Sample Wide, Pick Smart

For a Fixed 0.5B Countdown Reasoner, Test-Time Selection Beats More Training

Stanford CS224R Default Project

Manat Kaur **Felipe Teixeira**
Department of Computer Science, Stanford University
{manat, flt}@stanford.edu

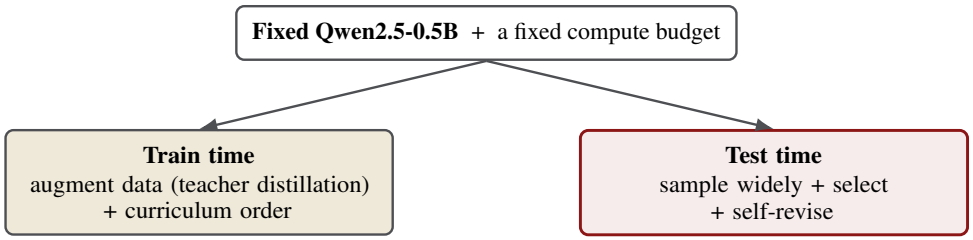
Key information. Mentor: Riya Karamunchi. External collaborators: none. Sharing project: this project is not shared with any other course.

Extended Abstract

Problem. Small language models are cheap to serve but reason poorly on multi-step problems. We study Countdown, a task where the model must combine a short list of numbers with $+ \; - \; \times \div$ to hit a target. Our base model is Qwen2.5-0.5B. The central question of our extension is simple to state: given a fixed model and a fixed budget of extra compute, where does that budget pay off, in training the policy or at test time?

Approach. We compare two arms on the same base policy. The train-time arm generates verifier-filtered teacher traces from a Qwen2.5-7B model and orders the RLOO curriculum from easy to hard. The test-time arm spends the budget on inference: we fine-tune a generative verifier that reads a candidate answer and emits a single calibrated $P(\text{correct})$ token plus a short critique, then use it two ways. First, selection: sample many candidates and let the verifier pick the best (best-of-N). Second, revision: feed the critique back to a fine-tuned reviser and let the model try again. We also sweep hybrid budgets that split a fixed budget between width (more samples) and depth (more revisions).

Findings. Test-time selection is the clear winner. A learned verifier turns a 0.316 pass@1 policy into a 0.750 solver at a 16-sample budget, capturing about 93% of the single-to-oracle gap, while being well calibrated (ROC-AUC 0.995, ECE 0.010). Fine-tuning the verifier was necessary: in-context prompting plateaus around 0.77 accuracy and cannot reproduce the structured critique (error-type accuracy 0.52 versus 0.98). Revision is the opposite story. At a matched budget, parallel best-of-N beats sequential self-revision at every budget of two or more (for example 0.619 versus 0.490 at budget four), and the critique adds nothing over a bare "try again." The hybrid sweep confirms that width dominates depth: the best split keeps at most one revision and never beats pure sampling by more than the noise floor. By contrast, train-time distillation and curriculum help only when combined, and only by about four points. The unifying explanation is that for this 0.5B model the bottleneck is generating a correct candidate, not recognizing one. So the efficient recipe is to sample broadly and let a learned validator select.



Abstract

We ask where to spend a fixed compute budget for a small Countdown reasoner (Qwen2.5-0.5B): on training the policy or at test time. On a shared base policy we compare train-time distillation and curriculum against test-time strategies built on a fine-tuned generative verifier. We find that test-time selection is the most effective use of compute. A learned verifier lifts pass@1 from 0.316 to 0.750 at a 16-sample budget (about 93% of the oracle ceiling), while self-revision underperforms plain sampling at every matched budget and a hybrid split never beats pure sampling beyond the noise. Train-time augmentation and curriculum help only in combination, and only modestly. The bottleneck for this model is generating a correct candidate, not recognizing one, so the efficient recipe is to sample widely and let a calibrated validator pick.

1 Introduction

As we see the rise of coding agents and long-running agents, it is clear that our most powerful models (Opus, GPT) are developing strong reasoning capabilities. However, these models are often prohibitively expensive, requiring a large amount of compute and inference cost. Smaller models, trained with far fewer parameters, struggle with complex and multi-step reasoning, a gap that becomes evident on tasks like Countdown, where the model is given a clear target and a constrained list of numbers and operations to reach the target.

