LEARNING AT AND BEYOND THE COMPETENCE BOUNDARY FOR ON-POLICY POST-TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

On-policy post-training adapts learning to a model's current behavior, while adaptive curriculum learning adjusts task selection to its evolving competence. Yet adapting responses and selecting from existing tasks still leaves a critical gap: persistent failures reveal capability deficits but do not by themselves create the new practice needed to overcome them. To address this gap, we propose **CASE**, a **C**ompetence-**A**ware **S**election with **E**rror-targeted generation framework for adaptive on-policy post-training. **CASE** adapts both the response and task distributions: at each round, it retains tasks near the student's current competence boundary and uses a teacher agent to diagnose recurring failures and generate verified intermediate problems targeting the underlying weaknesses. The student then learns from freshly sampled responses on the selected and generated tasks using group relative policy optimization with executable rewards. We evaluate **CASE** with Qwen-3.5-9B across code, math, and finance, including analyses of task selection, error-targeted generation, and teacher choice. The results show complementary gains from selection and generation, with an average pass@1 improvement of 7.69 percentage points over the base model on out-of-domain benchmarks.

1 INTRODUCTION

Large language models acquire capabilities through successive training stages (Ouyang et al., 2022), shaped by two central choices: how they learn and what tasks they learn from. Pretraining primarily relies on large-scale static corpora (Brown et al., 2020), while conventional post-training typically uses demonstrations, preferences, or tasks collected in advance (Ouyang et al., 2022; Rafailov et al., 2023) . This offline setup has motivated more adaptive forms of post-training. Two directions are particularly relevant. On-policy distillation (OPD) adapts supervision to the model's current behavior (Agarwal et al., 2024) by providing teacher feedback on student-generated trajectories, whereas curriculum learning (Bengio et al., 2009) adapts the training tasks to the student's evolving competence (Graves et al., 2017).

These two forms of adaptation operate at different levels, yet each leaves a complementary mismatch unresolved. ***From the learning perspective***, OPD reduces the train-inference mismatch (Song & Zheng, 2026), but typically keeps the task distribution fixed. As the student improves, mastered tasks become less informative and overly difficult ones remain poorly matched to its competence (Kumar et al., 2010). ***From the data perspective***, adaptive curriculum learning addresses this task-competence mismatch (Graves et al., 2017; Zhang et al., 2025), but remains limited to the existing task pool and cannot convert persistent failures into targeted new practice. Under GRPO with binary execution rewards (Guo et al., 2025), mixed-outcome tasks yield non-zero within-group advantages, whereas all-pass and all-fail tasks provide no reward contrast. Rollout outcomes therefore offer a practical signal of evolving competence, as illustrated in Figure 1. Mixed-outcome tasks may approximate the competence boundary, and persistent failures indicate capability gaps beyond it.

However, identifying a capability gap does not by itself make it learnable. We therefore view adaptive on-policy post-training as requiring two complementary forms of task adaptation. *Selection* determines which existing tasks remain useful for direct optimization, and *generation* creates new practice beyond the current task pool. Recent failure-driven methods generate targeted tasks from model errors (Wang et al., 2026), but do not explicitly adapt which existing tasks should remain in the active curriculum. Generation expands the curriculum, whereas selection concentrates optimization

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

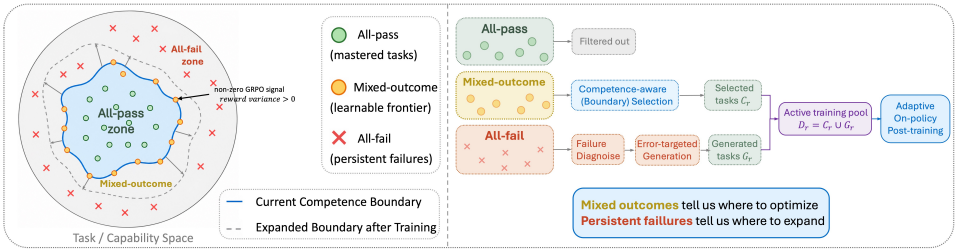


Figure 1: Empirical Competence Boundary from On-policy Rollouts.

on informative tasks. These curriculum roles are inherently dynamic as the student evolves. A task that is initially unsolved may later become suitable for direct optimization and eventually become mastered, requiring its role to be reassessed throughout training.

Building on this view, we introduce **CASE**, a **C**ompetence-**A**ware **S**election with **E**rror-targeted generation framework. We characterize the student's current competence boundary through mixed-outcome tasks, where the student succeeds on some rollouts but fails on others, and retain these tasks for direct optimization. Persistent failures instead reveal unresolved capability gaps and guide a teacher agent to generate new verified problems. Rather than providing target responses or token-level supervision, the teacher serves as a black-box curriculum generator, and the student learns from fresh on-policy responses using execution-based GRPO. In this way, **CASE** adapts the training curriculum while keeping learning grounded in the student's own on-policy experience.

