SCALER: Synthetic Scalable Adaptive Learning Environment for Reasoning

Anonymous ACL submission

Abstract

Reinforcement learning (RL) offers a principled way to enhance the reasoning capabilities of large language models, yet its effectiveness hinges on training signals that remain informative as models evolve. In practice, RL progress often slows when task difficulty becomes poorly aligned with model capability or when training is dominated by a narrow set of recurring problem patterns. To jointly address these issues, we propose **SCALER** (**S**ynthetic s**C**alable **A**daptive **L**earning **E**nvironment for **R**easoning), a framework that sustains effective learning signals through adaptive environment design. SCALER introduces a scalable synthesis pipeline that converts real-world programming problems into verifiable reasoning environments with controllable difficulty and unbounded instance generation, enabling RL training beyond finite datasets while preserving strong correctness guarantees. Building on this, SCALER further employs an adaptive multi-environment RL strategy that dynamically adjusts instance difficulty and curates the active set of environments to track the model's capability frontier and maintain distributional diversity. This co-adaptation prevents reward sparsity, mitigates overfitting to narrow task patterns, and supports sustained improvement throughout training. Extensive experiments show that SCALER consistently outperforms dataset-based RL baselines across diverse reasoning benchmarks and exhibits more stable, long-horizon training dynamics.

1 Introduction

Reinforcement learning (RL) has become a key post-training paradigm for enhancing the reasoning capabilities of large language models (LLMs) (OpenAI et al., 2024; DeepSeek-AI et al., 2025). By optimizing a policy under explicit, verifiable rewards, RL can sharpen decision making, improve long-horizon credit assignment, and expand the model's reasoning frontier (Shao et al., 2024; Yu et al., 2025; Hu, 2025). However, scaling RL for reasoning is often bottlenecked not by the optimizer itself, but by the availability of continuously effective reward signals throughout training (Razin et al., 2023, 2025; Zhang et al., 2025).

In this work, we argue that the training signal, for RL to continue improving LLMs, should remain effective in two complementary senses. First, problems should stay near the model's current capability boundary during training (Parashar et al., 2025b; Chen et al., 2025d), i.e., remain neither trivial nor unsolvable. When the model mostly sees easy problems, learning saturates; when it mostly sees overly difficult problems, exploration becomes unproductive and the reward becomes sparse. Second, the training distribution should retain sufficient diversity over time (Li et al., 2025). Even if difficulty is well matched, repeatedly interacting with a narrow task distribution can lead to overfitting to a limited set of patterns, weakening generalization and reducing exploration. Note that difficulty variation is not equivalent to diversity. Even if a single environment can generate infinitely many problems by scaling parameters (e.g., longer arrays or larger graphs), it still shares a limited set of templates and failure modes, so learning can plateau once those patterns are mastered.

To this end, we propose **SCALER** (**S**ynthetic s**C**alable **A**daptive **L**earning **E**nvironment for **R**easoning), a system that combines scalable reasoning environment synthesis with adaptive multi-environment reinforcement learning. For the first limitation, SCALER provides a way to generate verifiable tasks at scale with controllable difficulty. We develop a synthesis pipeline that programmatically converts real-world programming problems into reasoning environments with (i) verifiable interaction via deterministic oracles and unit tests, (ii) controllable difficulty via explicit scale parameters, and (iii) unbounded instance generation within each environment through randomized testcase

generation. The pipeline automatically extracts meta-information, validates testcase generators via breadth/depth checks. This enables scaling training beyond finite reasoning datasets or a small set of hand-crafted environments, while retaining strong correctness guarantees.

On the other hand, SCALER designs an adaptive multi-environment RL framework to adapt both instance difficulty and environment selection so that training continues to encounter informative challenges as the model improves. Specifically, within each environment, an online difficulty controller adjusts scale parameters based on on-policy rollout accuracy to keep sampled instances near a target success rate, thereby tracking the model's capability frontier and avoiding degenerate "all-correct" or "all-wrong" regimes. Across environments, an environment curation mechanism maintains an active set of environments and replaces those whose learning signal has saturated (e.g., difficulty no longer increases or the environment becomes consistently trivial/unlearnable). This realizes an environment-level effectiveness. As the model and environments co-evolve, the marginal learning benefit can diminish, so continuously refreshing the active set helps preserve novelty and sustained learning signals.

Our contributions can be summarized as follows:

- We highlight the importance of difficulty and diversity controllable environments for scaling RL post-training in reasoning.
- We propose **SCALER** that combines verifiable, difficulty-controllable environment synthesis with adaptive multi-environment RL.
- Extensive experiments demonstrate SCALER yields consistent improvements across diverse reasoning benchmarks and exhibits more sustained training dynamics than dataset-based RL baselines under comparable budgets.

2 Related Work

Data-centric RL. A data-centric line of work improves the scalability of RL for LLM reasoning by continuously expanding the training distribution with synthetic data (Wu et al., 2025; Setlur et al., 2024) and self-play (Fang et al., 2025; Liang et al., 2025; Wang et al., 2025b; Chen et al., 2025a). On the synthetic-data side, works like Synthetic Data RL (Guo et al., 2025b) and SWiRL (Goldie et al., 2025) generate task-specific supervision from question–answer pairs to multi-step reasoning and tool-use trajectories, and then apply RL on the resulting synthetic corpus. On the self-play side, recent methods produce interactions and hard cases via adversarial games or role-based play, enabling continual data self-generation for policy improvement (Zhao et al., 2025; Liu et al., 2025a). However, these approaches are easy to encounter bottlenecks: as the max difficulty of tasks is bounded by the generator agent (Chae et al., 2025), the data may drift out of sync with the evolving policy, collapsing into instances that are too easy or too hard and thus weakening learning signals in later stages. In contrast, SCALER applies difficulty controller and environment curation mechanism on synthesized verifiable reasoning environment to keep training informative throughout RL.

Difficulty-aware RL. To mitigate vanishing learning signals, difficulty-aware RL often employs curriculum learning (Team, 2025a; Shi et al., 2025; Chen et al., 2025b) or difficulty scheduling (Wang et al., 2025a; Chen et al., 2025c). Curriculum-based method trains models from easy to hard (Liu et al., 2025b; Parashar et al., 2025a), keeping effective learning signals in training process. However, curricula designed over static dataset faces the challenge of being coarse-grained and hard to design. This motivates environment-based formulations where difficulty of each environment can be monitored and adjusted online (Guo et al., 2025a). Reasoning Gym (Stojanovski et al., 2025) offers a suite of procedurally generated, verifiable reasoning environments with tunable difficulty, while RLVE (Zeng et al., 2025b) further adapts the difficulty distribution online as the policy improves. Despite enabling difficulty tracking and adaptation, existing environment suites are largely hand-engineered and limited in environment diversity at scale. SCALER addresses this by automatically synthesizing a large and diverse set of reasoning environments with verifiable oracles and controllable difficulty, supporting sustained multi-environment RL.