In this project we study how different train-time and test-time strategies improve a Qwen2.5-0.5B model on Countdown-style mathematical reasoning. Following the default requirements, we begin with supervised fine-tuning on solution traces, then compare reinforcement learning variants that optimize directly against a rule-based verifier.

For our extension, we ask the question of what is the most effective place to spend compute, train-time or test-time. For train-time we generate verifier-filtered teacher traces and test whether distillation and curriculum learning improve the base policy. For test-time we train a generative verifier that judges candidate solutions and experiment with both best-of-N selection and iterative self-revision. Our main hypothesis is that for a fixed 0.5B model, test-time sampling with learned validation is a more effective use of additional compute than further training.

Our results validate this hypothesis. Train-time distillation and curriculum provide modest gains, and only when combined. Test-time sampling with a learned validator produces much stronger improvements: it lets the model generate many candidate solutions and select the most promising one. We also observe that self-revision performs worse than broad sampling, which shows that the bottleneck is not recognizing a correct solution but generating one in the first place.

2 Related Work

Distillation has become one of the main strategies for transferring reasoning behavior from larger models to smaller and cheaper ones. The DeepSeek team increased performance on AIME to 72.6% when distilling from Qwen-32B and to 50.4% when distilling from Llama-8B [1]. These results motivated us to try distillation from Qwen-7B.

However, distillation does not depend only on the parameter size of the teacher. It also depends heavily on data quality. LIMO shows that a relatively small number of high-quality reasoning

examples can be enough to elicit strong mathematical reasoning, reaching 63.3% on AIME from an 800-example dataset [2]. One limitation of standard distillation is the distribution mismatch between teacher and student: the student is trained on teacher distributions but at inference must generate from its own, weaker policy. Generalized Knowledge Distillation addresses this by having the student generate sequences on-policy while the teacher provides feedback on those student-generated outputs, so the student is only trained on examples from its own distribution [3].

We augment our distillation work with curriculum learning, where models are trained on examples of progressively increasing difficulty [4]. This helps because starting with problems that are too difficult means many rollout groups contain no correct solution, producing sparse rewards. By mirroring human learning, we keep early rewards non-zero so the gradient carries useful signal.

On the test-time side, Snell et al. [5] study whether language-model reasoning can be improved by spending more computation at inference rather than scaling model size or pretraining. This includes searching over sampled completions with a verifier or updating the model's response distribution at inference. They find that the benefit of test-time compute depends heavily on prompt difficulty and the base model's capability: for easy prompts extra sampling is unnecessary, and for the hardest prompts it may not help. Verifier-guided selection itself dates to Cobbe et al. [6], who train a separate verifier to rank candidate solutions, and process-level variants follow [7]. On the revision side, Huang et al. [8] report that current language models struggle to self-correct reasoning without external feedback, which directly motivates our selection-versus-revision comparison. Together these works shape our study of where extra compute is best spent.

3 Approach

3.1 Train time: more and better-ordered data

We generate a synthetic Countdown distillation dataset at three difficulty levels: easy, medium and hard. Difficulty is defined with a heuristic over the count of input numbers, the allowed arithmetic operations, the magnitude of the sampled numbers and target, and whether division is required. The assumption is that easy problems use fewer input numbers, while harder problems require larger magnitudes and more complex operations. We use a larger Qwen2.5-7B teacher to generate solution traces and pass them through the provided Countdown verifier, which checks that the answer uses the provided numbers, evaluates to the target, and is properly formatted. We chose the 7B teacher rather than 32B to reduce distributional mismatch with the 0.5B student: 7B offers stronger reasoning while keeping the output distribution closer to the student. We then use this distilled dataset for supervised fine-tuning.

We also use this data to augment RLOO training with curriculum learning. We order the problems from easy to hard using the same heuristic. For RLOO, the model samples multiple responses per prompt and receives a reward from the verifier: correct equations receive full reward, while incorrectly formatted equations receive partial or zero reward.