We evaluate **CASE** using `Qwen-3.5-9B` as the base model across code, math, and finance benchmarks. Our experiments address three research questions. **(i)** Does competence-aware construction improve on-policy post-training? **(ii)** What does error-targeted generation contribute beyond boundary selection alone? **(iii)** How does teacher choice affect the effectiveness of our method? These questions examine the effects of competence-aware task selection, targeted curriculum expansion, and teacher choice on adaptive on-policy post-training. Our contributions are threefold:

- We identify two complementary dimensions of adaptive on-policy post-training, adapting responses to the current policy and adapting tasks to the student's evolving competence.
- We propose **CASE**, which combines competence-aware selection with error-targeted generation. It retains mixed-outcome tasks near the competence boundary and turns persistent failures into new verified practice.
- We conduct an extensive evaluation of **CASE** with 3 teacher models across 3 code validation benchmarks and 9 out-of-domain benchmarks. Our experiment systematically examines the effects of task selection, failure-targeted task generation, and teacher choice. The results show that **CASE** provides complementary gains, achieving an average +7.69 pass@1 improvement over the base model on out-of-domain benchmarks.

2 Related Work

2.1 On-policy distillation

Conventional distillation (Hinton et al., 2015) for autoregressive language models is predominantly off-policy, training students on fixed or teacher-generated sequences and thus introducing train–inference mismatch (Song & Zheng, 2026). On-policy distillation (OPD) instead learns from student-generated trajectories, allowing the training signal to better reflect the student's current behavior (Agarwal et al., 2024). Existing OPD methods differ mainly in how the teacher signal is obtained. *White-box OPD* uses teacher logits or probabilities, with GKD (Agarwal et al., 2024) matching token-level distributions, MiniLLM (Gu et al., 2024) optimizing sequence-level reverse KL, and DistiLLM (Ko et al., 2024) improving stability with skewed KL objectives. *Black-box OPD* relies on teacher-generated text or scalar feedback when full teacher distributions are inaccessible (Jiang et al., 2023; Ye et al., 2025). *Self-distillation* further removes the external teacher and derives signals from the model itself (Shenfeld et al., 2026; Hübotter et al., 2026). Across these settings, OPD adapts learning over student-generated responses, whereas our method adapts the tasks the student encounters.

2.2 Curriculum Learning

Curriculum learning organizes training examples or tasks according to their learning difficulty, typically progressing from easier to harder instances (Bengio et al., 2009). A general curriculum consists of a *difficulty measurer* and a *training scheduler*, which respectively estimate task difficulty and control their sampling or pacing during training (Wang et al., 2021). Recent work has moved from predefined, model-agnostic curricula (Weinshall et al., 2018) toward adaptive curricula that respond to the learner's evolving competence (Vakil & Amiri, 2023; Zhang et al., 2025). Teacher–student approaches further select tasks according to learning progress (Matiisen et al., 2017), while recent curriculum RL methods can generate challenging yet achievable goals (Sayar et al., 2024). Our method differs by using persistent student errors to both retain informative tasks and generate targeted new problems beyond the existing pool.

3 Preliminary

3.1 On-Policy Post-Training with GRPO

Let $\mathcal{D}$ denote the current training-task distribution and π_θ the student policy. For each task $x \sim \mathcal{D}$, the behavior policy $\pi_{\theta_{\text{old}}}$ samples a group of G responses,

$$y_i \sim \pi_{\theta_{\text{old}}}(\cdot \mid x), \qquad i = 1, \ldots, G. \tag{1}$$

Each response is evaluated by a verifiable reward function $r_i = \mathcal{R}(x, y_i)$. We focus on tasks with binary outcome rewards $r_i \in \{0, 1\}$, where $r_i = 1$ indicates a correct response and $r_i = 0$ otherwise. Such rewards can be obtained automatically from deterministic verification procedures, including exact-answer matching and execution-based evaluation.

Group relative policy optimization (GRPO) (Guo et al., 2025) estimates each response's advantage relative to the rewards of other responses sampled for the same task, avoiding a separate value model. Given group rewards $\{r_i\}_{i=1}^G$, the advantage is

$$A_i = \frac{r_i - \frac{1}{G}\sum_{j=1}^G r_j}{\text{Std}_{j\in[G]}[r_j] + \epsilon}, \tag{2}$$

where ϵ is a small constant for numerical stability. GRPO adopts the clipped surrogate objective of proximal policy optimization (PPO) (Schulman et al., 2017). For token $y_{i,t}$ in response y_i, define the policy ratio as

$$\rho_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})}. \tag{3}$$

Using $\mathbb{E}_{i,t}$ to denote averaging over sampled responses and their tokens, the GRPO objective is

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{i,t}[\min(\rho_{i,t}A_i,\ \text{clip}(\rho_{i,t}, 1-\varepsilon_c, 1+\varepsilon_c)A_i)]\,. \tag{4}$$

where ε_c denotes the clipping range. As training proceeds, $\pi_{\theta_{\text{old}}}$ is refreshed from the evolving student policy, so newly collected rollouts continually reflect the student's current behavior.