3 SCALER

To enable the model to explore, adapt, and generalize in multiple dynamic environments, we design **SCALER** with two components: (i) an adaptive multi-environment training framework that learns through interaction with a set of verifiable environments, and (ii) a systematic synthesis pipeline that converts programming problems into difficulty-controllable reasoning environments. Specifically,

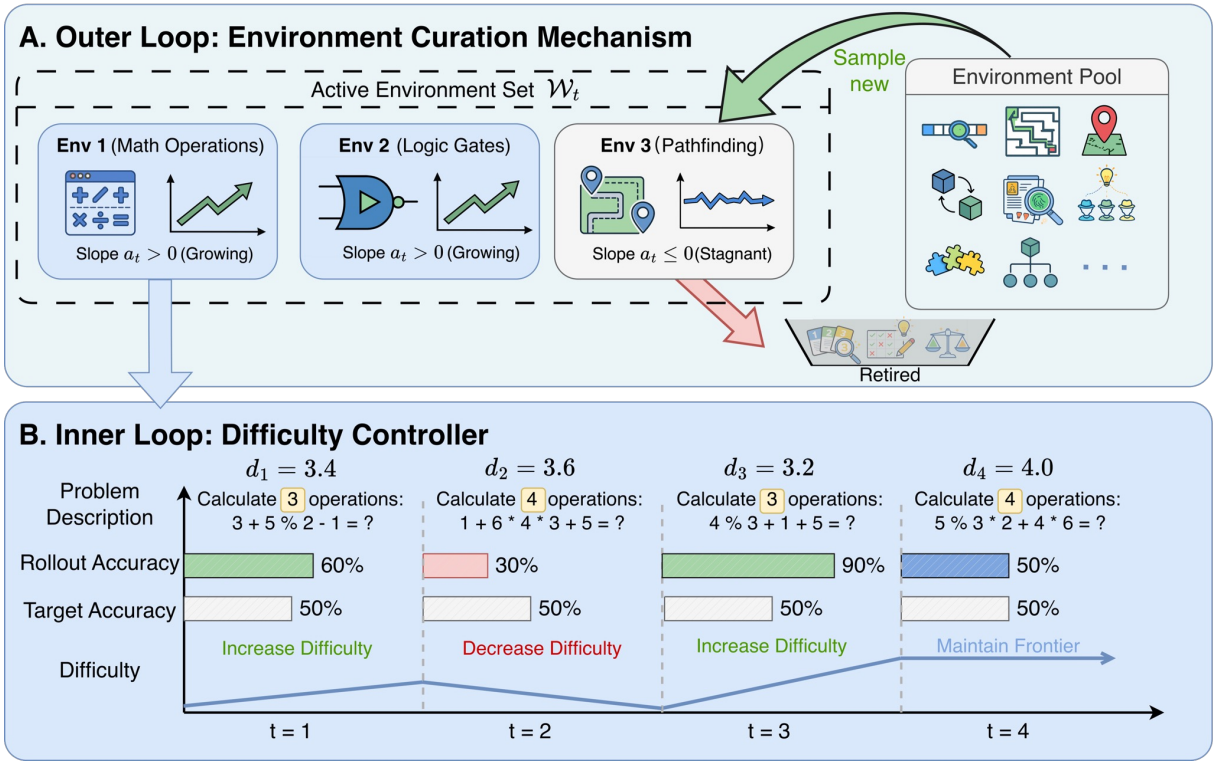

Figure 1: An illustrative training example of **SCALER**. The model interacts with a set of active environments: instances are used for training, online accuracy updates the in-environment difficulty controller, and environments whose learning signal saturates are retired and replaced by new environments via the curation mechanism.

we first introduce the learning strategy over environments, including in-environment difficulty controller and the environment curation mechanism (§3.1), then describe the environment synthesis process from programming problems (§3.2).

3.1 Multi-environment Training Framework

We first illustrate the SCALER environment to facilitate understanding. We define an environment as a context-conditioned, parameterized problem space in which individual instances can evolve through controllable parameters. For example, as shown in the math-operations environment in Figure 1, all problems share a common contextual description, *calculate N operations*, while problem difficulty is modulated by the scale parameter N. More illustrative cases are provided in Appendix B.

To fully unleash the potential of these difficulty-controllable reasoning environments, we adopt two complementary learning strategies: (i) Difficulty Controller. In environments with explicit verifiability, we dynamically adjust the environment's scale parameters according to the online accuracy, which provides continuous and stable learning signals. (ii) Environment Curation Mechanism. For general environments, we introduce environment curation mechanism within which the model is trained exclusively on the current active set of environments. As the model and the environment gradually co-adapt and the learning signal diminishes in marginal utility, the current environment is retired and replaced with a newly sampled one.

The detailed implementation of SCALER is provided in Algorithm 1.

3.1.1 Difficulty Controller

A key goal of SCALER is to maintain training at the agent's capability frontier by continuously sampling instances near the boundary of what the current policy can solve. To achieve this, difficulty is dynamically adjusted within each environment based on current accuracy.

Specifically, the difficulty of the instances is characterized by the array length or the number of edges in the graph and we discretize the difficulty into distinct difficulty levels. Let $\mathrm{acc}_t \in [0, 1]$ denote the average accuracy over k sampled instances at step t in a given environment, and let $\tau \in [0, 1]$ denote the target accuracy. The continuous difficulty score $d_t \in \mathbb{R}$, initializing $d_0 = 0$, evolves according to:

$$d_{t+1} = \mathrm{clip}(d_t + \beta \cdot (\mathrm{acc}_t - \tau),\ 0,\ D), \quad (1)$$

Algorithm 1 SCALER Training Framework

Require: Policy π_θ, Environment Pool $\mathcal{E}_{pool}$, Target accuracy τ, Step size β, Active set size M

