to 64.8 and 69.2, while EvoEnv still improves it. This suggests that, once a static verified distribution becomes saturated or misaligned, stable reward alone is not enough; the training environment must also evolve with the solver. Third, the gains are not confined to the exact environments used for training. EvoEnv trains only on self-generated executable environments from ten seeds, without using problem-answer data from the evaluation benchmarks. Nevertheless, the resulting policy improves on external benchmarks with different formats, including GPQA Diamond (65.2→70.2) and LiveCodeBench (59.0→63.2) on Qwen3-4B-Thinking. This suggests that the evolving environment pool induces transferable reasoning behavior rather than only memorizing synthetic environment templates.

4.3 Training dynamics: lower training score can mean a healthier frontier

Figure 3 contrasts EvoEnv with a fixed-environment RLVR baseline over 100 on-policy steps on Qwen3-4B-Instruct-2507. The striking pattern here is the inverse relationship between training score and held-out accuracy. In EvoEnv, the mean training score decreases because the generator keeps raising the difficulty of the solver's environment pool. Rather than indicating degradation, the declining score is the mechanism that preserves frontier signal: the solver is repeatedly placed in regimes where it can sometimes succeed and sometimes fail. The fixed baseline maintains a roughly constant training score on a saturated pool and correspondingly shows little held-out improvement. The crucial effect is that environment synthesis keeps reward informative.

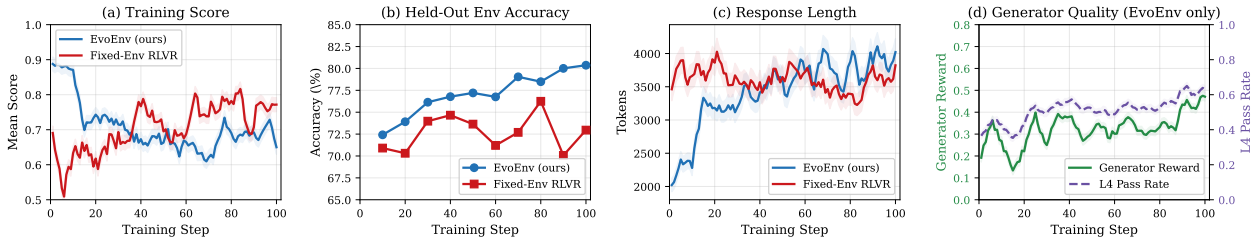

Figure 3 Training dynamics of EvoEnv vs. fixed-environment RLVR on Qwen3-4B-Instruct-2507. In EvoEnv, the mean solver training score decreases from 0.88 to 0.61 as the generator synthesizes harder environments, while held-out accuracy on 50 unseen RLVE environments rises from 72.4% to 80.4% [46]. The fixed baseline stagnates near 72%.

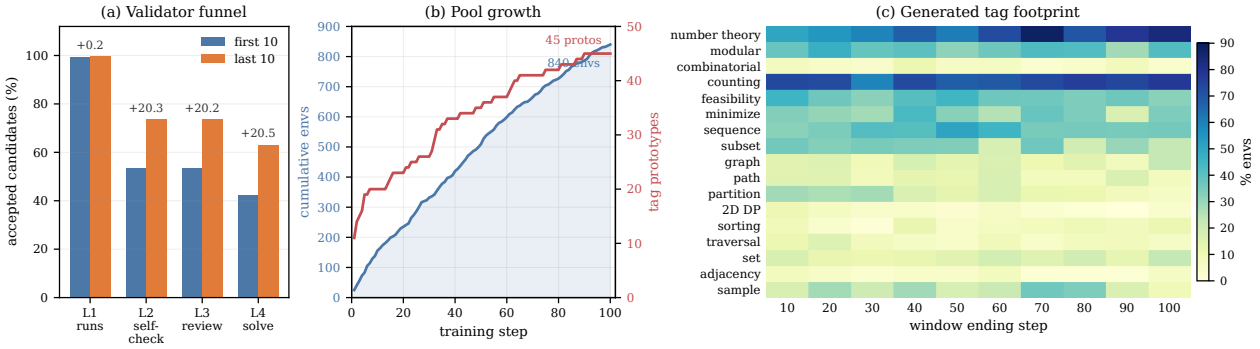

Figure 4 Data-level audit for the 100-step Qwen3-4B-Instruct run. (a) The generator's accepted candidates improve through the validation funnel: L2/L3 self-check and review pass rates rise by about 20 percentage points, and L4 validation rises from 42.4% to 63.0% between the first and last ten steps. (b) The accepted pool keeps growing, reaching 840 generated environments and 45 tag prototypes by step 100. (c) Generated-only semantic tag frequencies across 10-step windows; the pool contains various types including counting, number-theoretic, modular, and sequence/subset-style environments rather than a single copied seed template.