3.2 Task-Level Outcome Signals

The group structure of GRPO also provides a natural signal for characterizing the student's competence on each task. We summarize the outcomes of task x by its empirical success rate

$$\bar{r}(x) = \frac{1}{G}\sum_{i=1}^G r_i. \tag{5}$$

Under binary rewards, the rollout group falls into three regimes:

$$\bar{r}(x) = 1, \qquad 0 < \bar{r}(x) < 1, \qquad \bar{r}(x) = 0, \tag{6}$$

corresponding to all-pass, mixed-outcome, and all-fail groups, respectively. For all-pass and all-fail groups, identical rewards yield $A_i = 0$ in Equation 2, whereas mixed-outcome groups provide non-zero group-relative signals. These regimes evolve with the student policy: a task may move from all-fail to mixed-outcome and eventually to all-pass as training progresses. Hence, task informativeness must be reassessed throughout training, motivating the adaptive curriculum in Section 4.1.

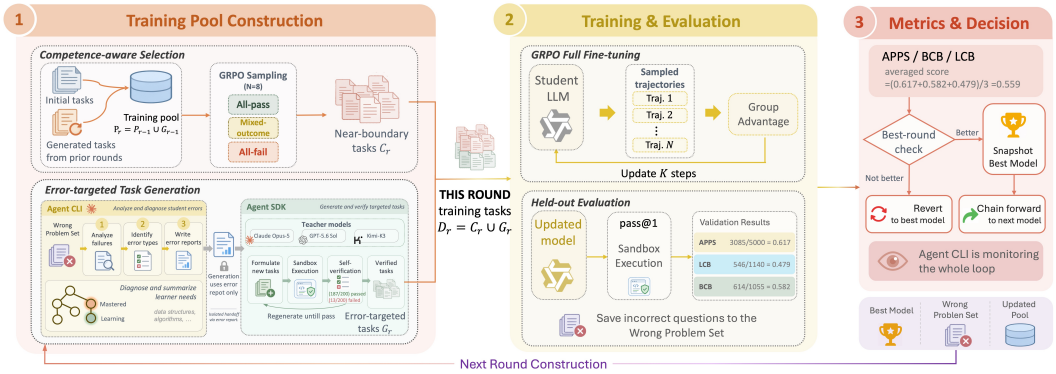


Figure 2: **One round of CASE.** The round-initialized student first filter accumulated task pool according to its rollout outcomes, and the resulting set is combined with Agent SDK's targeted tasks for post-training on fresh on-policy trajectories. Subsequent evaluation provides the wrong problem set for the next round of error analysis, and the averaged score on validation sets determines the model initialization for the next round.

4 METHOD

We improve the reasoning capability of a student LLM through successive rounds of adaptive on-policy post-training, as illustrated in Figure 2. Each round adapts the task distribution in two complementary ways. Rollout outcomes on the accumulated task pool identify existing tasks that remain informative for direct optimization, while failures on held-out validation sets reveal recurring capability gaps for targeted task generation. The selected and generated tasks are combined into an active training pool, on which the student is optimized with fresh on-policy trajectories using GRPO. After training, evaluation provides the failure feedback and model-selection signal for the next round.

4.1 ADAPTIVE CURRICULUM

Let $\mathcal{P}_r$ denote the full task pool accumulated before round r. We evaluate $\mathcal{P}_r$ with the current student and retain tasks $\mathcal{C}_r \subseteq \mathcal{P}_r$ that are matched to its current competence. Agent SDK generates and verifies a task set $\mathcal{G}_r$ from the error reports produced by Agent CLI. Together, the two subsets form the active training pool for round r: $\mathcal{D}_r = \mathcal{C}_r \cup \mathcal{G}_r$.

Competence-aware Selection. The first round serves as a warm-up over the initial task pool, with $\mathcal{C}_0 = \mathcal{P}_0$ and $\mathcal{G}_0 = \varnothing$. For each subsequent round, the round-initialized student samples K responses $y_i \sim \pi_{\theta_r}(\cdot \mid x)$ for every task $x \in \mathcal{P}_r$. Each response is evaluated by the binary verifier $\mathcal{R}$ through the task's associated test suite. Following the task-level outcome regimes characterized in Section 3.2, we compute the round-specific empirical success rate $\widehat{p}_r(x)$ and retain tasks:

$$\widehat{p}_r(x) = \frac{1}{K}\sum_{i=1}^{K} \mathcal{R}(x, y_i), \qquad \mathcal{C}_r = \{x \in \mathcal{P}_r : 0 < \widehat{p}_r(x) < 1\}. \tag{7}$$

Thus, $\mathcal{C}_r$ retains tasks on which the current student exhibits mixed outcomes, providing empirical evidence that they lie near its current competence boundary and avoiding the degenerate group-relative signals associated with all-pass or all-fail groups. Re-estimating $\widehat{p}_r(x)$ in each round allows the curriculum to track the student's evolving competence,yielding a curriculum that adapts to the evolving policy rather than following a fixed easy-to-hard ordering.