3.2 Test time: a learned generative verifier

The core of our extension is a generative verifier. Given a Countdown problem and a candidate answer, it emits a verdict as a single token (`correct` or `wrong`) followed by a short critique that names the error type. Reading the logit of the verdict token gives a calibrated probability $P(\text{correct})$ that we use as a ranking and stopping signal. We build the verifier's training data offline by wrapping the provided `countdown.py` scorer to label each candidate's correctness and error type, with real negatives sampled from the policy itself so the verifier sees the kinds of mistakes it will face at deployment. The verifier is trained with plain next-token supervised fine-tuning, reusing the unchanged SFT loop. Crucially, no oracle is ever called at inference: every online signal comes from the learned verifier, which predicts (and can be wrong about) whether an expression evaluates to the target. Figure 1 shows the pipeline: the offline scorer labels each candidate during data construction, and at inference the trained verifier emits the verdict, probability, and critique on its own.

Selection (best-of-N). The policy samples N candidates for a problem under the official decoder (temperature 0.6, top-p 0.95, top-k 20). The verifier scores each candidate's $P(\text{correct})$ and we return the highest-scoring one. We compare this against the single-sample baseline (pass@1) and an oracle

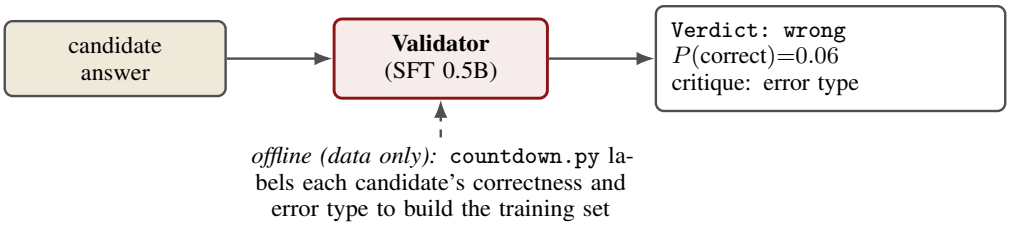
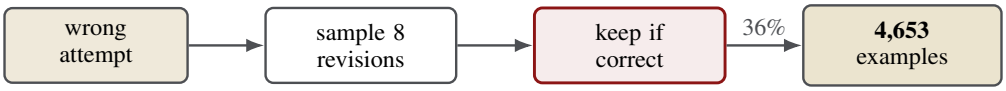
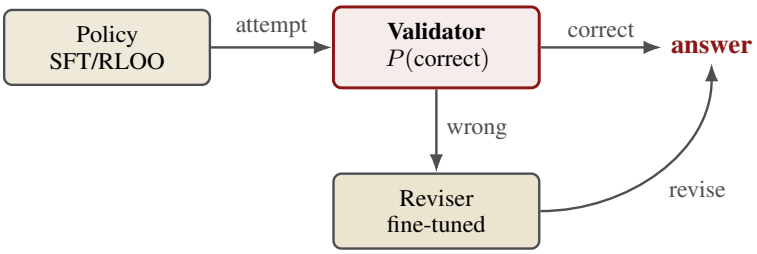


Figure 1: Generative verifier



(a) Keep only verifier-passing revisions of wrong attempts (4,653 examples, 36% kept).



(b) Verifier decides when to stop and what feedback to pass.

Figure 2: The reviser.

selector that always picks a correct candidate when one exists (pass@N), which is the ceiling for any selector.

3.3 Teaching the model to revise

Small models are notoriously bad at self-correction [8], so rather than prompt the policy to fix itself we train a dedicated reviser. The data is built entirely offline (Figure 2, top). We start from the wrong first attempts in our corpus. For each one we assemble a revision prompt that contains the original problem, the verbatim wrong attempt, and a feedback string, then sample eight revisions from the policy under the official decoder. We keep only the revisions that pass the Countdown scorer, which is a legal offline filter, and emit one training row per kept revision. This yields **4,653** revision examples at a 36% keep rate. We build the data in two feedback flavors that differ in exactly one substring, which keeps the later ablation clean: a critique flavor that inserts the verifier's natural-language critique (for example, "the equation '9 * 3 + 9' is invalid: it uses 9, which is not available with that multiplicity"), and a binary flavor that inserts only "incorrect, try again." The reviser is then trained with the same unchanged SFT loop.

