When a Fixed Task Generator Falls Behind: Towards Adaptive Curricula for Training Terminal Agents

Dang Nguyen[♡*] **Ruta Desai**[♣] **Ahmed Elgohary**[♣]
[♣]Microsoft Research [♡]University of Maryland, College Park
dangmn@umd.edu, {rutadesai,ahmedghoneim}@microsoft.com

Abstract

Training terminal agents with reinforcement learning (RL) on challenging tasks yields sparse learning signals when a weak policy rarely succeeds. Adaptive curriculum generation addresses this: a Generator proposes easier intermediate tasks, and a Solver attempts them. Co-training the Generator, however, adds optimization, stability, and systems complexity, so we ask whether a strong fixed Generator is enough. We use GPT-5 as a fixed Generator that produces simplified variants of challenging terminal tasks, and train Qwen3.5-9B as the Solver on both the original and the simplified tasks. Across three agent harnesses, this improves the Solver's pass@1 over RL on the original tasks alone by 1.0–4.7 points (2.9%–11.0% relative), but our analyses reveal two limitations. First, the fixed Generator fails to keep pace with the improving Solver: the share of informative generated tasks falls from 55% in the first 20 training steps to 24% in the last 20. Second, prompting alone does not reliably calibrate task difficulty: even with GPT-5, generated tasks can undershoot or overshoot the intended difficulty. These findings motivate adaptive Generator–Solver co-training to keep the curriculum informative.

1 Introduction

Reinforcement learning (RL) with verifiable rewards is behind some of the most striking recent advances in language models, from reasoning [17, 10] to software engineering [20] and, most recently, terminal agents [22, 13, 8]. We focus on terminal agents, whose tasks can require long chains of shell commands, tracking the state of the environment, and recovering from errors. On the hardest of these tasks, however, the policy being trained (the *Solver*) rarely succeeds, so most rollouts earn no reward and leave it with little to learn from. Curriculum learning alleviates this by introducing intermediate tasks that are easier than the targets but still exercise the capabilities needed to solve them [3, 9, 24, 28]. Adaptive curriculum generation builds such curricula automatically [31, 6, 32, 5]: instead of a hand-designed sequence of tasks, a Generator proposes tasks, the Solver attempts them, and the Solver's success rate indicates whether the tasks are appropriately difficult [6, 38, 12]. The same feedback can update the Generator as the Solver improves [31, 5, 12]. This adaptation matters because useful difficulty is relative to the current Solver: tasks that are informative early in training may become trivial later [16, 7]. Adaptive curricula therefore have the potential to track the Solver's capabilities as they evolve over training [38, 12, 39, 1].

But is adapting the Generator actually necessary to realize this benefit? Training the Generator and Solver jointly adds complexity: the Generator's reward must be designed and protected against reward hacking, the updates of the two policies must be balanced so that they progress at comparable rates, and the training infrastructure must run both efficiently. At the same time, frontier language models

[*]Work done during Dang Nguyen's internship at Microsoft Research. [♣]Equal mentorship.

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: Towards Test-Time Continual Learning Agents.

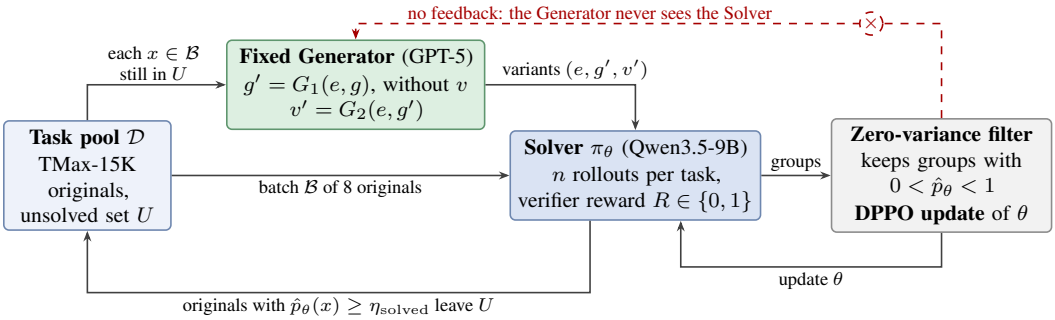


Figure 1: **Training with the Fixed-Generator Curriculum (FGC)** (Section 3.2). Only the Solver is updated. The dashed arrow marks the feedback to the Generator that adaptive curriculum generation would add.

are already strong task generators, which suggests a simpler alternative: keep the Generator fixed and train only the Solver on the tasks it generates. We therefore study whether a strong fixed Generator can keep the curriculum appropriately challenging as the Solver improves, and use its failure modes to understand what adaptive Generator training would need to accomplish.

To answer this question, we build the Fixed-Generator Curriculum (FGC; Figure 1), an RL pipeline in which GPT-5, as a fixed Generator, produces simplified variants of challenging terminal tasks, and Qwen3.5-9B is trained as the Solver on both the original and the generated tasks. The approach is useful: on OpenThoughts-TBLite [25], training with the generated tasks improves Solver pass@1 over RL on the original tasks alone by 1.0 to 4.7 points (2.9% to 11.0% relative) across three agent harnesses. However, the fixed Generator's task distribution does not move with the Solver as it improves: the share of generated tasks that are informative, neither solved nor failed in every rollout, falls from 55% in the first 20 training steps to 24% in the last 20. We also study prompting strategies meant to calibrate the generated tasks to the Solver's current capabilities, but they do not reliably control difficulty: even with a strong Generator such as GPT-5, generated tasks can undershoot or overshoot the intended level. These findings motivate co-training the Generator with the Solver, so that curriculum generation adapts as the Solver improves.