Initialize: Sample initial active set $\mathcal{W} \leftarrow \{e_1, \ldots, e_M\} \subset \mathcal{E}_{pool}$,Difficulty $d \leftarrow d_{min}$ for all $e \in \mathcal{W}$
for step $t = 1, 2, \ldots, T_{max}$ **do**
  $\mathcal{D}_{batch} \leftarrow \emptyset$
  for each environment $e \in \mathcal{W}$ **do**
    Sample instances x from e based on difficulty d
    Collect trajectories using π_θ and calculate accuracy acc_t
    $\mathcal{D}_{batch} \leftarrow \mathcal{D}_{batch} \cup$ Trajectories
    Update difficulty d for this environment ▷ Eq. 1
  Update π_θ with $\mathcal{D}_{batch}$
  for each environment $e \in \mathcal{W}$ **do**
    Calculate slope a_t^e over last K_{slope} steps ▷ Eq. 3
    if $a_t^e \leq 0 \vee \mathrm{acc} = 0$ for K_{zero} steps $\vee\ d = D^e$ for K_{sat} steps **then**
      Retire e: $\mathcal{W} \leftarrow \mathcal{W} \setminus \{e\}$; $\mathcal{E}_{pool} \leftarrow \mathcal{E}_{pool} \cup \{e\}$
      Sample new $e_{new} \sim \mathcal{E}_{pool}$
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{e_{new}\}$

where $\beta > 0$ controls the adaptation rate and $D \in \mathbb{N}$ is the maximum discrete difficulty level in the environment. This update rule increases difficulty when $\mathrm{acc}_t > \tau$ and decreases it otherwise, thereby maintaining the training distribution near the agent's capability boundary.

For the next sampling step, while d_{t+1} is real-valued, the environment can only generate k instances at integer-valued levels in $\{0, 1, \ldots, D\}$. To approximate $d_{t+1} \in \mathbb{R}$ under this integer constraint, we construct a multiset of k integers whose mean closely matches d_{t+1}. Specifically, let $\ell = \lfloor d_{t+1} \rfloor$ and define $h = \mathrm{round}\big(k(d_{t+1} - \ell)\big)$. We assign h instances the value $\ell + 1$ and the remaining $k - h$ instances the value ℓ, thereby balancing rounding up and rounding down; for example, when $d_{t+1} = 2.3$ and $k = 10$, this yields three instances at level 3 and seven at level 2, whose average equals 2.3.

The proposed controller naturally tracks model competence. As the model improves, online accuracy rises above the target and difficulty is pushed upward to stay near the performance boundary. Conversely, if the model forgets or the policy distribution shifts, accuracy drops and the controller reduces difficulty, recovering usable learning signals without manual retuning.

3.1.2 Environment Curation Mechanism

To keep sufficient diversity during training and avoid a "tourist" learning pattern where the model randomly explores a variety of environments without sustained progress, SCALER introduces the environment curation mechanism which maintains an active set of environments and restricts training to this dynamic set, continuously refreshing it to avoid spending budget on environments that no longer provide meaningful learning signals.

Let $\mathcal{W}_t = \{e_1, \ldots, e_{|\mathcal{W}|}\}$ denote the active environment set at training step t. At each step, instances are sampled only from environments in $\mathcal{W}_t$ and used for policy optimization. Environments are periodically evaluated at every step and may be retired from the set and replaced by newly synthesized environments.

For each environment e, let d_t^e denote its difficulty at step t. To detect stalled progress, we argue that when the recent difficulty-step slope is less or equal to zero, the learning on this environment is saturable and should be retired. Specifically, we estimate the slope by fitting a least-squares line over the last K_{slope} steps, defining the index set $T = \{t - K_{slope} + 1, \ldots, t\}$ and fitting

$$d_i^e \approx a_t^e \cdot i + b_t^e,\ i \in T, \tag{2}$$

where the slope a_t^e is given in closed form by

$$a_t^e = \frac{\sum_{i \in T}(i - \bar{i})(d_i^e - \bar{d}^e)}{\sum_{i \in T}(i - \bar{i})^2}, \tag{3}$$

with $\bar{i} = \frac{1}{|T|}\sum_{i \in T} i$ and $\bar{d}^e = \frac{1}{|T|}\sum_{i \in T} d_i^e$. An environment is retired if $a_t^e \leq 0$, indicating that difficulty is temporarily saturated in that environment.

To accelerate retirement decisions, two additional heuristics are applied. Let $\mathrm{acc}_t^e \in [0, 1]$

denote the accuracy at step t and let D^e be the maximum difficulty of environment e. An environment is retired early if it is unlearnable, i.e., $\text{acc}^e = 0$ for K_{zero} consecutive steps, or saturated, i.e., $d^e = D^e$ for K_{sat} consecutive steps.

When an environment is retired, it is replaced by a newly synthesized environment sampled from the environment pool, keeping the set size fixed. Such retirement is temporary, as it targets situations where the model has reached a plateau in learning within the current environment, rendering further progress unproductive. However, as training continues and the model's capability boundary improve, there remains potential for continued learning within these environments. Therefore, retired environments are reintegrated into the environment pool and will be resampled in future iterations, offering renewed opportunities for the model to explore and adapt.

In general, the environment curation mechanism supports sustained learning within multiple environments while preventing training from stagnating on environments that have become either trivial or unproductive.

3.2 From Code Generation to Reasoning Environments

Our multi-environment training framework continuously tracks the model's capability frontier while maintaining data diversity and freshness. This requires a steady supply of environments that are difficulty-controllable and automatically verifiable. We therefore introduce an environment synthesis pipeline that converts real-world programming problems into such environments.

Accordingly, our synthesis pipeline is organized into three components:

- **Meta information extraction** (§3.2.1). Ensures rewardable and unambiguous supervision by selecting problems with well-defined, verifiable outputs, mitigating output-format mismatch and reward hacking.
- **Testcase generation and verification** (§3.2.2). Provides a stream of diverse, valid, and automatically verifiable instances, addressing the mismatch between static problem statements and stochastic environment inputs.
- **Heuristic difficulty calibration** (§3.2.3). Defines an executable difficulty range under context and runtime budgets, addressing the scale mismatch between real-world inputs and agentic prompting.

3.2.1 Extracting Meta Information

To offer high-quality programming problems for reasoning environment synthesis and mitigate the risk of reward hacking, we adopt a prompt-based method to extract key meta-information from candidate problems, followed by a rule-based filtering process. The complete extraction prompt is provided in Appendix A.1.

For each programming problem p, we extract a metadata tuple $\texttt{meta}(p)$ comprising (i) scale parameters that characterize complexity-related constraint variables, and (ii) output requirements that describe the output type and whether the correct output is unique. Accordingly, we discard ill-formed problems and retain only those with a unique output whose type lies in $\{\texttt{number}, \texttt{array}, \texttt{string}\}$.

3.2.2 Generating and Verifying Testcases

To sample different instances with fixed extracted scale parameters from one specific environment, we construct a testcase generator agent that takes a target scale parameter configuration and produces an input whose content is randomized but conforms to the original problem specification. We provide the full prompting details in Appendix A.2. Since the environment relies on the synthetic agent, we validate generator functions with two complementary checks.