At inference (Figure 2, bottom) the loop chains two learned models. The policy produces a first attempt, the verifier judges it, and if $P(\text{correct})$ is high we stop and return it. Otherwise the verifier's critique (or the fixed binary string) is fed to the reviser, which produces a new attempt, and we repeat for a few rounds.

Hybrid. Finally we ask whether splitting a fixed budget between the two is best. At a total budget B we allocate W parallel samples and D sequential revisions with $W + D = B$: the verifier ranks the W first attempts, and the best is revised D times. The endpoints recover the two pure strategies, so the sweep shows whether any mix wins.

4 Experiments

In this section we describe the datasets, evaluation metrics, and setup used to test our hypothesis.

4.1 Data

We use the Countdown arithmetic reasoning task from the default project pipeline. Each example contains a target number and a list of allowed numbers. The model must output an arithmetic expression that evaluates to the target while using each provided number exactly once. The final expression is expected to appear inside answer tags, which is what allows the verifier to evaluate the proposed solution. Difficulty in the official test split is the number of input numbers, which is either three or four.

For the default pipeline we use the provided Countdown SFT and RL datasets. The SFT data consists of prompts paired with solution completions, while the RL data consists of prompts and ground-truth metadata used by the verifier for evaluation.

For the train-time extension we create an additional synthetic distillation dataset. We generate 9,000 solvable Countdown problems split evenly across easy, medium and hard. The Qwen2.5-7B teacher generates solutions, and we keep only the traces that pass the Countdown verifier to use for further training.

For the test-time extension we build two further datasets, both offline. The verifier dataset pairs each policy-generated candidate with a single-token verdict and an error-type critique, labeled by the offline scorer; we train on roughly 132k rows and evaluate on a held-out split of 2,339 candidates. Negatives are real policy mistakes, not synthetic corruptions, so the verifier learns the failure modes it will actually see. The revision dataset is built by drawing eight revisions of each wrong attempt and keeping only the verifier-passing ones, which yields 4,653 revision examples (a 36% keep rate). All test-time evaluation uses the held-out test split with no leakage.

4.2 Evaluation method

We evaluate generation with pass@k. For each prompt we sample k candidates from the model, and a prompt is solved if at least one candidate passes the Countdown verifier. We report pass@1, pass@4, and pass@16. Pass@1 measures single-sample performance, while pass@4 and pass@16 measure whether the model can produce a correct solution given several attempts. For RLOO we report pass@1 at intermediate checkpoints, comparing vanilla RLOO, RLOO with curriculum, RLOO with distillation, and RLOO with both.

For the test-time experiments we use three families of metrics. For the verifier as a judge we report verdict accuracy, ROC-AUC, expected calibration error (ECE), Brier score, and the error-type accuracy of the generated critique. For selection we report the success rate of the verifier-chosen answer as a function of the sample budget, against the single-sample baseline and the oracle ceiling. For revision, we report the success rate at a matched generation budget and the recovery rate (the fraction of wrong first attempts that become correct after revision).

4.3 Experimental details

For train-time experiments we compare default SFT against our teacher-distilled SFT. We then run RLOO under four settings:

1. Vanilla RLOO
2. RLOO with Curriculum Learning
3. RLOO with distillation
4. RLOO with curriculum + distillation

For test-time experiments we evaluate (1) the generative verifier as a judge and how its initialization affects quality, (2) whether fine-tuning the verifier was necessary versus in-context prompting, (3) best-of-N selection against single-sample and oracle baselines, (4) sequential self-revision against parallel sampling at a matched budget, and (5) the hybrid width-versus-depth budget sweep. All training and inference run on Modal H100 GPUs reusing the existing infrastructure.

Table 1: SFT: teacher distillation widens the solution distribution (pass@4 up) at a small cost to the single greedy guess (pass@1).

SFT model	pass@1	pass@4	pass@16
baseline	**0.34**	0.60	0.78
teacher-only	0.32	**0.68**	0.78

Table 2: RLOO pass@1 at step 40. Only the combination of curriculum and distillation clears vanilla RLOO.