2 Related Work

Training terminal agents. Language-model agents for software engineering are commonly trained in the SWE-bench setting, building on agent scaffolds [35], executable training environments [27, 36, 14], and RL with test-based rewards [20]. Terminal-Bench [22] and OpenThoughts-TBLite [25] moved the target to general command-line tasks. TMax [13], whose corpus and RL recipe we build on, and other pipelines [8, 30, 18] generate corpora of terminal tasks for training. At the other end, Ornith-1.5 [26] lets the policy itself learn to propose tasks, scaffolds, and solutions. Our work sits between the two: we keep a strong Generator fixed, but run it during RL to rewrite the tasks the Solver cannot yet solve into easier ones, and measure what it can and cannot do for the Solver over the course of training.

Adaptive curricula for LLMs. Curricula that track the learner are a long-standing idea in RL, from teacher-student curriculum learning [21] to asymmetric self-play [31] and unsupervised environment design [5]. In RL fine-tuning of language models, DAPO [37] discards prompts whose sampled responses are all correct or all incorrect, the rule our zero-variance filter follows, and LILO [7] prioritizes questions with high variance of success. Recent work trains language models to propose problems as well as solve them, rewarding the proposer for problems that are neither too easy nor too hard for the solver [38, 12]. This proposer-solver training has been extended to software repositories [34], tool use [1], and executable environments [19]. Closest to our curriculum, GASP [15] has a teacher write an easier variant of a hard coding question and then a harder variant of that easier question, while Socratic-Zero [33] refines questions with a fixed strong teacher and distills it into a trainable generator. Trained proposers can also collapse, toward artificially complex or less diverse problems, unless a guide [2] or a diversity signal combined with a difficulty gate [23] keeps them

Shared environment e: one Docker image for both tasks

Ubuntu image with Python 3 and `jq`. `/app/logs/` contains `api.jsonl` (JSON lines), `worker.log` (syslog format, timestamped in local time, UTC−7), and `web.log` (Apache combined format).

Original task $x = (e, g, v)$, *intricate* tier

Instruction g. Produce `/app/report.json` giving, for each service (`api`, `worker`, `web`), the number of failed requests per UTC hour. Retried requests share a `request_id` and must be counted once, by their final outcome.

Verifier v. *Never shown to G_1 or G_2.*

FGC variant $x' = (e, g', v')$: same image, new g' and v'

Instruction $g' = G_1(e, g)$. Read `/app/logs/api.jsonl`. Each line is a JSON object with fields `ts` (ISO-8601, UTC) and `level`. Count lines where `level` is `"ERROR"`, grouped by hour, and write `/app/api_errors.json` as `{"2025-08-10T10": 3, ...}`, keyed by the hour prefix of `ts`. Omit hours with zero errors.

Verifier $v' = G_2(e, g')$.

```
def test_counts():
    rows = [json.loads(l)
            for l in open("/app/logs/api.jsonl")]
    expected = Counter(r["ts"][:13] for r in rows
                       if r["level"] == "ERROR")
    assert json.load(open("/app/api_errors.json")) \
        == dict(expected)
```

Figure 2: **An example terminal task (left) and its FGC simplification (right).**

useful. We complement these methods with a diagnosis of the fixed-Generator alternative: on hard terminal tasks, even a frontier model used as a fixed Generator saturates within tens of steps, and prompting alone does not reliably bring it back to the Solver's frontier.

3 Setup and Fixed-Generator Curriculum

3.1 Terminal Tasks

A terminal task is a triple $x = (e, g, v)$. The *environment definition* e specifies where the agent operates: in our setting, a Dockerfile, optionally with installation scripts, that builds a Docker image containing the files, packages, and services the task needs. The *instruction* g states in natural language what the agent must accomplish in this environment, and the *verifier* v is a script that inspects the container once the agent has finished and returns a binary verdict. Figure 2 shows an example task and its FGC simplification.

In each turn, the agent issues one Bash command and observes its standard output, standard error, and exit code. The episode ends when the agent issues a designated submit command: the verifier is then uploaded and run on the resulting container, and the agent receives a reward of 1 or 0. An episode that ends in any other way receives a reward of 0, and the verifier is not run.

3.2 Fixed-Generator Curriculum

Synthetic task simplification. Modifying all three components of a task $x = (e, g, v)$ is prohibitively expensive: the search space is large, and every new environment definition e' would have to be built into a Docker image and checked to run correctly. We therefore keep the environment fixed and simplify a task by rewriting its instruction g and updating its verifier v accordingly, which also lets us reuse Docker's build cache. Generating (g', v') in a single call from the full task x, however, risks leaking the original verifier into the new instruction: v may encode the exact paths, file contents, constants, and hashes that the agent is supposed to produce, and a Generator that sees v may copy them into g', making the variant solvable by reading them off. To avoid this, we generate the instruction and the verifier in two stages, denoted by G_1 and G_2 (see Appendix D for the prompts):

$$g' = G_1(e, g), \qquad v' = G_2(e, g'), \tag{1}$$

so G_1 never sees v, and G_2 sees neither g nor v, which reduces the risk that v' checks anything not specified by g'.