Error-targeted task generation. Consistently failed tasks provide little group-relative signal yet reveal systematic capability gaps, which we use to guide targeted task generation. For each round $r > 0$, Agent CLI (command-line interface), which serves as the failure-diagnosis agent, analyzes the wrong problem set $\mathcal{W}_{r-1}$ collected from the preceding evaluation and groups the student's failures by algorithmic structure and likely failure mode. It produces dataset-specific error reports that summarize recurring conceptual gaps, reasoning errors, implementation pitfalls, and missed edge cases.

We explicitly separate this diagnostic stage from task generation. Agent SDK (software development kit), which serves as the task-generation agent, receives only these reports, together with dataset-specific format examples and the requested task counts, but has no access to the original problems in $\mathcal{W}_{r-1}$, preventing direct reformulation of failed instances. The reports specify which capability gaps to target, while the examples constrain generated tasks to the schema and evaluation format of the corresponding dataset.

Each generated task x is accompanied by a reference solution s_x and a test specification z_x. Depending on the dataset, evaluation may be based on input-output cases, function calls, or unit tests. Let τ_x denote the corresponding evaluation interface. Each generated task is verified in a sandbox using the same execution-based grader employed during training, and only tasks satisfying

$$\mathrm{Verify}(s_x, \tau_x, z_x) = 1 \tag{8}$$

are retained; invalid and duplicate tasks are discarded. The accepted tasks constitute $\mathcal{G}_r$, and are added to the cumulative task pool for subsequent rounds through $\mathcal{P}_{r+1} = \mathcal{P}_r \cup \mathcal{G}_r$.

4.2 Curriculum-Guided On-Policy Training

Given the adaptive training pool $\mathcal{D}_r$, we optimize all student parameters using standard GRPO. Our method does not modify the learning objective; instead, it adapts the task distribution on which on-policy optimization is performed. For each task $x \in \mathcal{D}_r$, the current student samples a group of G responses, which are evaluated by the binary execution reward and optimized using the group-normalized advantages. After curriculum construction, GRPO is performed using fresh on-policy responses from the current student, independent of those used for task selection.

Rather than using teacher solutions for supervised fine-tuning (Ouyang et al., 2022) or imitation learning (Gudibande et al., 2023), we train the student entirely from its own responses and execution-based rewards. For each generated task, the reference solution is used only to verify correctness and executability, and is never used as a training target. For each generated task, the reference solution is used only to verify correctness and executability, and is not used as a training target. The student instead generates its own group of responses and learns from their verifiable outcomes through group-relative advantages. Thus, the teacher determines *what the student should practice*, and GRPO learns from the student's own on-policy experience.

4.3 Closed-Loop Adaptation

Inspired by auto-research (Karpathy, 2026), we organize post-training as a closed loop across rounds, where evaluation feedback guides model progression and prevents substantial regressions from carrying forward. At round r, the round-initialized student θ_r^{in} first evaluates the cumulative task pool to obtain the competence-aware subset $\mathcal{C}_r$, which is combined with the generated tasks to form the training pool $\mathcal{D}_r$. The student is optimized on $\mathcal{D}_r$, producing the updated model θ_r. We then evaluate θ_r on the held-out validation sets and obtain two forms of feedback: the wrong problem set $\mathcal{W}_r$, which reveals capability gaps for the next round, and an average validation score s_r, which indicates whether training has improved overall performance.

We therefore maintain the best-performing student observed so far. If the current student achieves a new best validation score, it replaces the historical best. Otherwise, it is still used to initialize the next round unless its score falls more than a tolerance δ below the historical best, in which case the current update is discarded and training reverts to the best-performing model. Formally, let $s_r^{\mathrm{best}} = \max_{0 \le j \le r} s_j$ and θ_r^{best} denote the corresponding model. The next round is initialized as

$$\theta_{r+1}^{\mathrm{in}} = \begin{cases} \theta_r, & s_r \ge s_r^{\mathrm{best}} - \delta, \\ \theta_r^{\mathrm{best}}, & s_r < s_r^{\mathrm{best}} - \delta. \end{cases} \tag{9}$$

Meanwhile, Agent CLI analyzes $\mathcal{W}_r$ to guide the construction of $\mathcal{G}_{r+1}$, which expands the cumulative pool $\mathcal{P}_{r+1}$. The selected student then reassesses this updated pool to construct $\mathcal{C}_{r+1}$ and the next training pool $\mathcal{D}_{r+1}$. In this way, evaluation closes the loop by jointly controlling model progression and curriculum adaptation across rounds.

5 EXPERIMENTS

5.1 EXPERIMENTAL DESIGN

Our evaluation covers code, math and finance benchmarks, which assess structured reasoning under different task formats and correctness signals. Together, they test whether training on executable code problems improves the student's reasoning performance across domains.

Training, validation, and test datasets. We use APPS-train (Hendrycks et al., 2021) as the initial training pool because its executable tests support automatic reward computation and competence-aware task selection. Code reasoning is evaluated on APPS-test, LiveCodeBench (LCB) (Jain et al., 2025), and BigCodeBench (BCB) (Zhuo et al., 2025) to cover complementary sources of difficulty and execution environments. APPS supports both standard-input and call-based tests, LCB contributes recent contest problems, and BCB adds function-level tasks with dependencies drawn from 139 libraries. Together, they test whether improvements extend beyond the training distribution. After each round, the mean pass@1 across these benchmarks determines checkpoint carryover. Errors from the same evaluation guide task generation for the next round. To assess generality across domains, we also evaluate mathematical and financial reasoning benchmarks, which do not participate in training or the adaptation loop. Details of the training, validation, and test datasets are provided in Table 1.