Variant	pass@1
Vanilla RLOO	0.500
RLOO + curriculum	0.460
RLOO + distillation	0.500
RLOO + curriculum + distillation	**0.540**

5 Results and Analysis

5.1 Train time: distillation and curriculum help only together

Table 1 compares the baseline SFT model against the teacher-distilled SFT model. Pass@4 rises from 0.60 to 0.68, which indicates that the distilled model learns a broader and more useful solution distribution, even though pass@1 dips slightly. The 8-point gain at pass@4 is not transformational, but it was an early signal that a multi-candidate, test-time approach was worth exploring.

Table 2 shows the RLOO results. Vanilla RLOO and RLOO with distillation alone both reach 0.500 pass@1, so teacher traces alone do not translate into better on-policy performance. Curriculum alone slightly worsens performance to 0.460, which we attribute to the easy-first ordering reducing early prompt diversity and encouraging overfitting to easy examples. The best result comes from combining curriculum and distillation at 0.540, which suggests the two are complementary: distillation gives the model better solution patterns, while the curriculum makes the RLOO updates less sparse.

5.2 The verifier is a near-perfect judge

Table 3 reports verifier quality across three initializations that differ only in their starting checkpoint. All three reach about 98% verdict accuracy and ROC-AUC near 0.998, and the generated critique names the true error type about 98% of the time, so this is a real generative verifier and not just a binary head. The single-token verdict design never failed: the verdict token was always readable, so $P(\text{correct})$ is always a clean logit. Initialization barely matters for accuracy or AUC. The only real effect is calibration: base-initialized ECE (0.0117) is about twice the domain-adapted inits (0.006), visible in Figure 3. We therefore use the SFT-initialized verifier downstream, since it is marginally best on accuracy and AUC and tied-best on calibration.

5.3 Fine-tuning the verifier was necessary

A natural objection is that the verifier's ability could be prompted in rather than trained in. Table 4 and Figure 4 answer this directly by prompting the same checkpoint with 0, 5, 10, and 20 in-context examples. Zero-shot is at chance (accuracy 0.542 against a 0.520 base rate): the latent signal exists

Table 3: Generative-verifier quality on 2,339 held-out candidates by initialization (standard error below 0.004). Initialization barely affects accuracy or AUC; it mainly affects calibration.

init	verdict acc	ROC-AUC	ECE ↓	Brier ↓	reason acc
SFT (used)	**0.983**	**0.998**	0.0062	**0.0137**	**0.982**
RLOO	0.982	0.997	**0.0061**	0.0146	0.981
base Qwen	0.982	0.998	0.0117	0.0149	0.981

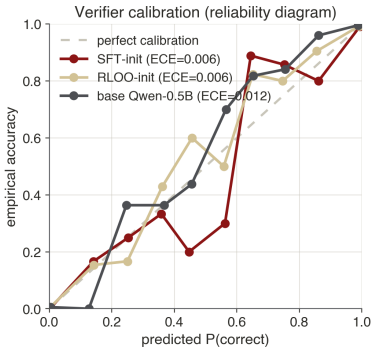


Figure 3: Reliability diagram.

Table 4: In-context prompting versus the fine-tuned verifier on the same 2,339 candidates. ICL plateaus well below fine-tuning and never reproduces the critique.

metric	0-shot	5-shot	10-shot	20-shot	**fine-tuned**
verdict accuracy	0.542	0.584	0.765	0.752	**0.983**
ROC-AUC	0.762	0.812	0.857	0.874	**0.998**
ECE ↓	0.223	0.241	0.063	0.146	**0.006**
reason: error-type	0.000	0.086	0.417	0.519	**0.983**

(AUC 0.76) but the decision is a coin flip. In-context examples help but plateau far below fine-tuning, reaching about 0.77 accuracy and 0.87 AUC by 10-shot and then stopping; 20-shot does not improve accuracy. The decisive gap is the critique: error-type accuracy is 0.00 at 0-shot and only 0.52 at 20-shot, versus 0.98 fine-tuned. Generating a correct, formatted explanation is a capability that has to be trained into the weights, not prompted in. The forced-verdict readout was equally valid for both models (no missing verdict tokens), so the gap is real and not a measurement artifact.