Incorporating synthetic tasks in RL. The Fixed-Generator Curriculum (FGC; Figure 1 and Algorithm 1) trains the Solver on the original tasks together with simplified variants written by a fixed

Algorithm 1 Fixed-Generator Curriculum

Require: task pool $\mathcal{D}$, Solver π_θ, Generator $G = (G_1, G_2)$, n, η_{solved}

$U \leftarrow \mathcal{D}$ ▷ every task starts unsolved
for each training step **do**
  sample a batch $\mathcal{B} \subset \mathcal{D}$ and set $\mathcal{B}' \leftarrow \emptyset$
  for each $x = (e, g, v) \in \mathcal{B}$ with $x \in U$ **do**
    $g' \leftarrow G_1(e, g)$ ▷ v is never shown
    $v' \leftarrow G_2(e, g')$
    $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{(e, g', v')\}$
  end for
  for each $x \in \mathcal{B} \cup \mathcal{B}'$ **do**
    roll out $y^{(1)}, \ldots, y^{(n)} \sim \pi_\theta(\cdot \mid x)$ and compute their rewards $R^{(i)}$
    if $x \in \mathcal{B}$ and $\hat{p}_\theta(x) \geq \eta_{\text{solved}}$ **then** $U \leftarrow U \setminus \{x\}$
    end if
  end for
  discard the groups whose rollouts all receive the same reward
  update θ with the DPPO loss on the remaining groups
end for

Generator. All original tasks start in the unsolved set U. At each training step, we sample a batch of original tasks, generate a simplified variant of each sampled task still in U, and roll out the Solver on the originals and the variants alike. An original task x leaves U once its empirical solve rate reaches $\hat{p}_\theta(x) \geq \eta_{\text{solved}}$; it is then no longer simplified, but it is still sampled and trained on, since a task above the solved threshold can still yield groups with mixed rewards that carry learning signal.

4 Do Simplified Tasks Improve RL Training?

Our first question is whether training on simplified tasks alongside the original tasks benefits the Solver. To answer it, we train the Solver twice, once on the original tasks only and once with FGC (Section 3.2), and compare the two checkpoints on a terminal-agent benchmark under three agent harnesses.

4.1 Experimental Setup

Training setup. Training tasks come from TMax-15K [13], a corpus of 14,601 synthetic tasks in the format of Section 3.1. The tasks cover nine domains, such as security, software engineering, and data science, and TMax divides them into four difficulty tiers: *short*, *moderate*, *complex*, and *intricate*. Both runs start from Qwen3.5-9B and follow the training recipe, harness, and infrastructure of TMax [13]. In particular, we train the Solver with DPPO [29], fully asynchronously with a maximum lag of 4 training steps. Each step samples 8 tasks with $n = 32$ rollouts each, and we use active sampling: when all n rollouts of a task receive the same reward, a zero-variance filter discards the group, and freshly drawn tasks replace discarded groups until a step's batch holds 8 groups. Both runs train for 80 steps, and FGC uses a solved threshold of $\eta_{\text{solved}} = 0.5$. Table 3 in Appendix A lists the hyperparameters.

Evaluation setup. We evaluate on OpenThoughts-TBLite (TBLite) [25], 100 Terminal-Bench tasks curated to track Terminal-Bench 2 performance at a fraction of the cost, with difficulty balanced against the pass rates of a reference model. Because the training harness is not a standard evaluation scaffold, we evaluate every checkpoint under three existing ones: an internal agent harness, and Terminus-2 [22] and mini-SWE-agent [35] as implemented in Harbor [11]. The three differ in, among other settings, how they manage context (Appendix A), so we report each harness separately and never pool across them. We sample at temperature 0.6 under the internal harness and Terminus-2 and with the model's default sampling settings under mini-SWE-agent, and report pass@k for $k \in \{1, 3, 5\}$ from 40 trials per task [4].

4.2 Quality of the Generated Tasks

Before training, we measured how reliably a Generator turns a TMax-15K task into a usable simplified task. We had GPT-5 and Qwen3.5-9B, the Solver's own base model, simplify the same 400 tasks, 100 from each difficulty tier, with the same prompts. A GPT-5 judge, which may favor its own outputs, scored each candidate on *relevancy* to the original (0 to 3), whether it is *easier* (0/1), and the *validity* of its verifier (0/1). We call a candidate *usable* if it parsed, had relevancy of at least 2, was judged easier, and had a valid verifier (Appendix B).

Table 1: Simplified-task quality. Rates are over the candidates each Generator produced.

	GPT-5	**Qwen3.5-9B**
Candidates produced (of 400)	400	342
Relevancy, mean (0–3)	2.68	2.32
Relevancy ≥ 2	0.998	0.939
Judged easier	1.000	0.997
Verifier judged valid	0.920	0.509
Usable (relevant, easier, valid)	0.918	0.477

GPT-5 produced a well-formed candidate, with a verifier that parses as valid Python, for all 400 tasks, whereas Qwen3.5-9B failed on 58 of them. Both models almost always made the task easier, but they differ in verifier validity (Table 1): GPT-5's verifiers were judged valid for 92.0% of its candidates and Qwen3.5-9B's for 50.9%. Over all 400 tasks, with Qwen3.5-9B's 58 failures counted as unusable, GPT-5 thus yields a usable simplified task for 91.8% of them and Qwen3.5-9B for 40.8%. A Solver-sized Generator writes a valid verifier only about half the time, which is also a practical hurdle for co-training the Generator. We therefore use GPT-5 as the Generator throughout our experiments.

4.3 RL + FGC Results

Table 2 reports the results. Training with DPPO raises the pass@1 of Qwen3.5-9B on TBLite under every harness, by 3.9 to 7.2 points, which confirms that the RL setup works. Adding FGC raises pass@1 further under every harness, by 1.0 to 4.7 points, and raises pass@3 and pass@5 under every harness as well. Training on simplified tasks alongside the original ones therefore benefits the Solver, which answers our first question.

5 Does a Fixed Curriculum Stay Challenging?

Section 4 showed that training on simplified tasks helps. Here we ask whether they keep helping as the Solver improves, by tracking the solve rate of every simplified task generated during the 80 steps of the FGC run.