Breadth check. We sample diverse scale parameters and corresponding inputs , when following (Fu et al., 2025), each instance is evaluated by multiple independent ground-truth solutions. Consistency across solutions is required, which simultaneously (i) detects malformed generator function and (ii) revalidates the assumption that the output is unique.

Deep check. For a fixed scale parameter configuration, we call generator function multiple times and compare the resulting ground-truth outputs across generated instances. Based on the numbers and max size requirement of output clustering, we enforce sufficient diversity to prevent the training process from overfitting to a narrow pattern distribution or exploiting reward hacking.

3.2.3 Calibrating Heuristic Difficulty Levels

As inputs for real-world programming problems can be millions, which is not suitable for the prompt used in agentic training, we aim to re-define the difficulty range for each environment that is both scalable and executable. Apparently, the effective

maximum difficulty is bounded by two practical constraints: (i) the maximum prompt length accepted by the policy model, and (ii) the execution time limit enforced by the original problem setting.

To estimate the largest feasible scale parameter configuration $\mathbf{s}_{\max}$ under the prompt-length constraint, a binary search is performed over a single global scale factor. Feasibility is determined by whether the testcase fits within the context window while respecting the fixed execution time limit.

Given $\mathbf{s}_{\min}$ and $\mathbf{s}_{\max}$, the environment defines a finite set of difficulty levels by discretizing the scaling range. To keep the number of levels within a reasonable budget, the discretization strategy depends on the span of each parameter: when the range is small, levels follow an arithmetic progression; when the range is large, levels follow a geometric progression. This heuristic yields a compact but expressive difficulty ladder, enabling smooth scheduling while preserving meaningful granularity across scales.

4 Experiments

SCALER addresses two key challenges: (i) providing a synthesis pipeline that systematically generates environments with infinite data generation capabilities, and (ii) developing a multi-environment training framework, which enables sustaining model improvement through adaptive learning across diverse environments. To valid the efficiency of our approach, experiments are organized around three research questions (RQs):

- RQ1: How does SCALER compare with dataset-based training baselines under comparable training budgets?
- RQ2: How does performance vary as the number of environments increases?
- RQ3: Are all components of SCALER Necessary for their gains?

4.1 Experimental Setup

Synthesis Pipeline Settings. We use CodeContests (Li et al., 2022) as the seed dataset and GLM-4.6 (Zeng et al., 2025a) as the data-synthesis agent. For code execution, we use SandboxFusion (Bytedance-Seed-Foundation-Code-Team et al., 2025). After filter, we obtain the subset consisting of 4973 programming problems and synthesize 2739 SCALER environments. Specifically, we offer environment cases in Appendix B.

Training setup. All experiments start from Qwen3 (Team, 2025b)series, and we use Qwen3-1.7B-base and Qwen3-4B-base as the policy model. For the environment curation mechanism, we set $K_{slope} = 10$ and $K_{zero} = K_{sat} = 5$. The training batch size equals the environment set size: at each optimization step, one problem is sampled from each of the 64 environments according to the environment-specific difficulty controller, resulting in 64 prompts per step. Reinforcement learning is performed with GRPO (Shao et al., 2024). For each prompt, n_{resp}=8 responses are sampled. Training-time decoding uses temperature T=1.0 and top_p=1.0. The prompt length budget is capped at 4096 tokens and the response length budget at 8192 tokens. Additional hyperparameters and infrastructure details are deferred to Appendix C.

Evaluation protocol. Performance is evaluated on five benchmarks: AIME24 (Zhang and Math-AI, 2024), AMC23 (Art of Problem Solving, 2023), MATH-500 (Lightman et al., 2023), MMLU-Pro (Wang et al., 2024), and BBEH (Kazemi et al., 2025). Results are reported as avg@16 for AIME24, AMC23 and MMLU-Pro, avg@1 for the remaining benchmarks. Unless otherwise specified, decoding uses temperature $T = 0.6$ and $top\_p = 0.95$.

4.2 RQ1: Environment-based training is better than Dataset-based training

A common paradigm for improving reasoning is to scale up training on static datasets, such as curated corpora with verified solutions or collections augmented by teacher-generated answers. However, static corpora provide limited effective learning signals, where agents must repeatedly adapt to new environments and generalize beyond previously seen distributions. To evaluate whether interactive environment-based learning better supports the improvement of model capability, this RQ compares SCALER with two dataset-based baselines under comparable training budgets: MATH (Hendrycks et al., 2021), a curated corpus with verified solutions, and DeepMath (He et al., 2025), a dataset of teacher-generated solutions.

Table 1 shows that both dataset baselines substantially improve over the Qwen3-1.7B-Base and Qwen3-4B-Base, with particularly large gains on math-centric benchmarks. In contrast, SCALER achieves the best overall average and delivers con-

Model	MATH-500	AMC23	AIME24	MMLU-Pro	BBEH	AVG
Qwen3-1.7B-base	59.6	29.21	3.33	33.30	3.26	25.74
+DeepMath-103K	73.6	47.97	14.58	49.64	9.56	39.07
+MATH-7.5k	75.6	**50.78**	**15.20**	46.78	6.08	38.89
+SCALER	**75.8**	49.53	12.91	**50.89**	**11.74**	**40.18**
Qwen3-4B-Base	66.4	44.70	8.75	51.60	8.10	35.91
+DeepMath-103K	**86.6**	65.60	22.29	68.03	12.82	51.08
+MATH-7.5k	86.2	70.63	24.16	69.19	10.00	52.04
+SCALER	84.4	**75.0**	**27.29**	**70.00**	**14.56**	**54.25**

Table 1: Performance comparison of Dataset-based RL and SCALER on five reasoning benchmarks. AVG is the unweighted mean over the five benchmarks. The highest performance is **bolded**.

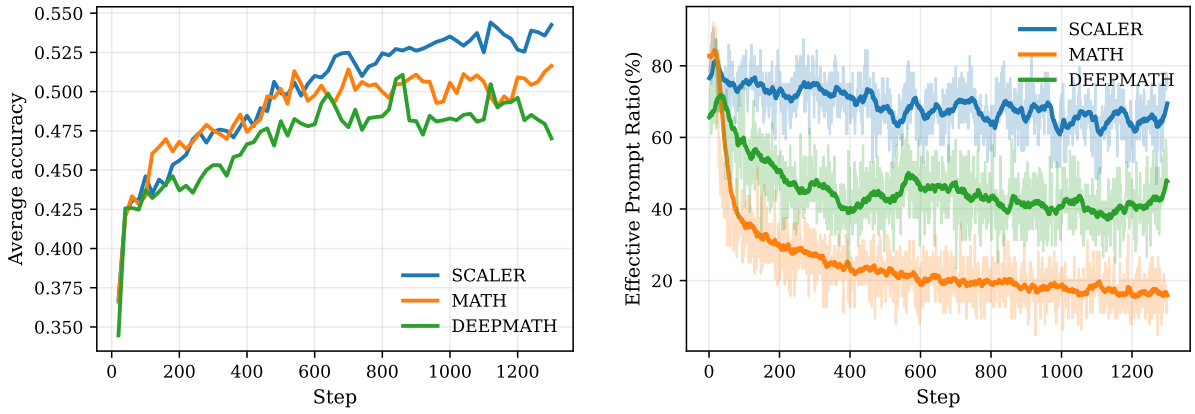

Figure 2: Left: average performance across the five evaluation benchmarks during Qwen3-4B-base training, comparing dataset-based baselines (MATH, DeepMath) and SCALER. Right: effective sampling statistics under SCALER, indicating that most sampled instances remain near the model's capability boundary.

sistent improvements across all five evaluations, including three math benchmarks, MMLU-Pro and BBEH, suggesting stronger transfer beyond narrowly curated math data.