5.4 Selection: a learned verifier is almost an oracle

The SFT policy generated $N = 16$ candidates for 200 test problems (3,200 candidates), and the verifier ranked them by $P(\text{correct})$. Table 5 and Figure 5 show that selecting with the verifier more than doubles pass@1, from 0.316 ± 0.033 to 0.750 ± 0.031, capturing about 93% of the single-to-oracle gap. The verifier line tracks the oracle ceiling at every budget ($N = 1, 2, 4, 8, 16$), so it extracts nearly all the available signal across the whole range, not just at one point. Acting as a judge on these deployment candidates, the verifier reaches accuracy 0.973, ROC-AUC 0.995, and ECE 0.010.

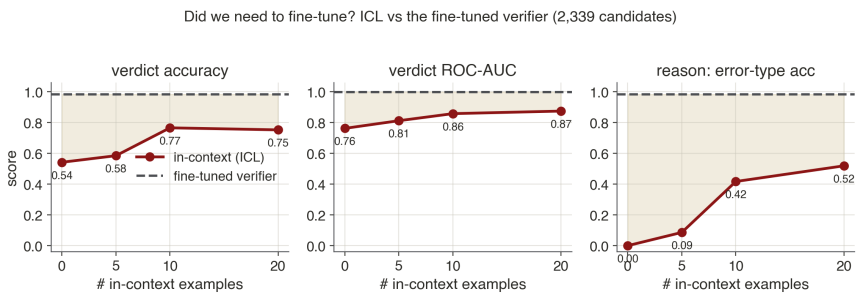


Figure 4: In-context learning (solid) against the fine-tuned verifier (dashed). ICL recovers some ranking ability but plateaus on accuracy and AUC and barely produces the structured critique.

Table 5: Best-of-N selection (mean $\pm$ standard error over 200 prompts; the curve also carries one bootstrap std over 200 subset draws per budget). The verifier sits within a few points of the oracle at every budget.

budget N	1	2	4	8	16
single-sample	0.316	0.316	0.316	0.316	0.316
learned verifier	0.316	0.470	0.609	0.704	**0.750**
oracle (pass@N)	0.316	0.473	0.617	0.727	0.780

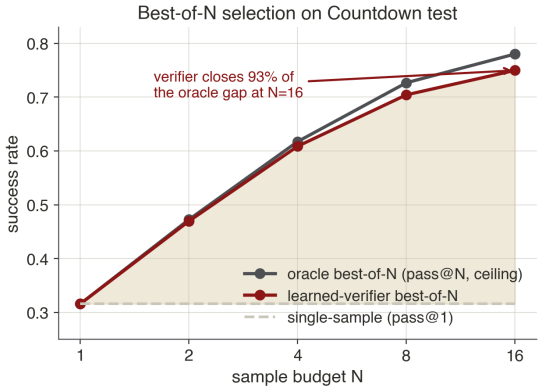


Figure 5: The learned verifier (red) hugs the oracle ceiling (gray) at every sample budget and far exceeds the single-sample baseline (dashed).

5.5 Revision underperforms sampling at every budget

The selection result motivated the opposite experiment: instead of selecting among candidates, spend the budget improving one. We compare, at a fixed generation budget B, parallel best-of-N against sequential revision. Both start identically at $B = 1$, so any divergence is the strategy's doing. To give revision every advantage we used oracle feedback and oracle stopping, which is the ceiling for any deployable revision loop.

Table 7 and Figure 6 show that parallel best-of-N wins at every budget of two or more, and the gap widens with budget (about $+12$ points at $B = 4$). A revision conditioned on a wrong attempt recovers it less often than an independent re-sample lands a correct answer: recovery of the 550 wrong first attempts is only 0.258 ± 0.019 (critique) and 0.267 ± 0.019 (binary) even after three rounds. The two reviser lines sit on top of each other, so the natural-language critique adds nothing over "try again" for revision. This is a clean negative ablation: the verifier's critique is worth a lot as a selection signal but nothing as a revision signal. The finding is robust precisely because we used the oracle-feedback ceiling, so any real verifier-fed loop can only do worse.