The simplified tasks become too easy as the Solver improves. A task teaches the Solver only when some of its 32 rollouts succeed and some fail; if all succeed or all fail, the zero-variance filter discards its group (Section 4.1). We call such tasks *informative*, *fully solved*, and *unsolved*, respectively, and record the category of every simplified task at the step it was generated, before any filtering (Figure 3). The simplified tasks are easy from the start: in the first 20 steps, the Solver solves them 76% of the time on average, and one in three is solved in every rollout. They then get easier still. By the last 20 steps, two thirds are fully solved and only a quarter are informative, down from 55%,

Table 2: pass@k (%) on TBLite. Bold marks the better trained model under each harness.

Harness	Model	Pass@1	Pass@3	Pass@5
internal	Qwen3.5-9B (base)	27.2	43.0	51.0
	+ DPPO	34.4	51.0	57.0
	+ DPPO + FGC	**35.4**	**52.6**	**61.0**
Terminus-2	Qwen3.5-9B (base)	35.7	58.0	62.0
	+ DPPO	42.2	58.1	64.0
	+ DPPO + FGC	**45.8**	**63.3**	**69.0**
mini-SWE-agent	Qwen3.5-9B (base)	39.0	55.0	60.0
	+ DPPO	42.9	62.0	68.8
	+ DPPO + FGC	**47.6**	**63.9**	**69.8**

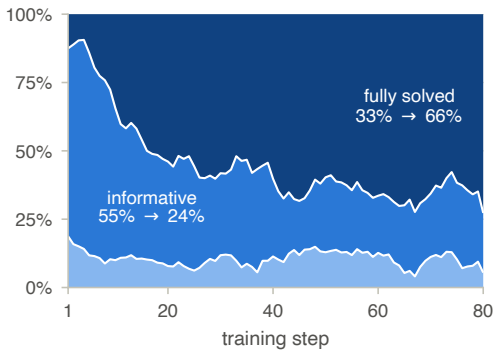


Figure 3: **The simplified tasks saturate.** Share of the simplified tasks at each step of the FGC run that were unsolved (■), informative (■), or fully solved (■). Counts are pooled over a 7-step centered window, and the labels give the means over steps 1–20 and 61–80.

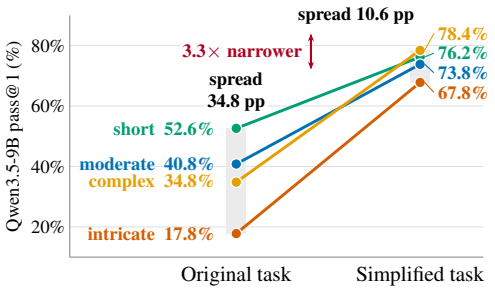


Figure 4: **Simplifying a task once with the FGC prompt maps every tier into the same band.** Pass@1 of the base Solver on each tier's 100-task subset, before and after one simplification by GPT-5. A 34.8-point spread across tiers becomes 10.6 points, and the tier order is not preserved.

while the share of unsolved tasks stays near one in ten. The tasks thus move straight from informative to fully solved.

Most of the Generator's output is wasted. The Generator writes more variants per step as training proceeds (12.7 at the start and 16.9 at the end), yet the number of informative variants per step falls from 7 to 4. Over the whole run, only 399 of the 1,197 variants (33%) contribute to the gradient; the other 798 (671 fully solved, 127 unsolved) contribute nothing. The original tasks avoid this decline only because of selection: active sampling keeps drawing tasks until eight groups survive the zero-variance filter, so the originals in every update are informative by construction, even though the Solver improves on them too (their mean solve rate rises from 62% to 71%). Variants go through the same filter, but a fully solved variant is simply discarded, and nothing replaces it. Each variant is written for a parent that the Solver still solves less than half the time, and its difficulty is wherever GPT-5's fixed notion of "easier" puts it; late in training, that is usually a task the Solver already solves in all 32 rollouts.

6 Can Prompting Alone Calibrate Task Difficulty?

Can this limitation be fixed simply by prompting the Generator more carefully? We compare three prompting strategies: the FGC prompt as is; a graded ladder, which asks for N variants ordered by difficulty; and a few-shot prompt that names a target difficulty tier, lists the reference Solver's pass@1 on each tier, and shows one example task per tier. Throughout, the reference Solver is the base Qwen3.5-9B, which does not change during these measurements, and the difficulty scale is given by the four tiers of TMax-15K (Section 4.1), on which its pass@1 is 52.6, 40.8, 34.8, and 17.8% from *short* to *intricate*.

Left to itself, the Generator maps every tier into the same narrow band. We apply the FGC prompt once to each tier's 100-task subset and measure the base Solver's pass@1 before and after (Figure 4). The four tiers' pass@1 of 52.6, 40.8, 34.8, and 17.8% become 76.2, 73.8, 78.4, and 67.8%: a spread of 34.8 points collapses to 10.6, and even the order is lost, as the simplified complex tasks come out easier than the simplified short ones. The prompt asks only for an easier task, not for how much easier, and the Generator appears to settle on roughly the same level regardless of how hard the original was. We believe this is the mechanism behind the saturation in Section 5.