4.4 Data-level audit: executable, expanding, and interpretable

Figure 4 gives three direct checks. First, the generator becomes a better environment author under the actual admission pipeline: L2 and L3 pass rates increase from 53.4% to about 73.6%, L4 validation increases from 42.4% to 63.0%, and the generator reward rises from 0.292 to 0.452 between the first and last ten steps.

Second, the accepted data stream does not collapse after the initial seed expansion. Starting from only ten seed environments, EvoEnv accumulates 840 accepted environments within 100 steps, while the tag-prototype count reaches 45. Here a *tag prototype* is registered whenever a new environment's binary tag vector has Jaccard similarity below τ=0.5 to every existing prototype, so the count lower-bounds the number of structurally distinct environment families. Reaching 45 prototypes from 10 seeds demonstrates that EvoEnv generates environment types far beyond the initial seed distribution rather than recycling a narrow set of templates.

Third, the generated-only tag heatmap shows why these environments are useful for RLVR training. The dominant mass is in countable, automatically checkable tasks—number theory appears in 68.2% of generated environments, and modular or sequence structure in about 38% each—but the pool also includes feasibility, minimization, subset, graph/path, traversal, set, and sampling mechanisms. This gives the solver many executable reasoning problems with shared verifiability but varied surface forms, which is the data-side explanation for why EvoEnv can keep supplying non-saturated reward.

4.5 Reward component ablation: difficulty and diversity are jointly necessary

We isolate the contribution of the two generator-reward components: quality reward (validation and difficulty shaping) and diversity reward (novelty-gated exploration bonus). Each ablation removes one component while keeping training steps, seeds, and evaluation protocol identical to the full EvoEnv run on Qwen3-4B-

Table 2 Reward component ablation on Qwen3-4B-Thinking-2507. "w/o Quality" removes validation and difficulty shaping; "w/o Diversity" removes the novelty-gated exploration bonus. Avg Δ is the mean improvement over the untrained baseline across the seven reported benchmarks.

Method	AIME'24	AIME'25	HMMT	B-AIME	Brumo	GPQA	LCB	Avg Δ
Untrained	**86.7**	78.8	56.9	53.1	81.9	65.2	59.0	—
EvoEnv (full)	86.2	**83.0**	**60.0**	**53.6**	81.9	**70.2**	**63.2**	**+2.4**
w/o Quality	85.2	80.8	55.2	52.4	**82.7**	67.2	61.5	+0.5
w/o Diversity	85.2	81.5	57.3	53.4	81.2	66.7	60.4	+0.6

Thinking-2507. Both components are important. Removing quality reward reduces the average gain from +2.4 to +0.5, with the largest drops on HMMT, GPQA, and AIME 2025. Removing diversity reward reduces the gain to +0.6 and particularly hurts GPQA and LiveCodeBench, suggesting that template breadth is a driver of out-of-distribution transfer rather than merely a cosmetic regularizer. The interaction is intuitive: difficulty without diversity yields narrow competence, while diversity without difficulty yields many environments that fail to supply gradient.

5 Conclusion

EvoEnv suggests a path toward models that build part of their own training environment while retaining stable reward. The key is not to trust the model's answers, but to ask it for executable structure that can be validated, frozen, sampled, and reused. In zero-data reasoning RL, this turns self-improvement from a problem-generation loop into an environment-construction loop. Compile-once, solve-many is the mechanism; the broader claim is that stable self-improvement may depend on models learning to construct the worlds that train them.

References

[1] Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. 2026. URL https://arxiv.org/abs/2505.23281.

[2] Hyungjoo Chae, Jungsoo Park, and Alan Ritter. Safe and scalable web agent learning via recreated websites, 2026. URL https://arxiv.org/abs/2603.10505.

[3] 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, 2025.

[4] Lili Chen, Mihir Prabhudesai, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Self-questioning language models, 2025.

[5] Yixing Chen, Yiding Wang, Siqi Zhu, Haofei Yu, Tao Feng, Muhan Zhang, Mostofa Patwary, and Jiaxuan You. Multi-agent evolve: Llm self-improve through co-evolution, 2025.