The training dynamics in Figure 2 provide additional evidence for this gap. Figure 2 (left) shows that SCALER not only reaches a higher level, but also exhibits a more sustained improvement trend: the aggregated evaluation performance continues to rise for more than 1,000 training steps, whereas dataset-based baselines plateau earlier. Importantly, this behavior is supported by boundary-focused sampling: Figure 2 (right) indicates that SCALER keeps a higher effective sampling rate than dataset-based baselines, where most prompts remain neither trivial nor intractable. By continuously providing instances near the model's current capability boundary from SCALER environments, SCALER sustains informative reward during RL, mitigates premature saturation, and enables longer-horizon performance gains.

4.3 RQ2: Scaling environment size leads to incremental performance gains

SCALER introduces the automatic environment synthesis pipeline which significantly reduces labor costs. This RQ examines how model performance changes as the number of environments increases, with results summarized in Figure 3.

We conduct the experiment by randomly sample 8, 64, 512 environments from 2739 SCALER environments, where the larger size of environments always contain the smaller size one. As shown in Figure 3, increasing the number of environments from 8 to 2739 leads to incremental performance improvements. The model benefits from encountering a greater diversity of tasks, which allows it to maintain consistent learning progress. With exposure to a wider variety of challenges and environments facilitating the development of broader reasoning skills, scaling the number of environments enables the model to continuously adapt and enhance its reasoning capabilities.

It is worth noting that even with a smaller num-

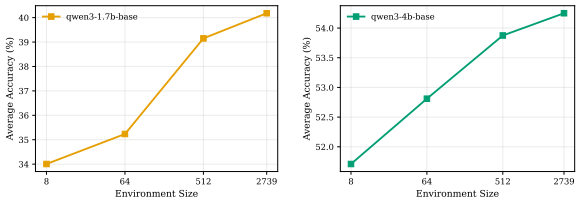

Figure 3: Accuracy improvements for both Qwen3-4B-base and Qwen3-1.7B-base as environment size increases. Both models show a consistent increase in performance with larger environment sizes.

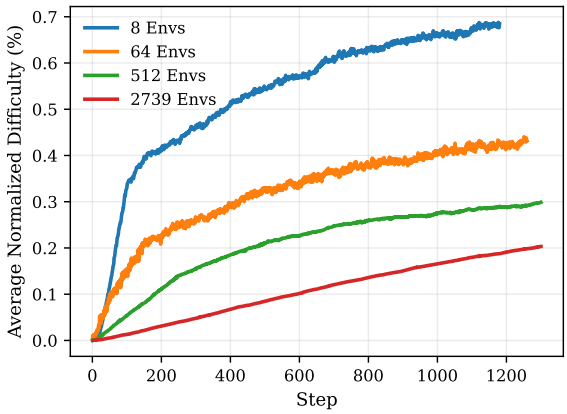

Figure 4: Training dynamics of Qwen3-4B-Base with different numbers of environments. The plot shows that even with smaller environments, the model continues to learn with increasing difficulty levels.

ber of environments, the model continues to engage in ongoing exploration, with increasing difficulty levels within each individual environment, as illustrated in Figure 4. The key insight is that scaling the number of environments with fixed training budgets, inherently involves a trade-off between task difficulty and diversity. Striking right balance is critical: focusing solely on increasing difficulty may result in limited improvements, while excessive diversity without a proper difficulty controller, i.e. DeepMath dataset with 103k samples, does not necessarily lead to optimal performance.

4.4 RQ3: All components of SCALER are necessary for its gains

This experiment studies whether SCALER's improvements require its two core components in multi-environment training framework: adaptive difficulty controller and environment curation mechanism.

Two ablations based on Qwen3-4B-base are compared against the full SCALER system in Figure 5. Removing the curation mechanism disables

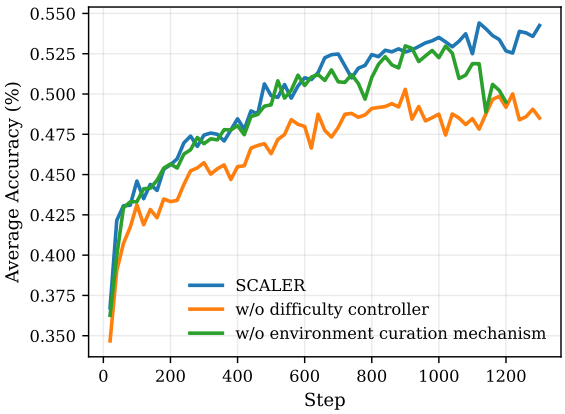

Figure 5: Ablation study of **SCALER** on Qwen3-4B-Base. Validation accuracy over training steps for the full system and two variants that remove difficulty controller or the environment curation mechanism. Both components contribute to stronger and more sustained performance improvements.

environment replacement, forcing training to random sample every environment. To model the ablation of difficulty controller, we replace the boundary tracking with random difficulty sampling, selecting 5 instances per environment. This modification reduces the consistency of the supervision signal, as the environment selection is no longer guided by a difficulty-based approach.

Both ablations lead to lower performance than the full system, indicating that difficulty controller and the curation mechanism each make a material contribution. Qualitatively, difficulty controller prevents training from drifting into regimes that are too easy or too hard, while the curation mechanism promotes sustained learning within learnable environments and mitigates shallow adaptation.

5 Conclusion

In this paper, we proposes a novel approach SCALER containing two key components: (i) reasoning environment synthesis pipeline, (ii) multi-environment training framework. Through our synthesis pipeline to generating a variety of difficulty-controllable environments , our approach offers a platform for research community to explore the impact of properties of environments on RL training. Moreover, the multi-environment training framework ensures the difficulty of instances consistently matches the model's capabilities and maintains diversity and freshness, jointly preserving the **continuously effective** reward signals to improve model capabilities in long horizon.