Table 1: Overview of the datasets used for training, validation, and cross-domain testing.

Benchmark	Domain	Split	# Tasks	# Problem Tokens*	# Tests / Problem*	Dependencies	Release (version)
APPS-train (Hendrycks et al., 2021)	code	train	4449	395.6/13606	4.3/12	✗	2021
APPS-test (Hendrycks et al., 2021)	code	val	5000	501.0/1852	7.9/12	✗	2021
LiveCodeBench (Jain et al., 2025)	code	val	1055	152.9/1291	5.6/15	✗	2025 (release_v6)
BigCodeBench (Zhuo et al., 2025)	code	val	1140	599.1/2953	11.7/12	✓2.8 libs/task†	2024 (instruct)
CodeContests (Li et al., 2022)	code	test	276	638.6/1459	202.9/381	✗	2022 (validation+test)
GSM8K (Cobbe et al., 2021)	math	test	1319	61.2/188	–‡	✗	2021
MATH500 (Lightman et al., 2024)	math	test	500	72.3/831	–	✗	2023
AIME25 (Balunovic et al., 2026)	math	test	30	192.9/796	–	✗	2025
AIME24 (Hugging Face H4, 2024)	math	test	30	107.3/393	–	✗	2024
HMMT26 (Dekoninck et al., 2026)	math	test	33	94.4/217	–	✗	2026
HMMT25 (Dekoninck et al., 2026)	math	test	30	108.7/647	–	✗	2025
FinanceReasoning (Tang et al., 2025)	finance	test	2238	473.5/2480	–	✗	2025
FinChain (Xie et al., 2026)	finance	test	2,900	71.2/256	–	✗	2026

* Values in the *# Problem Tokens* and *Test Cases* columns are reported in Avg./Max. format.

† BigCodeBench contains 77 standard-library and 62 third-party libraries.

‡ indicates that executable test cases are not applicable to the benchmark.

Baselines. We consider both *inference only* and *post-training* baselines. For *inference-only* comparisons, we evaluate locally deployed open-source LLMs with substantially larger parameter counts than the student, including GPT-OSS-20B/120B (OpenAI, 2025), the Gemma 4 families (Team, 2026) and the Llama families (Meta AI, 2025; Grattafiori et al., 2024). We further include the frontier teacher models Claude Opus-5 (Anthropic, 2026), GPT-5.6-sol (OpenAI, 2026), and Kimi-K3 (Team et al., 2026), all accessed via APIs. The comparison results are reported in Table 3.

For *post-training* comparisons, we use `Qwen-3.5-9B` as the base model and apply GRPO Shao et al. (2024) on the original training data. We further include DAPO (Yu et al., 2026), whose dynamic sampling over-samples prompts and filters those with group accuracy of 0 or 1 to maintain batches with effective gradients. This provides a closely related training baseline that improves sample utilization through online filtering, while leaving the underlying task pool unchanged. Finally, we compare these baselines with our competence-aware selection variants under the same student model and training setting in Table 4. All results are reported as pass@1.

Implementation details. Across all experiments, post-training runs for six rounds, indexed from 0 to 5. Round 0 uses the complete APPS-train pool without difficulty filtering or teacher-generated tasks, and its evaluation errors provide the initial failure profile for subsequent targeted generation. From round 1 onward, the current student generates 8 rollouts per problem to estimate its problem-level competence. Problems with both successful and failed rollouts are retained and combined with 200 verified teacher-generated problems. Training is conducted on 4 NVIDIA H100 GPUs. Each round performs 15 steps with a batch size of 64 and 8 fresh on-policy responses per prompt, using a learning rate of 10^{-6} and a KL coefficient of 10^{-3}. Detailed settings for the maximum prompt and response lengths are provided in Appendix A.1.

5.2 Main Results

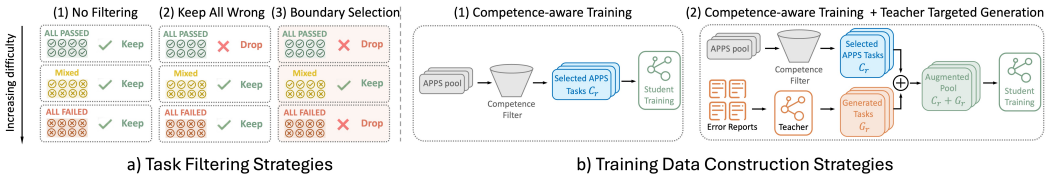


Figure 3: (a) Task filtering under *No Filtering*, *Keep All Wrong*, and *Boundary Selection*. (b) Competence-aware training versus teacher-targeted generation.