[6] Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason Weston, and Dat Huynh. Scaling agent learning via experience synthesis, 2025. URL https://arxiv.org/abs/2511.03773.

[7] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models, 2024.

[8] Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. Webevolver: Enhancing web agent self-improvement with coevolving world model, 2025. URL https://arxiv.org/abs/2504.21024.

[9] 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, 2025.

[10] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[11] Bingxiang He, Yuxin Zuo, Zeyuan Liu, Shangziqi Zhao, Zixuan Fu, Junlin Yang, Cheng Qian, Kaiyan Zhang, Yuchen Fan, Ganqu Cui, et al. How far can unsupervised rlvr scale llm training?, 2026.

[12] Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners, 2024.

[13] 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, 2025. URL https://arxiv.org/abs/2508.05004.

[14] 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.

[15] Jakub Grudzien Kuba, Mengting Gu, Qi Ma, Yuandong Tian, Vijai Mohan, and Jason Chen. Language self-play for data-free training, 2025. URL https://arxiv.org/abs/2509.07414.

[16] Wai-Chung Kwan, Joshua Ong Jun Leang, Pavlos Vougiouklis, Jeff Z Pan, Marco Valentino, and Pasquale Minervini. Opensir: Open-ended self-improving reasoner, 2025.

[17] Zixing Lei, Sheng Yin, Yichen Xiong, Yuanzhuo Ding, Wenhao Huang, Yuxi Wei, Qingyao Xu, Yiming Li, Weixin Li, Yunhong Wang, and Siheng Chen. Embomatrix: A scalable training-ground for embodied decision-making, 2025. URL https://arxiv.org/abs/2510.12072.

[18] Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, et al. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning, 2025.

[19] 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, 2025.

[20] Mickel Liu, Liwei Jiang, Yancheng Liang, Simon Shaolei Du, Yejin Choi, Tim Althoff, and Natasha Jaques. Chasing moving targets with online self-play reinforcement learning for safer language models, 2025.

[21] Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models, 2025.

[22] Hongliang Lu, Yuhang Wen, Pengyu Cheng, Ruijin Ding, Jiaqi Guo, Haotian Xu, Chutian Wang, Haonan Chen, Xiaoxi Jiang, and Guanjun Jiang. Search self-play: Pushing the frontier of agent capability without supervision, 2025.

[23] Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset. 2025. URL https://arxiv.org/abs/2504.16891.

[24] Archiki Prasad, Weizhe Yuan, Richard Yuanzhe Pang, Jing Xu, Maryam Fazel-Zarandi, Mohit Bansal, Sainbayar Sukhbaatar, Jason Weston, and Jane Yu. Self-consistency preference optimization, 2024.

[25] Ram Ramrakhya, Andrew Szot, Omar Attia, Yuhao Yang, Anh Nguyen, Bogdan Mazoure, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration, 2025. URL https://arxiv.org/abs/2509.25047.

[26] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. 2019. URL https://arxiv.org/abs/1908.10084.

[27] 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. 2023. URL https://arxiv.org/abs/2311.12022.

[28] ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, et al. Seed1. 5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*, 2025.

[29] Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, et al. Spurious rewards: Rethinking training signals in rlvr, 2025.

[30] 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, 2023.

[31] 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, 2026. URL https://arxiv.org/abs/2601.05808.

[32] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

[33] Boxin Wang, Chankyu Lee, Nayeon Lee, Sheng-Chieh Lin, Wenliang Dai, Yang Chen, Yangyi Chen, Zhuolin Yang, Zihan Liu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nemotron-cascade: Scaling cascaded reinforcement learning for general-purpose reasoning models. 2026. URL https://arxiv.org/abs/2512.13607.

[34] 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, 2019. URL https://arxiv.org/abs/1901.01753.

[35] Shaobo Wang, Zhengbo Jiao, Zifan Zhang, Yilang Peng, Xu Ze, Boyu Yang, Wei Wang, Hu Wei, and Linfeng Zhang. Socratic-zero: Bootstrapping reasoning via data-free agent co-evolution, 2025.

[36] Yiming Wang, Da Yin, Yuedong Cui, Ruichen Zheng, Zhiqian Li, Zongyu Lin, Di Wu, Xueqing Wu, Chenchen Ye, Yu Zhou, and Kai-Wei Chang. Llms as scalable, general-purpose simulators for evolving digital agent training, 2025. URL https://arxiv.org/abs/2510.14969.