Limitations

While our work provides a comprehensive analysis of SCALER's ability to improve model performance through dynamic environment synthesis and multi-environment training, there are several limitations that warrant further exploration:

- **Exploration of Environment Internal Attributes**: Our study primarily focuses on scaling environment size and its components. However, the internal characteristics of environments, such as the richness of context, intrinsic difficulty, and other environment-specific properties, have yet to be thoroughly investigated. Future research should examine how these factors influence model performance, especially in the context of dynamic difficulty adjustment and environment transitions during training.
- **Limited Scope of Environments**: Our study is based on 2739 SCALER environments, and while this is a larger number compared to handcrafted ones, the scaling still hasn't subsided due to our experiments. Further research is needed to explore scaling laws related to environment size, model size, and computational resources, and to understand how these scaling factors impact both model training efficiency and performance.

References

Art of Problem Solving. 2023. Amc 12 (2023) problems. https://artofproblemsolving.com/wiki/index.php/2023_AMC_12A_Problems. We use the 2023 AMC 12A/12B problems as hosted by AoPS; a processed version is available at https://huggingface.co/datasets/zwhe99/amc23.

Bytedance-Seed-Foundation-Code-Team, :, Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, Bowen Li, Linyi Li, Boyi Liu, Jiaheng Liu, Kaibo Liu, Qi Liu, Shukai Liu, and 37 others. 2025. Fullstack bench: Evaluating llms as full stack coders. *Preprint*, arXiv:2412.00535.

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

Jiaqi Chen, Bang Zhang, Ruotian Ma, Peisong Wang, Xiaodan Liang, Zhaopeng Tu, Xiaolong Li, and Kwan-Yee K. Wong. 2025a. Spc: Evolving self-play critic via adversarial games for llm reasoning. *Preprint*, arXiv:2504.19162.

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

Xinjie Chen, Minpeng Liao, Guoxin Chen, Chengxi Li, Biao Fu, Kai Fan, and Xinggao Liu. 2025c. From data-centric to sample-centric: Enhancing llm reasoning via progressive optimization. *Preprint*, arXiv:2507.06573.

Zhuoyue Chen, Jihai Zhang, Ben Liu, Fangquan Lin, and Wotao Yin. 2025d. Scale down to speed up: Dynamic data selection for reinforcement learning. *Training*, 2500:3000.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Wenkai Fang, Shunyu Liu, Yang Zhou, Kongcheng Zhang, Tongya Zheng, Kaixuan Chen, Mingli Song, and Dacheng Tao. 2025. Serl: Self-play reinforcement learning for large language models with limited data. *Preprint*, arXiv:2505.20347.

Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. 2025. Klear-codetest: Scalable test case generation for code reinforcement learning. *Preprint*, arXiv:2508.05710.

Anna Goldie, Azalia Mirhoseini, Hao Zhou, Irene Cai, and Christopher D. Manning. 2025. Synthetic data generation & multi-step rl for reasoning & tool use. *Preprint*, arXiv:2504.04736.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. 2025a. Genenv: Difficulty-aligned co-evolution between llm agents and environment simulators. *Preprint*, arXiv:2512.19682.

Yiduo Guo, Zhen Guo, Chuanwei Huang, Zi-Ang Wang, Zekai Zhang, Haofei Yu, Huishuai Zhang, and Yikang Shen. 2025b. Synthetic data rl: Task definition is all you need. *Preprint*, arXiv:2505.17063.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, and 1 others. 2025. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *Preprint*, arXiv:2103.03874.

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K Jain, Virginia Aglietti, Disha Jindal, Peter Chen, Nishanth Dikkala, Gladys Tyen, Xin Liu, Uri Shalit, Silvia Chiappa, Kate Olszewska, Yi Tay, Vinh Q. Tran, Quoc V. Le, and Orhan Firat. 2025. BIG-bench extra hard. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26473–26501, Vienna, Austria. Association for Computational Linguistics.

Shipeng Li, Shikun Li, Zhiqin Yang, Xinghua Zhang, Gaode Chen, Xiaobo Xia, Hengyu Liu, and Zhe Peng. 2025. Learnalign: Reasoning data selection for reinforcement learning in large language models based on improved gradient alignment. *Preprint*, arXiv:2506.11480.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Axel Gimeno Gil, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Alphacode data materials. Includes dm-code_contests (CodeContests competitive programming dataset used in AlphaCode).

Xiao Liang, Zhong-Zhi Li, Yeyun Gong, Yang Wang, Hengyuan Zhang, Yelong Shen, Ying Nian Wu, and Weizhu Chen. 2025. Sws: Self-aware weakness-driven problem synthesis in reinforcement learning for llm reasoning. *Preprint*, arXiv:2506.08989.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*. Releases PRM800K and the MATH-500 (500-problem) test subset via `https://github.com/openai/prm800k`.

Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, and 1 others. 2025a. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning. *arXiv preprint arXiv:2506.24119*.

Huanyu Liu, Jia Li, Hao Zhu, Kechi Zhang, Yihong Dong, and Ge Li. 2025b. Saturn: Sat-based reinforcement learning to unleash language model reasoning. *arXiv preprint arXiv:2505.16368*.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, and 244 others. 2024. Openai o1 system card. *Preprint*, arXiv:2412.16720.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and Shuiwang Ji. 2025a. Curriculum reinforcement learning from easy to hard tasks improves llm reasoning. *Preprint*, arXiv:2506.06632.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and 1 others. 2025b. Curriculum reinforcement learning from easy to hard tasks improves llm reasoning. *arXiv preprint arXiv:2506.06632*.

Noam Razin, Zixuan Wang, Hubert Strauss, Stanley Wei, Jason D Lee, and Sanjeev Arora. 2025. What makes a reward model a good teacher? an optimization perspective. *arXiv preprint arXiv:2503.15477*.

Noam Razin, Hattie Zhou, Omid Saremi, Vimal Thilak, Arwen Bradley, Preetum Nakkiran, Joshua Susskind, and Etai Littwin. 2023. Vanishing gradients in reinforcement finetuning of language models. *arXiv preprint arXiv:2310.20703*.

Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. 2024. Rl on incorrect synthetic data scales the efficiency of llm math reasoning by eight-fold. *Preprint*, arXiv:2406.14532.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Taiwei Shi, Yiyang Wu, Linxin Song, Tianyi Zhou, and Jieyu Zhao. 2025. Efficient reinforcement finetuning via adaptive curriculum learning. *arXiv preprint arXiv:2504.05520*.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. 2025. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. *arXiv preprint arXiv:2505.24760*.