A difficulty ladder controls the order of difficulty but never reaches the original task. We ask GPT-5 to rewrite each of 50 tasks into an N-rung ladder, $N \in \{3, 5, 7\}$, from rung 1, the easiest, to rung N, which should remove a single obstacle from the original and stay only slightly easier

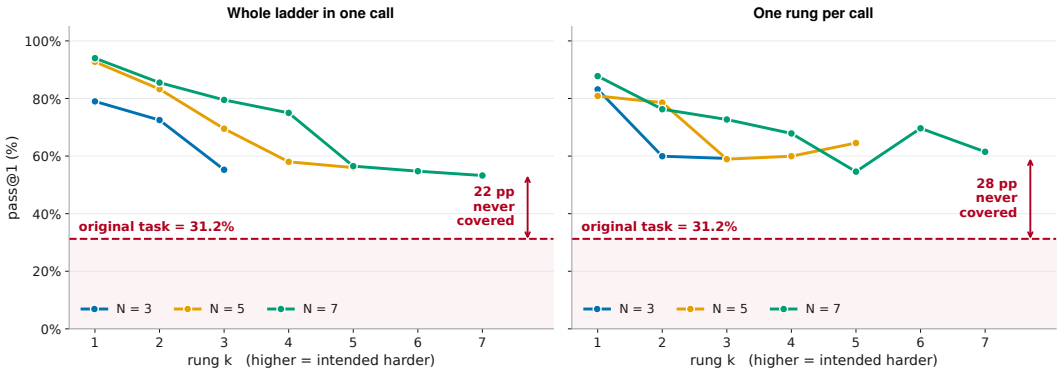


Figure 5: **Ladders are ordered but never reach the original.** Pass@1 of the base Solver on each rung of GPT-5's N-rung ladders for $N \in \{3, 5, 7\}$, written with the whole ladder in one call (left) or one rung per call (right). The dashed line is pass@1 on the original tasks, and the hardest rung of every ladder stays at least 22 (left) or 28 (right) points above it.

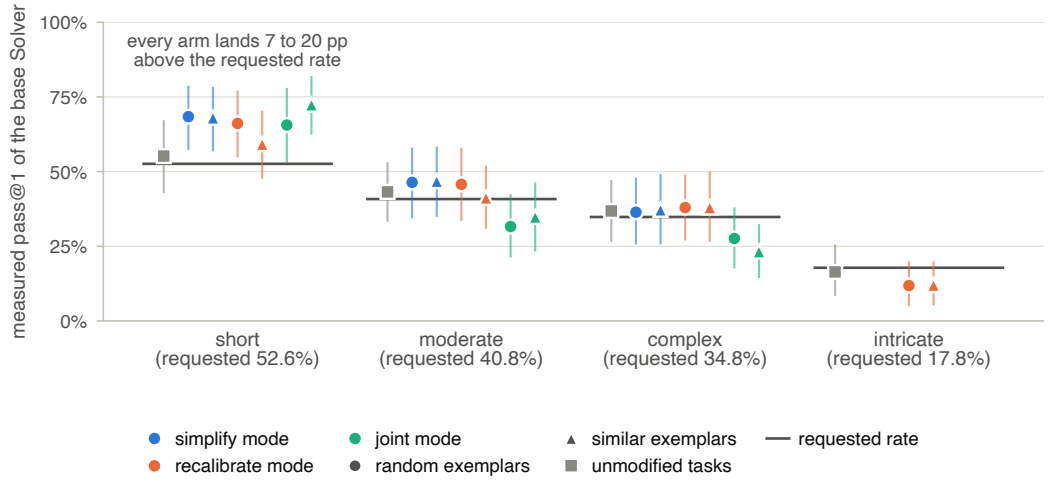


Figure 6: **Few-shot tier requests are ordered but not calibrated.** Mean pass@1 of the base Solver on the 50 tasks of each cell of the few-shot study, with a bootstrap 95% interval, against the requested tier's rate. Modes differ by hue, exemplar strategies by marker, and gray squares are unmodified tasks of the requested tier. A calibrated cell sits on its requested-rate line.

than it. We ask either for the whole ladder in one call, so that the model sees its own rungs while writing them, or for one rung per call, and score every rung with the reference Solver using 8 rollouts per task (Figure 5). The Generator orders the rungs but does not follow the instruction for the top one: the original tasks have pass@1 of 31.2%, yet the hardest rung of every ladder stays at least 22 points above that when the whole ladder is written in one call and at least 28 points above it when it is written one rung per call. Adding rungs does not close the gap, because the extra rungs land at the easy end.

Naming the target pass rate, with examples, restores the order but not the level. In the few-shot study, we ask GPT-5 to rewrite each of 50 intricate tasks to a named tier, showing it one example task from every tier. We vary the *mode* of the request, *simplify* (only easier tiers), *recalibrate* (any tier, including the parent's own), or *joint* (all targets in one call), and choose the examples either at random or by their similarity to the parent. This gives 20 cells of rewritten tasks plus 4 reference cells of unmodified tasks, 1,196 tasks in total, each rolled out 5 times; Appendix C details the design and lists every cell. Figure 6 shows that the order is restored: in every setting, the measured pass@1 rises

with the requested one, and the reference cells reproduce their tiers within 2.6 points, which confirms that the scale is sound. The level, however, is not calibrated. Every setting overshoots the short target by 7 to 20 points, so a request for an easy task yields an even easier one. Asking for the parent's own tier does not preserve its difficulty either: intricate tasks rewritten to intricate come out at 12% against 17.8%. Choosing similar rather than random examples has no detectable effect. Across settings, the generated tasks spread 1.4 to 2.8 times wider in difficulty than the tiers they were asked to match.

Prompting thus gives a fixed Generator control over the order of difficulty but not over its level. The Generator makes tasks easier, orders its own variants, and moves in the requested direction, but the easy end is always too easy and the top of a ladder stops well short of the original. These misses, as large as 28 points, are measured against a Solver that does not change; in training, the Solver improves, and Section 5 showed the simplified tasks becoming too easy within tens of steps, so the misses only grow.