[37] 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, 2025. URL https://arxiv.org/abs/2512.18552.

[38] Haoze Wu, Cheng Wang, Wenshuo Zhao, and Junxian He. Mirage or method? how model-task alignment induces divergent rl conclusions, 2025. URL https://arxiv.org/abs/2508.21188.

[39] Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, Liang Chen, Yuyao Zhai, Bang Liu, Chenglin Wu, and Yuyu Luo. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines, 2026. URL https://arxiv.org/abs/2602.14296.

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

[41] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025.

[42] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[43] Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models, 2024.

[44] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning, 2022.

[45] Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[46] Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, Chenyang Zhao, Yulia Tsvetkov, Simon Shaolei Du, Natasha Jaques, Hao Peng, Pang Wei Koh, and Hannaneh Hajishirzi. Rlve: Scaling up reinforcement learning for language models with adaptive verifiable environments, 2025. URL https://arxiv.org/abs/2511.07317.

[47] Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents, 2026. URL https://openreview.net/forum?id=pUpzQZTvGY.

[48] Qingyang Zhang, Haitao Wu, Changqing Zhang, Peilin Zhao, and Yatao Bian. Right question is already half the answer: Fully unsupervised llm reasoning incentivization, 2025.

[49] Zhengxin Zhang, Chengyu Huang, Aochong Oliver Li, and Claire Cardie. Better llm reasoning via dual-play, 2025.

[50] Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training, 2026. URL https://arxiv.org/abs/2601.04126.

[51] Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025. URL https://arxiv.org/abs/2505.03335.

[52] Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. Learning to reason without external rewards, 2025.

[53] Yifei Zhou, Sergey Levine, Jason E Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents, 2026. URL https://openreview.net/forum?id=9yusqX9DpR.

[54] Yujun Zhou, Zhenwen Liang, Haolin Liu, Wenhao Yu, Kishan Panaganti, Linfeng Song, Dian Yu, Xiangliang Zhang, Haitao Mi, and Dong Yu. Evolving language models without labels: Majority drives selection, novelty promotes variation, 2025.

[55] Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments, 2025. URL https://arxiv.org/abs/2512.12216.

[56] Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, et al. Ttrl: Test-time reinforcement learning, 2025.

A Detailed positioning against nearby self-improvement methods

Table 3 expands the sketch in Section 2 into a family-by-family comparison. Two axes distinguish EvoEnv from each adjacent family: the *unit* that self-improvement synthesizes (one problem, one verifier, one trajectory, or a reusable environment), and the *stability* of the reward source against further updates of the same policy. A reward source that is policy-coupled, learned, or per-instance can still drive useful curricula, but it does not by itself give a durable, sample-many object that can be calibrated and reused.

B Anatomy of a verifiable environment

Figure 5 traces the data flow inside a single environment. Reference answers are computed by deterministic Python execution and reach the scorer through the code path; the solver sees only the rendered natural-language prompt. This is the picture that the validation pipeline (L1–L5 in Appendix C) must protect: every safeguard ensures that one of these arrows continues to mean what it appears to mean.

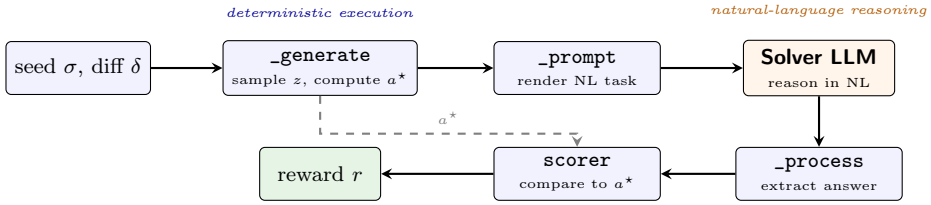

Figure 5 Anatomy of a verifiable environment. Reference answers flow through the code path from generation to scoring; the solver sees only the rendered natural-language task.

C Additional interface and validator details

The main text describes each candidate as a subclass of `VerifiableEnvironment`. Table 4 summarizes the validation layers used before a candidate can enter the pool. The layers are intentionally ordered from cheap syntactic checks to more expensive semantic and solver-facing tests, so early failures do not consume solver rollout budget.

D Illustrative planted subset-sum environment

The following minimal feasibility-check example illustrates the environment interface. The generator plants a solution and adds distractors. The solver must find any valid subset, while the scorer checks the advertised constraint by verifying both the target sum and multiset multiplicities.