5.6 Hybrid: width beats depth

Finally we ask whether a split could beat either pure strategy. Figure 7 sweeps the allocation of a fixed budget between width W (parallel samples) and depth D (revisions). Success rises monotonically as budget moves from depth into width: every curve climbs toward the pure best-of-N endpoint. Table 8 summarizes the budget $B = 7$ case. The specific "resample 4, then revise the best 3 times" idea

Table 6: Selection by difficulty ($N = 16$). The verifier captures almost all of the oracle on both, so the 4-number weakness is a generation problem, not a selection problem.

difficulty	single	verifier@16	oracle@16	verifier AUC
3 numbers	0.467	0.958	1.000	0.995
4 numbers	0.172	0.557	0.577	0.988

Table 7: Matched-budget comparison over 800 trajectories (mean $\pm$ standard error). Parallel best-of-N beats sequential revision at every budget of two or more, and the critique buys nothing over a bare retry.

budget B	parallel best-of-N	revision (critique)	revision (binary)
2	**0.472** $\pm$ 0.018	0.409 $\pm$ 0.017	0.419 $\pm$ 0.017
3	**0.562** $\pm$ 0.018	0.461 $\pm$ 0.018	0.471 $\pm$ 0.018
4	**0.619** $\pm$ 0.017	0.490 $\pm$ 0.018	0.496 $\pm$ 0.018

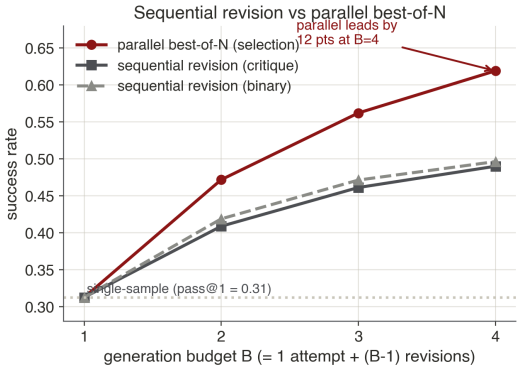


Figure 6: At a matched generation budget, parallel best-of-N (red) stays above both sequential revision variants (gray). The gap grows with budget.

(0.658) is clearly worse than spending all seven on sampling (0.683). The best split keeps exactly one revision and beats pure sampling by only 0.2 to 0.3 points, which is inside the Monte-Carlo and small-test-set noise. Every split with two or more revisions is strictly worse.

This is the unifying conclusion across the test-time results: the 0.5B policy's bottleneck is generating a correct candidate, and the verifier is an excellent selector, so the efficient recipe is to sample broadly and let the verifier select, not to revise.

6 Qualitative Analysis

The verifier does not only say whether an answer is wrong, it says how. Its critique is trained against four error types from the offline diagnosis, summarized with real critique templates in Table 9. The three failure modes are distinct: `malformed` answers have no usable equation in the tags, `invalid_numbers` answers use a number too many times or drop one, and `wrong_value` answers are well formed and use the right numbers but evaluate to something other than the target. Distinguishing these is exactly the structured critique that in-context prompting failed to reproduce.

To make the headline results concrete, Figure 8 shows representative behavior on a 3-number problem. The policy is unreliable on any single draw: across its 16 samples we see all three failure modes, an answer that uses a number twice, a malformed answer with no equation, and several that are well formed but evaluate to the wrong value, all wrapped in confident tags. The generative verifier reads

Table 8: Budget $B = 7$ allocation (50 prompts, 128 Monte-Carlo draws per prompt). Pure best-of-N is at or above every split; the best-split edge is within noise.

strategy ($B = 7$)	success
pure revision (W=1, D=6)	0.547
resample 4 + revise 3 (W=4, D=3)	0.658
best split (W=6, D=1)	0.685
pure best-of-N (W=7, D=0)	**0.683**

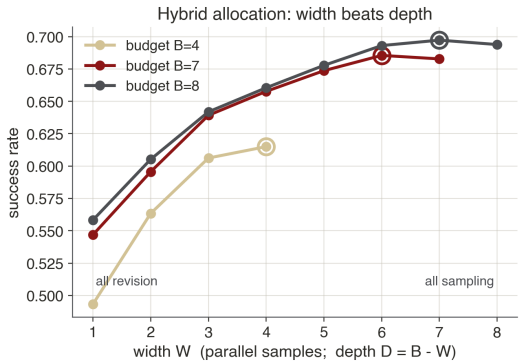


Figure 7: Hybrid budget allocation. For every budget, success climbs left to right as the split shifts from depth (revision) to width (sampling). The optimum hugs the pure best-of-N end.