7 Conclusion

Our results suggest that training the Solver on simplified variants of hard tasks alongside the originals can improve it in terminal-agent RL. In our experiments, however, a fixed Generator did not adapt as the Solver improved, so its contribution shrank during training, and the prompting strategies we tried did not correct this. Other ways of prompting a fixed Generator remain to be explored. Its prompt could include a summary of the Solver's current capabilities, for example a few carefully selected trajectories, or the Generator could be called repeatedly until a variant reaches the target difficulty, although this is expensive because every candidate must be rolled out to estimate its pass@k. More broadly, the Generator is the one part of the loop that is deployed once and never learns from what happens next, yet an agent that keeps improving through interaction needs a source of experience that adapts with it. A natural next step is therefore to train the Generator on Solver feedback, or at least condition it on that feedback, so that solve rates become its reward or input and its variants move with the Solver's frontier.

References

[1] E. C. Acikgoz, C. Qian, J. Hübotter, H. Ji, D. Hakkani-Tür, and G. Tur. Tool-R0: Self-evolving LLM agents for tool-learning from zero data. *arXiv preprint arXiv:2602.21320*, 2026.

[2] L. Bailey, K. Wen, K. Dong, T. Hashimoto, and T. Ma. Scaling self-play with self-guidance. In *Conference on Language Modeling (COLM)*, 2026.

[3] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning (ICML)*, 2009.

[4] M. Chen, J. Tworek, H. Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, and S. Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[6] C. Florensa, D. Held, X. Geng, and P. Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning (ICML)*, 2018.

[7] T. Foster, A. Sims, J. Forkel, and J. N. Foerster. LILO: Learning to reason at the frontier of learnability. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[8] K. Gandhi, S. Garg, N. D. Goodman, and D. Papailiopoulos. Endless terminals: Scaling RL environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

[9] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu. Automated curriculum learning for neural networks. In *International Conference on Machine Learning (ICML)*, 2017.

[10] D. Guo, D. Yang, H. Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

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

[12] C. Huang, W. Yu, X. Wang, H. Zhang, Z. Li, R. Li, J. Huang, H. Mi, and D. Yu. R-Zero: Self-evolving reasoning LLM from zero data. In *International Conference on Learning Representations (ICLR)*, 2026.

[13] H. Ivison, J. O. Yin, R. Shao, T. Xiao, N. Lambert, and H. Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

[14] N. Jain, J. Singh, M. Shetty, L. Zheng, K. Sen, and I. Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Conference on Language Modeling (COLM)*, 2025.

[15] S. Jana, C. Sancaktar, T. Daniš, G. Martius, A. Orvieto, and P. Kolev. GASP: Guided asymmetric self-play for coding LLMs. In *ICLR 2026 Workshop on AI with Recursive Self-Improvement*, 2026.

[16] M. Jiang, E. Grefenstette, and T. Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning (ICML)*, 2021.

[17] N. Lambert, J. Morrison, V. Pyatkin, et al. Tülu 3: Pushing frontiers in open language model post-training. In *Conference on Language Modeling (COLM)*, 2025.

[18] Z. Li, Y. Shi, Z. Li, et al. Recursive synthesis for long-horizon terminal tasks. *arXiv preprint arXiv:2608.05466*, 2026.

[19] B. Liu, S. Yu, Y. Jiang, et al. SPADE: Self-play in adaptive synthetic executable environments. *arXiv preprint arXiv:2608.19197*, 2026.

[20] M. Luo, N. Jain, J. Singh, et al. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. https://www.together.ai/blog/deepswe, 2025.

[21] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher-student curriculum learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3732–3740, 2020.

[22] M. A. Merrill, A. G. Shaw, N. Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, 2026.

[23] V. Mishra. Preventing curriculum collapse in self-evolving reasoning systems. *arXiv preprint arXiv:2603.13309*, 2026.

[24] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50, 2020.

[25] OpenThoughts-Agent Team, Snorkel AI, and Bespoke Labs. OpenThoughts-TBLite: A high-signal benchmark for iterating on terminal agents. https://www.openthoughts.ai/blog/openthoughts-tblite, 2026.

[26] Ornith Team. Ornith-1.5: From self-scaffolding to self-improvement. https://ornith.ai/ornith_1_5.html, 2026.

[27] J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning (ICML)*, 2025.

[28] R. Portelas, C. Colas, L. Weng, K. Hofmann, and P.-Y. Oudeyer. Automatic curriculum learning for deep RL: A short survey. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[29] P. Qi, X. Zhou, Z. Liu, T. Pang, C. Du, M. Lin, and W. S. Lee. Rethinking the trust region in LLM reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2026.

[30] Q. Shen, Z. Huang, V. Kamanuru, et al. SETA: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

[31] S. Sukhbaatar, Z. Lin, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations (ICLR)*, 2018.

[32] R. Wang, J. Lehman, J. Clune, and K. O. Stanley. POET: Open-ended coevolution of environments and their optimized solutions. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2019.

[33] S. Wang, Z. Jiao, Z. Zhang, Y. Peng, X. Ze, B. Yang, W. Wang, H. Wei, and L. Zhang. Socratic-Zero: Bootstrapping reasoning via data-free agent co-evolution. *arXiv preprint arXiv:2509.24726*, 2025.

[34] Y. Wei, Z. Sun, E. McMilin, J. Gehring, D. Zhang, G. Synnaeve, D. Fried, L. Zhang, and S. Wang. Toward training superintelligent software agents through self-play SWE-RL. In *International Conference on Machine Learning (ICML)*, 2026.