E Semantic-review audit against an external reviewer

The same-policy reviewer in Section 3.2 only gates pool admission; its verdict never enters the generator reward (Eq. equation 5). Even so, the filter would be cosmetic if it agreed with a stronger reviewer no better than chance, so we audit it offline.

Setup. We take 79 environments that already cleared all five mechanical layers ($\ell(e) = 5$) from earlier training runs. Ground-truth labels come from a single review by GPT-5.4 (35 `has_bugs`, 44 `correct`); GPT-5.4 is not a perfect oracle but is a substantially more capable reasoner than our 4B in-training policy. The audited reviewer matches the deployed configuration: identical `Qwen3-4B Instruct-2507` weights, the same four-step structured prompt (data flow, instance trace, algorithm check, scorer check), $K_{\text{rev}} = 3$ samples at temperature 0.6, aggregated by *any-reject*.

```
from collections import Counter

class SubsetSumEnv(VerifiableEnvironment):
    prompt_template = ("Given the multiset {S}, find a nonempty "
                       "submultiset that sums exactly to {target}. "
                       "Output the elements.")

    def _generate(self, seed, difficulty):
        rng = Random(seed)
        n = 8 + difficulty * 4
        k = rng.randint(2, n // 2)
        solution = [rng.randint(1, 50) for _ in range(k)]
        self.target = sum(solution)
        self.S = solution + [rng.randint(1, 50) for _ in range(n - k)]
        rng.shuffle(self.S)
        self.ref_answer = sorted(solution)

    def _prompt_generate(self):
        return self.prompt_template.format(
            S=self.S, target=self.target)

    def _process(self, response):
        return sorted(parse_int_list(response))

    def scorer(self, parsed):
        if not parsed or sum(parsed) != self.target:
            return 0.0
        have, need = Counter(self.S), Counter(parsed)
        if any(need[x] > have[x] for x in need):
            return 0.0
        return 1.0
```

Figure 6 A feasibility-check environment. The stored reference answer is useful for validation, but the scorer accepts any subset satisfying the advertised constraint.

Agreement. Table 5 reports agreement under four aggregation rules. The deployed any-reject configuration attains $F_1 = 87.0\%$ ($P = 85.7\%$, $R = 88.2\%$, accuracy 88.6%), well above the always-`correct` baseline (55.7% accuracy, zero recall) and the always-`has_bugs` baseline (44% precision).

Failure modes. A balanced manual case study matches the quantitative picture. The reviewer reliably catches greedy-for-optimal substitutions, hardcoded or trivially-derivable reference answers, mathematical-property violations (e.g., a non-prime claimed prime), and most incorrect-recurrence bugs that surface on a small traced instance. It is weakest on *cross-method data flow* (a parameter written in `_generate` but never surfaced by `_prompt_generate` is hard to spot when methods are reviewed in isolation), *subtle data-generation pathologies* (e.g., randomly removing edges silently breaking connectivity assumptions), and *hallucinated edge cases* on otherwise correct code. These three modes account for most of the FN and FP in Table 5.

Why this gate suffices. Residual disagreement does not directly contaminate the generator's gradient: the verdict enters only Eq. equation 7, not Eq. equation 5. A false-positive correct candidate still receives its mechanical $Q_{\text{val}} + \gamma_t N_t$ reward and only loses its slot in the solver-training pool, while a false-negative buggy candidate is still filtered by the solver-relative pass-rate condition $0 < \hat{a}_m < 1$ before it can train the solver. The reviewer therefore only needs to be biased correctly relative to the external reference, not to match it perfectly.

F Seed environments

All runs start from the same ten initial seed environments. The seeds are executable interface examples and initial pool items, not external problem–answer datasets: they define the file layout, parameter conventions, prompt-rendering conventions, and scorer conventions that the generator imitates, while leaving the generator to synthesize new environment code during training. We deliberately picked the ten seeds to cover (i) different algorithmic categories, (ii) different oracle structures (exact match, element-wise match, partial-credit ratios,

and feasibility), and (iii) different difficulty knobs (problem size, value range, structural density, recursion depth). The exact ten-seed set is held fixed across all experiments.

Table 6 lists every seed individually. Each row corresponds to one Python class implementing the `VerifiableEnvironment` interface from Appendix B; the parameter knobs are the keys the generator fills in via `self.parameter` and the scorer column names the scorer family the seed exemplifies. We expand on the four scorer families below the table.