Table 9: The four error types the verifier learns to name, with representative critiques. Only `ok` is correct; the others are the policy's three ways of failing.

error type	what went wrong	representative critique
`malformed`	no usable equation in tags	"no <answer> tags found; the response does not contain a final answer"
`invalid_numbers`	wrong set or count of numbers	"the equation '9 * 3 + 9' is invalid: it uses 9, which is not available with that multiplicity"
`wrong_value`	right numbers, wrong result	"the equation '7 * 3 + 9' evaluates to 30, not the target 34"
`ok`	correct	"the equation '9 * 3 + 7' correctly evaluates to the target 34"

each candidate, assigns a low $P(\text{correct})$ to the wrong ones and a high score to the correct one, and labels the specific failure in each case. Selection then returns the one good candidate, which is why best-of-N nearly matches the oracle. The contrast with revision is the instructive part: when a wrong attempt is fed back, the 0.5B reviser tends to produce another wrong expression of the same flavor rather than genuinely repairing the arithmetic, which is exactly why an independent fresh sample is the better use of the same budget.

7 Conclusion

We set out to answer where a fixed compute budget pays off for a small Countdown reasoner. The answer is clear: spend it at test time, on sampling. A learned generative verifier turns a 0.316 pass@1 policy into a 0.750 solver at a 16-sample budget, about 93% of the oracle ceiling, while staying well calibrated. Fine-tuning that verifier was necessary, since in-context prompting cannot reach the same accuracy or produce the structured critique. Self-revision, by contrast, loses to plain sampling at every matched budget, and a hybrid split never beats pure sampling beyond the noise. Train-time distillation and curriculum help only in combination, and only modestly. The single explanation behind all of this is that for a 0.5B model the bottleneck is generating a correct candidate, not recognizing one.

Problem. Use `[3, 7, 9]` exactly once to reach **34**.

Policy samples (best-of-16, illustrative).

`<answer>7 * 3 + 9</answer>`	P=0.04	`wrong_value`: "evaluates to 30, not 34"
`<answer>9 * 3 + 9</answer>`	P=0.03	`invalid_numbers`: "uses 9 twice; 7 is unused"
`<answer> </answer>`	P=0.02	`malformed`: "no usable equation in tags"
`<answer>(7 - 3) * 9</answer>`	P=0.06	`wrong_value`: "evaluates to 36, not 34"
`<answer>9 * 3 + 7</answer>`	$\mathbf{P{=}0.97}$	`ok`: "correctly evaluates to 34"

Selection returns `9 * 3 + 7 = 34` (verifier-ranked best).

Figure 8: Representative selection-versus-revision behavior.

The main limitations are the small official test splits and a single decoding seed for candidate generation, which we addressed with bootstrap and Monte-Carlo resampling but did not fully remove. The most promising future direction follows directly from the difficulty split: since 4-number generation is the true bottleneck, compute is best invested in improving the policy's coverage on hard instances rather than in deeper revision.

Team contributions

Manat Kaur led the train-time experiments, including the synthetic distillation dataset and the curriculum and RLOO comparisons, and wrote the corresponding sections. Felipe Teixeira led the test-time extension, including the generative verifier, the selection, revision, and hybrid evaluations, the variance analysis and the poster. Both authors jointly designed the study and wrote the report.

Generative AI use

We used generative AI for brainstorming, debugging, plotting, and prose editing/formatting. In line with the project specifics, we also used AI to help aid our extension.

References

[1] Daya Guo, Dejian Yang, Haowei Zhang, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[2] Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*, 2025.

[3] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024.

[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *International Conference on Machine Learning (ICML)*, 2009.

[5] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[7] Hunter Lightman, Vineet Kosaraju, Yura Burda, et al. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

[8] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations (ICLR)*, 2024.