RQ1: Does competence-aware construction improve on-policy post-training? To examine how task selection shapes on-policy post-training, we compare three strategies for constructing the active training pool, as illustrated in Figure 3(a). *No Filtering* retains the entire cumulative pool; *Keep All Wrong* removes mastered tasks but keeps both mixed-outcome and all-fail tasks; and *Boundary selection* retains only mixed-outcome tasks.

Table 2: Round-wise training-pool sizes.

R	No Filtering	Keep All Wrong	Boundary Selection
0	4449 (initial)	4449 (initial)	4449 (initial)
1	4649 (+ teacher 200)	3162 (- all pass 1487)	1559 (- all fail 1603)
2	4849 (+ teacher 200)	3110 (- all pass 1739)	1529 (- all fail 1581)
3	5049 (+ teacher 200)	3030 (- all pass 2019)	1452 (- all fail 1578)
4	5249 (+ teacher 200)	2983 (- all pass 2266)	1366 (- all fail 1617)
5	5449 (+ teacher 200)	3069 (- all pass 2380)	1357 (- all fail 1712)

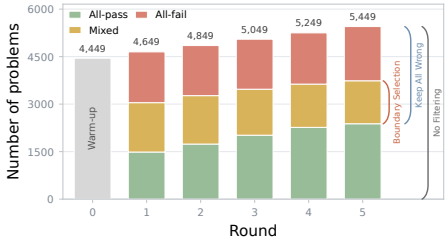


Figure 4: Training-pool composition

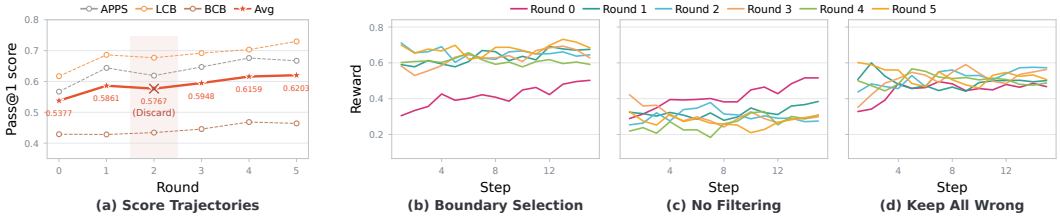


Figure 5: Score and reward trajectories under different filtering strategies. (a) Boundary selection strategy validation. (b-d) Training rewards under three filtering strategies.

To characterize how the three task-filtering strategies reshape the training process, we jointly analyze the round-wise active training-pool sizes in Table 2, the corresponding pool composition in Figure 4, and the validation and reward trajectories in Figure 5. The following observations are drawn from these complementary views of the training data and optimization dynamics:

1) *Progressive pool refinement.* The number of all-pass problems increases from 1,487 to 2,380 as the student masters more tasks. By retaining only mixed-outcome problems, *Boundary Selection* maintains the smallest active pool.
2) *A more informative optimization signal.* After the round-0 warm-up, *Boundary Selection* attains higher reward levels overall than *Keep All Wrong* and *No Filtering*. This pattern indicates that mixed-outcome tasks provide a denser training signal and is consistent with the stronger downstream performance for competence-aware training.
3) *Validation improvement with checkpoint control.* Under *Boundary Selection*, the average pass@1 across the three validation benchmarks generally improves over rounds, as shown in Figure 5(a). The regression at round 2 triggers checkpoint rejection and rollback to the previous best model.

Overall, competence-aware filtering improves on-policy post-training by concentrating optimization on a smaller, more informative set of tasks.

Table 3: Pass@1 comparison of inference-only baselines, teacher models, and CASE variants. Mean over 5 independent runs; mean ± standard deviation is reported in Table A.5.