Why ten seeds suffice. Despite their small number, the ten seeds expose the generator to every interface pattern used downstream. They span seven algorithmic families, four scorer families, and structural primitives covering arrays, graphs, recurrences, optimal substructure (DP), and game-theoretic state. Four seeds (`EuclidGame`, `Fibonacci`, `MonotonicStack`, `SlidingWindow`) are adapted from competitive-programming archives (Luogu OI); the remaining six are written from scratch to balance scorer-family and structural coverage. Section 4.4 and Appendix G.1 report on the trajectories the generator follows away from this starting set during training.

G Generated-environment data audit

G.1 Qualitative examples within the 100-step run

The examples below illustrate the same shift at the environment level. We restrict this gallery to environments generated no later than step 100 so that it matches the ablation study in Section 4.4. Distances are Jaccard distances from the environment's tag set to the nearest one of the ten original seeds.

These examples are not meant to certify correctness by inspection; every accepted environment is still admitted through the validation and self-review pipeline described in Section 3.2. Their purpose is to make the distributional audit concrete: the generator moves quickly from the seed pool toward number-theoretic, modular, and sequence-structured problems that are solver-calibrated rather than trivially copied from RLVE.

H Limitations and Risks

Scope of the environment interface. This paper studies a controlled setting: zero-data RLVR for reasoning with deterministic Python environments. The current interface is best matched to tasks where an executable sampler, oracle, renderer, and scorer can be written compactly and validated automatically. This includes many algorithmic and feasibility-style tasks, but it does not establish the same claims for open-ended judgment, human preference modeling, long-horizon physical simulation, or interactive tool-use environments. Extending verifiable environment synthesis to those settings would require different grounding mechanisms and stronger safety checks.

Executable-code safety. A method that asks a model to synthesize executable environments must treat generated code as untrusted. Our implementation executes candidates in sandboxed subprocesses with restricted imports, resource limits, deterministic seeds, and wall-clock timeouts; candidates that violate the interface or execution constraints are rejected before pool admission. These safeguards are part of the method rather than optional engineering details. Any extension to richer tool-use, web, file-system, or embodied environments would require correspondingly stronger isolation and monitoring.

Broader impact. A positive implication of verifiable environment synthesis is that the training curriculum becomes more inspectable: accepted environments are concrete code artifacts rather than opaque pseudo-labels or hidden preference judgments. This can reduce dependence on proprietary post-training datasets and make reward sources easier to audit. At the same time, improving autonomous curriculum construction could amplify the capabilities of reasoning models in both beneficial and dual-use settings. We therefore view transparent environment release, sandboxed execution, and careful auditing of generated artifacts as necessary safeguards for follow-up work.

I Hyperparameters

Table 8 lists the headline hyperparameters reported in the main text; Table 9 reports the full configuration of the deployed training run, grouped by component.

Table 3 Detailed positioning of EvoEnv against nearby self-improvement methods. "Unit of synthesis" is what the method produces and reuses across updates; "reward stability" summarizes whether the accepted reward depends on the current policy's beliefs; "representative methods" are illustrative rather than exhaustive.