Kimi Team. 2025a. Kimi k1.5: Scaling reinforcement learning with llms.

Qwen Team. 2025b. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Hong Wang, Zhezheng Hao, Jian Luo, Chenxing Wei, Yao Shu, Lei Liu, Qiang Lin, Hande Dong, and Jiawei Chen. 2025a. Scheduling your llm reinforcement learning with reasoning trees. *Preprint*, arXiv:2510.24832.

Pinzheng Wang, Juntao Li, Zecheng Tang, Haijia Gui, and Min zhang. 2025b. Improving rationality in the reasoning process of language models through self-playing game. *Preprint*, arXiv:2506.22920.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*. Accepted at NeurIPS 2024 Datasets and Benchmarks (Spotlight). Dataset: https://huggingface.co/datasets/TIGER-Lab/MMLU-Pro.

Zijian Wu, Jinjie Ni, Xiangyan Liu, Zichen Liu, Hang Yan, and Michael Qizhe Shieh. 2025. Synthrl: Scaling visual reasoning with verifiable data synthesis. *Preprint*, arXiv:2506.02096.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025a. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, and 1 others. 2025b. Rlve: Scaling up reinforcement learning for language models with adaptive verifiable environments. *arXiv preprint arXiv:2511.07317*.

Bonan Zhang, Zhongqi Chen, Bowen Song, Qinya Li, Fan Wu, and Guihai Chen. 2025. Confclip: Confidence-weighted and clipped reward for reinforcement learning in llms. *Preprint*, arXiv:2509.17730.

Yifan Zhang and Team Math-AI. 2024. Aime 2024. https://huggingface.co/datasets/math-ai/aime24.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. 2025. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*.

A Prompt Design

A.1 Extract Meta Information Prompt

Prompt 1: Prompt used for extracting meta information.

```
You are given as input the full statement of a single algorithmic problem. Your job
    is to **emit one JSON code block** that captures:
1. All numeric *scale parameters* that are relevant to the **time complexity** of
    typical solutions.
2. The **type** of the required output.
3. Whether the required output is **unique**, whether the problem has exactly one
    correct output for each valid input.
## 1. Scale parameters
A *scale parameter* is any integer quantity that bounds:
- The number of items, elements, or positions (n= number of elements, m = number of
    edges, q = number of queries).
- The size (length) of a grid, string, or sequence (e.g. length up to 2e5, grid up
    to 1000 * 1000).
- The size of a state space or iteration space that an algorithm must explicitly
    handle.
### What to INCLUDE as scale parameters
Include a parameter only if **all** of the following are true:
- It directly bounds the size or count of something that is iterated over, e.g.:
  - number of elements / vertices / edges / queries (n, m, q, etc.),
  - length of a string or array,
  - rows / columns of a grid.
- The bounds appear explicitly in the statement, usually in the Input / Constraints
    section, like:
  - 1 <= n <= 2*10^5
  - 0 <= m <= 2*10^5
- You can clearly identify the parameter name (e.g., n, m, q, k, N, etc.).
### What to EXCLUDE from scale parameters
- Number of test cases / groups: **never** include t, T, or similar when it means "
    number of test cases".
- Pure value ranges for single items that do **not** change the input size, e.g.:
  - -10^9 <= a_i <= 10^9 when a_i is just the value of an element.
  - Coordinate or weight ranges that are not used as sizes of arrays/grids.
- Any quantity that only affects output format or precision.
### Representation of scale parameters
In the JSON, represent scale parameters under the key "scale_params" as:
<scale_params example>
## 2. Output type classification
You must classify the type of the required output into exactly one of the following
    strings:
- "string": The required output is a single string or a small number of strings.
- "number": The required output is a single numeric value (integer, real, etc.), e.g
    . "print one integer as the answer".
- "array": The required output is a one-dimensional sequence (list) of values, e.g.
    an array of integers, a permutation, a sequence of answers for each query when
    printed as space-separated numbers or in multiple lines.
- "graph": The required output is a graph structure, such as a set of edges, tree
    description, adjacency list, or any structure where the output itself is
    naturally a graph.
- "matrix": The required output is a 2D grid or matrix.
- "bool": The required output is logically a boolean answer, e.g. "YES/NO", "True/
    False", "Alice/Bob", etc.
- "others": The required output is a complex or mixed structure that does not fit
    clearly into any of the above categories.
## 3. Output uniqueness
You must also decide whether the required output is unique for each valid input.
Define "is_output_unique" as:
- true if, for any fixed valid input, there is exactly one correct output that
    satisfies the problem statement.
- false if the statement allows multiple different outputs to be accepted as correct
     for the same input.
## JSON Output Specification
You must produce exactly one JSON object in a fenced JSON code block.
The JSON must have the following top-level keys:
```

```
- "scale_params": an object mapping parameter names to {{ "min": <int>, "max": <int> }}.
- "output_type": one of "string", "number", "array", "graph", "matrix", "bool", "others".
- "is_output_unique": a boolean.
## Example 1: <example 1>
## Example 2: <example 2>
## Example 3: <example 3>
## Final instruction
Now, read the provided problem statement and output the single JSON code block accordingly.
{problem}
```

A.2 Generate Test Case Prompt

Prompt 2: Prompt used for constructing generate_testcase function.

```
You are given as input a single *algorithmic problem statement* (like those from programming contests). Your job is to **emit one Python code block** that defines a *test-case generator* function for this problem.
The generator must produce **exactly one** valid test case per call, parameterized only by the numeric scale values provided via a JSON object.
## Input
You will be given:
1. A raw problem statement in natural language that fully specifies:
   - the input format,
   - the constraints,
   - and the meaning of each variable.
2. An example json_obj instance:
   - This is only an example to clarify field names and typical ranges.
   - Your code must work for any valid json_obj that matches the described schema.
## Required Python output (emit exactly one Python code block)
You must output a Python code block that defines **one single function** with the following signature:
```python
def generate_testcase(json_obj: dict) -> tuple[str, dict]:
    """
    Generate a test case based on the given json_obj.
    Parameters:
    - json_obj (dict): The input JSON object containing problem parameters.
    Returns:
    - tuple[str, dict]: A tuple containing:
      - The first element is a string representing the test case in input format.
      - The second element is a dictionary representing the same test case.
    """
    ...
```
Return value
- Your function must return both the string and the dictionary representation of the test case in a tuple. The first element of the tuple should be the string format, and the second element should be the dictionary format.
    - output_str:
        - A single string that is a valid input for the problem according to the Input section, representing exactly one logical test case.
        - If the problem statement defines a format with multiple test cases controlled by an integer T in the input, you must set T = 1.
        - Example: "1\n5\n1 2 3 4 5"
    - output_dict:
        - A Python dict that is a structured, formal description of the same test case.
        - If the problem statement contains multiple test cases, **do not** introduce T or any extra wrapper.
        - Example: {"n": 5, "list": [1, 2, 3, 4, 5]}
Constraints
- All sizes (counts, lengths, number of operations, etc.) must be determined only from json_obj.
- All other values (elements of arrays, weights, edges, indices, etc.) must be generated randomly within a reasonable range and **strictly smaller than 10000**, while satisfying the problem's constraints at the same time.
```

```
- If the problem allows "no-solution" cases (e.g., the intended output is -1 when no
    solution exists), you should **bias your random generation towards test cases
    that admit at least one valid solution**, and explicitly construct values to
    satisfy any hidden feasibility conditions, so that the correct solution is not
    trivially always the "no-solution" output.
## Problem statement
{problem}
## Example json_obj
{example_json_obj}
```