Setup	Models	APPS	LCB(v6)	BCB	CodeCont.	GSM8K	MATH500	AIME25	AIME24	HMMT26	HMMT25	FinReason	FinChain
nference only	GPT-oss-20B	70.21	70.46	42.04	50.94	95.42	95.88	72.67	70.00	41.21	38.00	74.24	79.29
	GPT-oss-120B	79.34	76.13	38.68	60.51	96.12	97.00	80.67	77.33	57.58	56.00	82.65	83.54
	Llama-4-Scout-17B-16E	51.28	33.74	37.28	36.23	83.05	75.00	26.67	26.67	12.12	6.67	56.77	57.26
	Llama-3.3-70B-Instruct	36.18	33.84	33.84	28.43	84.26	77.08	4.67	26.00	4.24	2.00	59.62	60.28
	Llama-3.1-8B-Instruct	9.75	16.04	23.19	6.96	84.17	48.24	0.00	3.33	1.82	0.67	38.66	35.52
	Gemma-4-12B-it	49.12	68.24	37.47	35.07	95.01	93.52	62.00	72.67	43.03	50.00	76.37	65.48
	Gemma-4-31B-it	87.14	83.41	41.23	74.28	95.88	99.32	90.67	92.00	79.39	81.33	83.83	84.23
	Qwen-3.5-9B (base)	47.35	54.31	43.00	30.94	68.73	86.12	56.67	63.33	32.73	46.67	84.23	80.58
	Claude Opus-5	82.70	74.20	55.88	67.03	97.57	99.20	100.00	100.00	63.64	70.00	86.01	85.76
	GPT-5.6 Sol	89.30	87.68	54.04	81.88	95.98	98.80	100.00	100.00	84.85	86.67	92.23	86.14
	Kimi-K3	79.52	74.50	54.21	55.80	97.73	98.60	93.33	93.33	63.64	76.67	90.39	81.31
APPS + Teacher	**CASE** (with `Claude Opus-5` as teacher)												
	-w/- No Filtering	60.12 +12.77	63.37 +9.06	44.56 +1.56	41.29 +10.35	71.04 +2.31	88.24 +2.12	66.00 +9.33	55.33 -8.00	37.58 +4.85	48.00 +1.33	86.56 +2.33	81.61 +1.03
	-w/- Keep All Wrong	66.97 +19.62	71.68 +17.37	46.16 +3.16	40.58 +9.64	69.37 +0.64	88.04 +1.92	60.67 +4.00	67.33 +4.00	40.61 +7.88	46.67 +0.00	86.77 +2.54	81.33 +0.75
	-w/- **Boundary Selection**	66.33 +18.98	69.48 +15.17	45.32 +2.32	43.91 +12.97	74.37 +5.64	90.44 +4.32	75.33 +18.66	67.33 +4.00	49.09 +16.36	48.67 +2.00	86.92 +2.69	82.37 +1.79
	CASE (with `GPT-5.6 Sol` as teacher)												
	-w/- No Filtering	67.28 +19.93	69.31 +15.00	45.28 +2.28	41.59 +10.65	72.92 +4.19	94.92 +8.80	60.00 +3.33	68.00 +4.67	44.85 +12.12	52.00 +5.33	86.53 +2.30	82.62 +2.04
	-w/- Keep All Wrong	66.76 +19.41	71.28 +16.97	45.53 +2.53	41.74 +10.80	73.60 +4.87	89.68 +3.56	60.67 +4.00	70.00 +6.67	37.58 +4.85	50.00 +3.33	87.53 +3.30	81.97 +1.39
	-w/- **Boundary Selection**	68.75 +21.40	72.36 +18.05	45.96 +2.96	45.36 +14.42	74.77 +6.04	88.76 +2.64	64.67 +8.00	72.00 +8.67	46.67 +13.94	56.67 +10.00	87.30 +3.07	82.52 +1.94
	CASE (with `Kimi-K3` as teacher)												
	-w/- No Filtering	64.59 +17.24	69.19 +14.88	45.12 +2.12	43.45 +12.51	76.09 +7.36	91.84 +5.72	62.00 +5.33	72.67 +9.34	43.03 +10.30	52.67 +6.00	86.99 +2.76	81.94 +1.36
	-w/- Keep All Wrong	62.48 +15.13	67.79 +13.48	45.04 +2.04	41.30 +10.36	71.78 +3.05	90.64 +4.52	61.33 +4.66	72.00 +8.67	40.61 +7.88	51.33 +4.66	86.80 +2.57	81.67 +1.09
	-w/- **Boundary Selection**	71.10 +23.75	72.40 +18.09	45.67 +2.67	47.10 +16.16	75.72 +6.99	93.84 +7.72	65.33 +8.66	72.00 +8.67	44.24 +11.51	52.00 +5.33	87.75 +3.52	82.46 +1.88

RQ2: What does Error-targeted Generation Contribute beyond Boundary Selection alone? Both settings retain problems that the student solves in some rollouts but fails in others. GRPO training on these selected tasks serves as *Boundary Selection Only*, while *Teacher-Targeted Generation* additionally trains on verified problems generated by a teacher model from the student's errors, as illustrated in Figure 3(b).

Our *targeted generation* pipeline separates error diagnosis from problem generation. Agent CLI summarizes the reasoning or algorithmic weaknesses revealed by the student's failed problems into an error-cause report. An isolated Agent SDK then generates new problems conditioned solely on this report, without access to the original problem instances, and the verified problems are added to the training pool as targeted practice.

For comparison, *untargeted generation* omits error analysis and generates problems using only general task-generation instructions, without access to failed problems or error-cause reports. The generated problems undergo the same verification procedure and are added in the same quantity into the training pool. Figure 6(b) shows that untargeted generation yields a simple, generic Fibonacci-style dynamic programming (DP) problem, whereas targeted generation produces a more challenging permutation-statistic DP problem requiring prefix-sum acceleration of its state transitions. This contrast highlights stronger relevance and potential training value beyond merely increasing difficulty.

Table 4: Boundary selected training versus teacher-targeted generation. All generation-enabled variants use `GPT-5.6 Sol` as the teacher. Results are reported as mean pass@1 over five runs.