Family	Unit of synthesis	Reward / oracle source	Reward stability	Representative methods	Contrast with EvoEnv
Self-questioning / self-instruction	One problem, question, or instruction	Majority vote, self-reward, or learned judge	Policy-coupled	R-Zero, SeRL, SQLM, TTRL, EMPO, Intuitor, OpenSIR [4, 9, 13, 16, 48, 52, 56]	Adaptive curricula, but correctness moves with the learner.
Dual-play / co-evolution	Teacher, proposer, solver, or judge roles	Generated answers, judge feedback, or co-training signal	Mostly policy-coupled	PasoDoble, Socratic-Zero, MAE, EVOL-RL [5, 35, 49, 54]	Strong curriculum pressure; EvoEnv externalizes accepted reward into executable artifacts.
Per-instance executable self-play	Program/input/output triple or per-task verifier	Python execution for the generated task or instance	Frozen per instance, consumed after one rollout	AZR, SCA, SPC, Agent0, SWE-Playground, SSR [3, 37, 40, 51, 53, 55]	Grounded but per-instance; EvoEnv validates and reuses environment-level oracles.
Corpus-grounded self-play	Document-grounded task	Retrieved-text grounding	Stable per corpus	SPICE, Search Self-Play [19, 22]	Complementary grounding for knowledge tasks; EvoEnv uses code grounding for algorithmic tasks.
Game / fixed-rule self-play	Trajectory in fixed rules	Game outcome or fixed simulator	Stable; rules are given	SPIRAL, Self-RedTeam, LSP [15, 18, 20]	Rules are given; EvoEnv synthesizes the rule-equivalent artifact.
Hand-crafted environment RLVR	Environment	Human-written scorer	Frozen per environment	RLVE [46]	Same training unit, but manually authored rather than self-synthesized.
Tool / agent environment synthesis	Tool sandbox and scenarios	Rule-based trajectory validation	Frozen post-pipeline	EnvScaler, AutoWebWorld, InfiniteWeb, VeriEnv, AutoPlay, EmboMatrix [2, 17, 25, 31, 39, 50]	Pipeline-built agent sandboxes; EvoEnv is on-policy reasoning-environment co-evolution.
World-model simulators	Learned simulator state transitions	LLM as transition / world model	Depends on a learned simulator	DreamGym, UI-Simulator, WebEvolver [6, 8, 36]	Reward depends on a learned simulator; EvoEnv's reward depends on Python execution.
Open-ended evolutionary	Agent or environment co-population	Population-level fitness	Varies across the population	POET, Darwin Gödel Machine [34, 47]	Broader open-ended agenda; EvoEnv is solver-calibrated, on-policy, and validated.
Foundational self-improvement (lineage)	Generated rationales, self-generated preferences, or filtered traces	Verifier, learned reward model, or self-preference	Mostly policy-coupled	STaR, ReST-EM, SPIN, Self-Rewarding LMs, V-STaR [7, 12, 30, 43, 44]	Bootstraps from labelled seeds; EvoEnv extends this lineage to environment-level objects.
EvoEnv (this work)	**Reusable verifiable environment**	**Frozen Python execution**	**Frozen between pool updates**	—	—

Table 4 Validator layers.

Layer	Check
L1	Parseable Python; expected class and methods exist.
L2	Class instantiates; generation, prompt rendering, parsing, and scoring run on multiple seeds and difficulty values.
L3	Determinism under identical seeds; prompt, state, and reference answer are stable.
L4	Non-triviality across seeds and difficulty values; prompts and answers vary.
L5	Scorer contract: reference answers score positively; perturbations, malformed answers, and type mismatches do not; parser does not leak hidden references.

Table 5 Agreement between the same-policy reviewer (Qwen3-4B-Instruct, $K_{\text{rev}} = 3$, $T = 0.6$) and GPT-5.4 on the 79 environments with parseable verdicts (35 buggy, 44 correct under GPT-5.4).

Aggregation	Acc.	Prec.	Recall	F_1	(TP, FP, TN, FN)
$K = 1$ (single review)	65.8%	75.0%	34.3%	47.1%	$(12, 4, 40, 23)$
$K = 3$, majority	65.8%	78.6%	31.4%	44.9%	$(11, 3, 41, 24)$
$\mathbf{K = 3}$, **any-reject**	**88.6%**	**85.7%**	**88.2%**	**87.0%**	$(\mathbf{30}, \mathbf{5}, \mathbf{40}, \mathbf{4})$
$K = 3$, all-reject	63.3%	87.5%	20.0%	32.6%	$(7, 1, 43, 28)$

Table 6 The ten initial seed environments. Each row gives the environment class name, the algorithmic category it represents, the task asked of the solver, the difficulty knobs surfaced to the generator (read from `self.parameter`), and the family of scorer the seed exemplifies (defined below the table).

#	Seed class	Category	Task	Difficulty knobs	Scorer family
1	`Sorting`	Array / ordering	Sort N integers in ascending order.	N, value range	Element-wise
2	`SlidingWindow`	Array / deque	Output the minimum of every contiguous size-K window.	N, K	Element-wise
3	`MonotonicStack`	Array / stack	For each i, count $j > i$ with $A[i] > \max(A[i{+}1..j])$, then sum.	N	Exact match
4	`Knapsack`	0/1 DP / optimization	Pick items maximizing total value under a weight budget.	N, $W_{\max}$, value range	Partial-credit ratio
5	`SubsetSum`	Combinatorial / feasibility	Pick a subset of indices summing exactly to a planted target.	N	Feasibility
6	`BoundedIntervalIntersection`	Combinatorial / counting	Count non-empty interval subsets whose intersection has length $\geq K$.	N, K	Partial-credit ratio
7	`Bridge`	Graph / connectivity	Find every bridge edge of an undirected graph.	N, component count, edge density	Partial-credit ratio
8	`EuclidGame`	Game theory / number theory	Determine the optimal-play winner of the Euclid subtraction game on (X, Y).	$\max(X, Y)$	Exact match (binary)
9	`Fibonacci`	Linear recurrence	Compute $A[N] \bmod m$ for $A[n] = P \cdot A[n{-}1] + Q \cdot A[n{-}2]$.	N, modulus	Exact match
10	`RecursiveFunction`	Recursion / Ackermann-like	Compute $f(M, N)$ under a three-case recurrence with self-nested calls.	$\max(M, N)$	Exact match