[35] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[36] J. Yang, K. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

[37] Q. Yu, Z. Zhang, R. Zhu, et al. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[38] A. Zhao, Y. Wu, T. Wu, Q. Xu, Y. Yue, M. Lin, S. Wang, Q. Wu, Z. Zheng, and G. Huang. Absolute Zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[39] Y. Zhou, S. Levine, J. Weston, X. Li, and S. Sukhbaatar. Self-challenging language model agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

A Training and Evaluation Details

Table 3: Training hyperparameters. Both training runs share every value except those under *FGC only*.

Policy and data		**Objective**	
Base model	Qwen3.5-9B	Algorithm	DPPO, Binary-TV
Seed	42	Divergence threshold (δ)	0.1
Training tasks	TMax-15K (14,601 tasks)	Ratio clipping	none
Max prompt tokens	2,048	KL coefficient (β)	0
Rollouts		**Optimization**	
Tasks per step	8	Optimizer	AdamW
Rollouts per task (n)	32	Learning rate / schedule	10^{-6} / constant, no warmup
Turn budget (H)	64	Weight decay	0
Per-turn generated tokens	16,384	Gradient-norm clip	1.0
Generated tokens per episode	65,536	Epochs per batch	1
Context length	67,584	Training steps	80
Temperature	1.0	**FGC only**	
Async steps / max staleness	4 / 4	Generator	GPT-5
Active sampling	on	Reasoning effort	medium
Solve-rate ceiling filter	none	Solved threshold (η_{solved})	0.5
Episode wall-clock timeout	none		

Evaluation harnesses. The internal harness gives the model a single tool, `bash`. Like mini-SWE-agent, and unlike Terminus-2, it does not support context compaction. We cap it at 16 turns and 32K tokens of context, as in Terminus-2; mini-SWE-agent shares the context limit. In all three harnesses, the agent has a time limit of 40 minutes per task.

B Quality of the Simplified Tasks

This appendix details the study summarized in Section 4.2. Before training, we measured how reliably a Generator turns a TMax-15K task into a usable simplified task, both to choose the Generator and to decide whether generated verifiers needed an execution check before use. We compared GPT-5, the FGC Generator, with Qwen3.5-9B, the Solver's own base model, on the same tasks and prompts. Table 1 in Section 4.2 summarizes the results.

Tasks and generation. We sampled 400 tasks from TMax-15K, 100 from each tier, with a fixed seed, and each Generator simplified every task once. Both Generators ran offline with the two-stage protocol of Section 3.2, in a form whose second stage asks for a pytest verifier, and GPT-5 used medium reasoning effort.

Judge. A GPT-5 judge at medium reasoning effort saw the environment description and the original and candidate instructions and verifiers. It scored each candidate on three axes: *relevancy* (0 to 3: does solving the easier task give insight into the original?), *easier* (0/1: does it take fewer steps or have a smaller scope?), and *validity* (0/1: does its verifier check completion of the easier task, rather than always passing or testing something else?). A candidate is *usable* if it parsed, scored at least 2 on relevancy, was judged easier, and had a valid verifier.

Yield. GPT-5 produced a candidate for all 400 tasks, and all 400 of its verifiers parsed as valid Python; no reply lacked a block, and no verifier copied the original. Qwen3.5-9B lost 58 tasks (14.5%): 24 replies lacked a required block, 33 verifiers did not parse, and one copied the original verifier. This result is why FGC does not execute generated verifiers before using them (Section 3.2).

Relevancy and easiness. GPT-5's simplified tasks were judged relevant to their parent (score 2 or 3) in 99.8% of cases, with a mean score of 2.68, against 2.32 for the tasks Qwen3.5-9B produced. On the 342 tasks both models simplified, GPT-5 scored higher on 150, tied on 151, and scored lower on 41 (mean difference $+0.36$; Wilcoxon signed-rank $p < 10^{-15}$). The *easier* axis is saturated: every GPT-5 candidate and all but one Qwen3.5-9B candidate were judged easier. Making a task smaller is not where simplification fails.

Verifier validity. This is where the two Generators separate. GPT-5's verifiers were judged valid in 92.0% of cases (368 of 400), with little variation across source tiers (95%, 92%, 89%, and 92% from *short* to *intricate*). Qwen3.5-9B's were valid in 50.9% of the candidates it produced and 43.5% of all 400 attempts, falling to 40% on *intricate* parents. Counting every attempt, GPT-5 yields a usable simplified task 91.8% of the time and Qwen3.5-9B 40.8%, a ratio of 0.44. Almost all of the loss is in the verifier rather than in the task design, which is why FGC uses an external Generator rather than the Solver's own model.

Length. The two Generators draft in opposite styles. Relative to the original instruction (median 2,111 characters), Qwen3.5-9B compresses to a median length ratio of 0.71 and lengthens only 5.8% of tasks, whereas GPT-5 expands to a median ratio of 1.18 and lengthens 69.3%, because it spells out an explicit contract: exact paths, expected file contents, and a list of what will be checked. Its verifiers are correspondingly longer (median 3,017 versus 2,109 characters).

C Difficulty-Calibration Study

This appendix details the few-shot study of Section 6.

Design. The parents are 50 tasks from the *intricate* tier of TMax-15K, and every setting rewrites the same parents, so comparisons between settings are paired. A setting is a (mode, exemplar strategy) pair, and a cell holds a setting's rewrites of the 50 parents to one target tier. The *simplify* and *joint* modes request the three easier tiers, one per call or all in a single call, and *recalibrate* requests all four, including *intricate* itself as a control. This gives 3, 3, and 4 cells per exemplar strategy. Four *reference* cells hold 50 unmodified tasks from each tier. The 24 cells hold 1,196 tasks, 996 rewrites and 200 reference tasks, each rolled out 5 times by the reference Solver (the base Qwen3.5-9B) under the internal harness (Appendix A), for 5,980 rollouts. The Generator is GPT-5 at medium reasoning effort, and it writes each rewrite in two stages: the instruction first, without the parent's verifier, and then a verifier for that instruction.