B Environment Cases

B.1 Example Problem: 33_C. Wonderful Randomized Sum

Name	33_C. Wonderful Randomized Sum
Problem Description	Learn, learn and learn again — Valera has to do this every day. He is studying at mathematical school, where math is the main discipline. The mathematics teacher loves her discipline very much and tries to cultivate this love in children. That's why she always gives her students large and difficult homework. Despite that Valera is one of the best students, he failed to manage with the new homework. That's why he asks for your help. He has the following task. A sequence of n numbers is given. A prefix of a sequence is the part of the sequence (possibly empty), taken from the start of the sequence. A suffix of a sequence is the part of the sequence (possibly empty), taken from the end of the sequence. It is allowed to sequentially make two operations with the sequence. The first operation is to take some prefix of the sequence and multiply all numbers in this prefix by - 1. The second operation is to take some suffix and multiply all numbers in it by - 1. The chosen prefix and suffix may intersect. What is the maximum total sum of the sequence that can be obtained by applying the described operations?

Generate Testcase	`def generate_testcase(json_obj, output_format="str"):`
	`  # Generate random test case for the problem`
	`  n = int(json_obj.get("n", 10))`
	`  numbers = random.sample(range(1, 100), n)`
	
	`  if output_format == "dict":`
	`  return {"n": n, "numbers": numbers}`
	`  else:`
	`  return f"{n}\n{' '.join(map(str, numbers))}"`

Output Requirement	The first and the only line of the output should contain the answer to the problem.
Difficulty Mapping	{ "0": 0, "1": 1, "2": 2, "3": 2, "4": 3, "5": 4, "6": 5, "7": 6, "8": 8, "9": 11, "10": 14, "11": 18, "12": 23, "13": 30, "14": 39, "15": 51, "16": 67, "17": 87, "18": 112, "19": 146, "20": 190, "21": 247, "22": 318 }

B.2 Example Problem: 1497_D. Genius

Name	1497_D. Genius

Problem Description	Please note the non-standard memory limit.There are n problems numbered with integers from 1 to n. i-th problem has the complexity $c_i = 2^i$, tag tag_i and score s_i.After solving the problem i it's allowed to solve problem j if and only if $IQ < |c_i - c_j|$ and $tag_i \neq tag_j$. After solving it your IQ changes and becomes $IQ = |c_i - c_j|$ and you gain $|s_i - s_j|$ points.Any problem can be the first. You can solve problems in any order and as many times as you want.Initially your IQ = 0. Find the maximum number of points that can be earned.

Generate Testcase	`def generate_testcase(json_obj, output_format="str"):`
	`  # Generate a test case for the Genius problem`
	`  n = int(json_obj.get("n", 10))`
	`  tags = random.sample(range(1, n+1), n)`
	`  scores = random.sample(range(1, 100), n)`
	
	`  if output_format == "dict":`
	`  return {"n": n, "tags": tags, "scores": scores}`
	`  else:`
	`  return f"{n}\n{' '.join(map(str, tags))}\n{' '.join(map(str, scores))}"`

Output Requirement	For each test case print a single integer — the maximum number of points that can be earned.
Difficulty Mapping	{ "0": 0, "1": 2, "2": 3, "3": 4, "4": 7, "5": 10, "6": 17, "7": 27, "8": 43, "9": 69, "10": 110, "11": 176, "12": 281, "13": 450, "14": 721, "15": 1153 }

B.3 Example Problem: 1466_B. Last Minute Enhancements

Name	1466_B. Last Minute Enhancements
Problem Description	Athenaeus has just finished creating his latest musical composition and will present it tomorrow to the people of Athens. Unfortunately, the melody is rather dull and highly likely won't be met with a warm reception. His song consists of n notes, which we will treat as positive integers. The diversity of a song is the number of different notes it contains. As a patron of music, Euterpe watches over composers and guides them throughout the process of creating new melodies. She decided to help Athenaeus by changing his song to make it more diverse.Being a minor goddess, she cannot arbitrarily change the song. Instead, for each of the n notes in the song, she can either leave it as it is or increase it by 1.Given the song as a sequence of integers describing the notes, find out the maximal, achievable diversity.

Generate Testcase	`def generate_testcase(json_obj, output_format="str"):`
	`  # Generate test case for the song diversity problem`
	`  n = int(json_obj.get("n", 5))`
	`  notes = random.sample(range(1, 10), n)`
	
	`  if output_format == "dict":`
	`  return {"n": n, "notes": notes}`
	`  else:`
	`  return f"{n}\n{' '.join(map(str, notes))}"`

Output Requirement	For each test case, you should output a single line containing precisely one integer, the maximal diversity of the song, i.e. the maximal possible number of different elements in the final sequence.

Difficulty Mapping	{ "0": 0, "1": 1, "2": 2, "3": 2, "4": 3, "5": 4, "6": 5, "7": 6, "8": 8, "9": 11, "10": 14, "11": 18, "12": 23, "13": 30, "14": 39, "15": 51, "16": 67, "17": 87, "18": 112, "19": 146, "20": 190, "21": 247, "22": 321, "23": 418 }

C Training Detail

Model Name	Qwen3-4B-Base Qwen3-1.7B-Base
K_{slope}	10
K_{zero}	5
K_{sat}	5
Learning Rate	1e-6
Learning Rate Warmup Steps	20
Batch Size	64
Max Prompt Length	4096
Max Response Length	8192
Entropy Coefficient	0
Number of Environments per Step	64
Training Prompt Batch Size	64
Mini Batch Size	64
Reward Estimator	grpo
KL Loss Coefficient	0.0
Clip Ratio Low	0.2
Clip Ratio High	0.2
Temperature (Training)	1.0
Top P (Training)	1.0
Top K (Training)	-1
Validation Temperature	0.6
Validation Top P	0.95
Number of Response per Prompt	8