Table 7 Representative generated environments from the 100-step run. Prompts are shortened to their first clause for readability.

Class	Step	d_{seed}	# tags	Prompt sketch
`PrimePartition`	1	0.73	8	Given a list of positive integers, count or decide prime-constrained partitions.
`PrimePairProductSum`	10	0.71	6	Given N and a set of distinct prime numbers, optimize a product/sum expression.
`PrimeGCDChain`	60	0.61	17	Combine a sequence of values with GCD and prime-chain constraints.
`SubsetCycleGCD_PrimeChain`	69	0.65	16	Fuse subset selection, cyclic structure, and GCD/prime constraints.
`CycleDistanceSum`	83	0.75	9	Reason over distances induced by a cycle-like combinatorial structure.

Table 8 Training and evaluation hyperparameters reported in the main text.

Hyperparameter	Value
Generator advantage scale w_{gen}	0.3
Solver test budget for candidate calibration	$m = 8$
Semantic self-review samples	$K_{\text{rev}} = 3$ with any-reject aggregation
Prompt/code novelty mixture λ	0.5
Original-seed floor $\rho_{\min}$	0.2
Training steps	100 steps
Evaluation context	up to 128k
Evaluation temperature / top-p	0.6 / 0.95

Table 9 Full configuration of the deployed 4B-Instruct training run. Generator group, solver group, novelty, pool, review, and sandbox settings are shared across backbone variants; only the batch sizes and context lengths in the optimizer block are rescaled when scaling up to 8B.

Block	Hyperparameter	Value
Generator group	Number of distinct generator prompts per step	16
	GRPO group size M (responses per generator prompt)	8
	Generator sampling temperature	1.0
	Generator max response length	8k tokens (rollout side)
Solver group	Solver batch size B (`data.train_batch_size`)	64
	PPO mini-batch size	64
	GRPO group size (`rollout.n`, responses per prompt)	8
	Solver sampling temperature	1.0
	Solver max prompt / response length	8k / 16k tokens
Novelty schedule	Novelty weight bounds $(\gamma_{\min}, \gamma_{\max})$	(2.0, 5.0)
	Activation thresholds $(\tau_{\text{low}}, \tau_{\text{high}})$	(0.45, 0.65)
	Pool-admission gate τ_{gate}	0.80
	EMA decay for $\bar{s}_t$	$\alpha_{\text{ema}} = 0.6$ (init 0.5)
	Embedding model	`all-MiniLM-L6-v2` (frozen)
	Prompt/code mixture λ	0.5
Calibration & admission	Calibration instances per L5 candidate m	up to 8 (Eq. equation 1)
	Solver responses per calibration instance	1
Pool rotation	Pool max epochs per environment	5
	Pool rotation cadence	every 10 training steps
	Original-seed floor $\rho_{\min}$	0.2
	Initial pool	ten seeds (Appendix F)
Self-review	Independent reviews K_{rev}	3
	Aggregation rule	any-reject
	Review temperature	0.6
	Review max response length	8k tokens
RL objective & optimizer	Algorithm	GRPO
	PPO clip ratio (low / high)	0.2 / 0.28
	Dual-clip constant c	10.0
	KL loss coefficient β	1×10^{-3}
	Entropy coefficient	0
	Optimizer	AdamW
	Learning rate / warmup / weight decay	5×10^{-7} / 5 steps / 0.1
	Gradient clip	1.0
Sandbox & code	Validation subprocess wall-clock timeout	30 s per layer
	Allowed imports (enforced via generator prompt)	`random`, `math`, `collections`, `itertools`, `heapq`, `bisect`, `functools`, `re`, `typing`
	Disallowed imports	any third-party (e.g. `numpy`, `networkx`, `scipy`)
	Code extraction	longest python fenced block