Exemplars. The Generator sees one exemplar task per tier, labeled with its tier. The *random* strategy draws them uniformly. The *similarity* strategy considers only tasks that share the parent's domain, skill type, and language, and picks the four exemplars that maximize a blend of their similarity to the parent and to one another, relaxing this restriction when a tier has no such task.

Metrics. Let t_j be the reference Solver's pass@1 on tier j, which is the target rate for that tier, and m_j the mean pass@1 of a cell's tasks. The **bias** of a cell is $m_j - t_j$. The **range recovery** of a setting is $(\max_j m_j - \min_j m_j)/(\max_j t_j - \min_j t_j)$: 0 means that its cells all land at the same level, 1 that it reproduces the requested spread, and above 1 that it spreads its tasks wider than asked (1.4 to 2.8 across settings; Section 6). A **paired comparison** of two settings uses the per-parent difference in pass@1. All intervals are 95% percentile bootstrap intervals with 10,000 resamples.

Results by cell. Table 4 lists every cell. Only three paired comparisons between modes have intervals that exclude zero, all of them joint against simplify: with similarity exemplars at the complex target (-14.0 points, $[-24.8, -4.0]$), and at the moderate target with random exemplars (-14.8, $[-28.4, -1.2]$) and with similarity exemplars (-12.0, $[-22.4, -1.6]$). All ten paired comparisons of similarity against random exemplars include zero; the largest difference is -7.4 points, for recalibrate to short ($[-18.8, +4.1]$).

D Prompts

Figure 7 shows the prompt the Generator uses to simplify a task (Section 3.2).

Table 4: Pass@1 of the reference Solver on each cell against its tier's target rate. Reference cells hold unmodified tasks of their tier; all other cells hold rewrites of intricate parents.

Mode	Exemplars	Tier	Measured (%)	Target (%)	Bias (pp)
reference	–	short	55.2	52.6	+2.6
reference	–	moderate	43.2	40.8	+2.4
reference	–	complex	36.8	34.8	+2.0
reference	–	intricate	16.4	17.8	−1.4
joint	random	short	65.6	52.6	+13.0
joint	similarity	short	72.4	52.6	+19.8
recalibrate	random	short	66.1	52.6	+13.5
recalibrate	similarity	short	59.2	52.6	+6.6
simplify	random	short	68.4	52.6	+15.8
simplify	similarity	short	68.0	52.6	+15.4
joint	random	moderate	31.6	40.8	−9.2
joint	similarity	moderate	34.8	40.8	−6.0
recalibrate	random	moderate	45.7	40.8	+4.9
recalibrate	similarity	moderate	41.2	40.8	+0.4
simplify	random	moderate	46.4	40.8	+5.6
simplify	similarity	moderate	46.8	40.8	+6.0
joint	random	complex	27.6	34.8	−7.2
joint	similarity	complex	23.2	34.8	−11.6
recalibrate	random	complex	38.0	34.8	+3.2
recalibrate	similarity	complex	38.0	34.8	+3.2
simplify	random	complex	36.4	34.8	+1.6
simplify	similarity	complex	37.2	34.8	+2.4
recalibrate	random	intricate	11.8	17.8	−6.0
recalibrate	similarity	intricate	12.0	17.8	−5.8

System Message

```
You are an expert curriculum designer for autonomous terminal/coding agents.

You are given an ORIGINAL task that an agent must solve inside a specific, sandboxed
Linux environment. Your job is to design an EASIER "stepping-stone" version of it.

The easier task MUST:
  - Be genuinely easier than the original: fewer steps, less background knowledge, or a
    smaller surface area.
  - Act as a stepping stone: an agent that solves the easier task should gain concrete,
    transferable insight that helps it solve the ORIGINAL task.
  - Stay concrete, actionable and objectively testable: success/failure must be decidable
    by a shell script whose exit code is 0 on success and non-zero on failure.
  - Assume the SAME environment as the original. The environment is defined by a Docker
    image build (Dockerfile + base_install.sh + post_install.sh). The easier task runs in
    the SAME image, so every package, file, service, git repository and artifact created
    during the build is already present and may be leveraged as a stepping stone. Do NOT
    rely on tools, files, packages or network the environment does not provide.

You will produce one artifact for the easier task:
  1. instruction.md -- the task statement shown to the agent (self-contained, unambiguous).
     Name the exact paths, file formats and observable outputs the agent must produce, so
     that a grading script written afterwards can decide success from the final state alone.

You are deliberately NOT shown the original test.sh, so nothing can be copied out of it
into instruction.md.
```

User Message

```
Here is the ORIGINAL task.

<environment>
{environment}
</environment>

<original_instruction_md>
{instruction_md}
</original_instruction_md>

Design an easier stepping-stone task for the SAME environment.

Think briefly, then output EXACTLY the following XML and nothing after it. Put the raw file
contents between the tags (CDATA is fine); do not add markdown fences inside the tags.

<result>
  <reasoning>One short paragraph: what makes the original hard, and why your easier task is a useful
stepping stone.</reasoning>
  <easier_instruction_md><![CDATA[
...full contents of the new instruction.md...
]]></easier_instruction_md>
</result>
```

Figure 7: Task-simplification prompt. Fields in curly braces are filled in for each task.