Pipeline	Filtering	Teacher Gen.	Training Pool Data	APPS	LCB	BCB	CodeCont.	GSM8K	MATH500	AIME25	AIME24	HMMT26	HMMT25	FinReason	FinChain
Qwen-3.5-9B (base)				47.35	54.31	43.00	30.94	68.73	86.12	56.67	63.33	32.73	46.67	84.23	80.58
GRPO	No Filtering	None	APPS	59.02	66.35	45.35	39.49	69.60	88.60	43.33	60.00	45.45	46.67	84.05	81.38
	Keep All Wrong	None	APPS	63.66	68.90	43.86	38.04	70.05	88.00	63.33	66.67	42.42	43.33	84.72	81.28
	Boundary Selection	None	APPS	66.80	68.25	43.51	39.49	69.14	90.40	53.33	70.00	30.30	60.00	83.91	81.48
	Boundary Selection*	Untargeted	APPS + generated	67.24	71.00	45.26	40.94	72.63	93.80	60.00	70.00	45.45	56.57	84.18	82.14
DAPO	Dynamic Sampling†	None	APPS	67.68	67.49	45.79	44.20	72.48	94.00	63.67	76.67	39.39	56.67	83.60	81.79
	Dynamic Sampling	Untargeted	APPS + generated	67.22	70.62	45.35	43.48	73.80	95.80	56.67	76.67	39.39	56.67	84.85	82.52
	Dynamic Sampling	Error-targeted	APPS + generated	67.40	68.91	45.88	42.75	74.68	93.00	56.67	60.00	45.45	53.33	84.63	81.97
CASE	No Filtering	Error-targeted	APPS + generated	67.28	69.31	45.28	41.59	72.92	94.92	60.00	68.00	44.85	52.00	86.53	**82.62**
	Keep All Wrong	Error-targeted	APPS + generated	66.76	71.28	45.53	41.74	73.60	89.68	60.67	70.00	37.58	50.00	**87.53**	81.97
	Boundary Selection	Error-targeted	APPS + generated	**68.75**	**72.36**	**45.96**	**45.36**	**74.77**	88.76	**64.67**	72.00	**46.67**	56.67	87.30	82.52

* GRPO with error-targeted generation is reported as CASE with boundary selection filtering strategy.

† DAPO does not apply additional filtering, as dynamic sampling is already built into its training procedure.

Table 4 examines whether the gains arise from boundary selection, training-pool expansion, or error-targeted generation. We use GRPO as the basic on-policy training pipeline, and additionally compare with DAPO to distinguish the effect of targeted generation from stronger adaptive sampling. We observe three findings: 1) under GRPO, teacher-generated problems improve over boundary selection alone, showing the benefit of expanding the learnable task pool; 2) *error-targeted generation* generally improves over untargeted generation, indicating that the gain comes from error-aligned practice rather than simply adding more data; 3) our full pipeline remains stronger overall than DAPO, suggesting that targeted task generation complements adaptive sampling by expanding the training distribution beyond the existing problem pool.

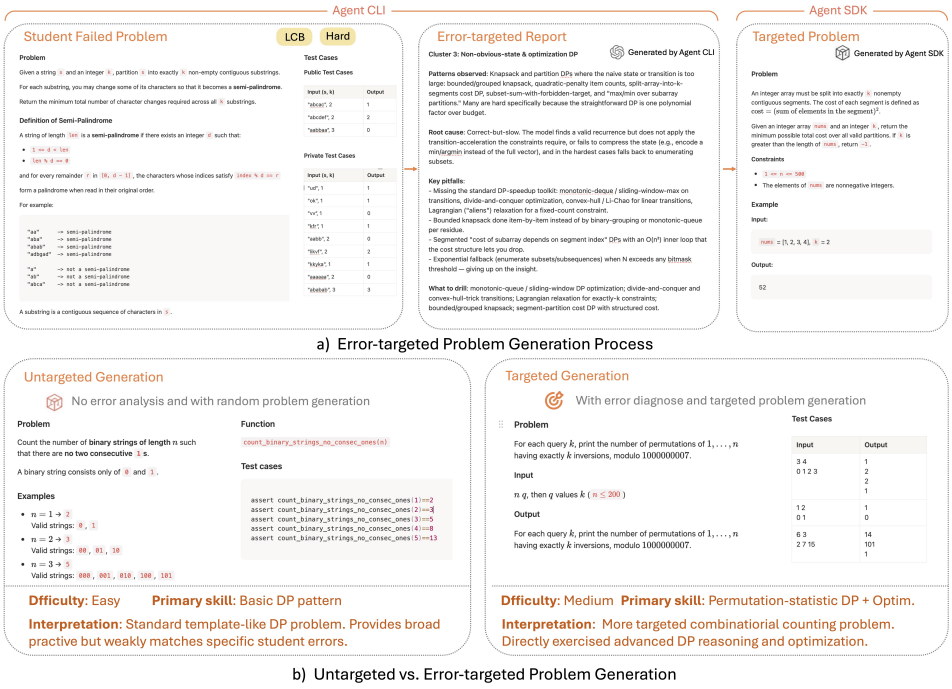


Figure 6: Case studies of problems generated by targeted and untargeted pipelines.

RQ3: How does teacher choice affect the effectiveness of our method? Figure 7 shows that the three teachers, `Claude Opus-5`, `GPT-5.6 Sol`, and `Kimi-K3`, all substantially improve the base student and, importantly, generalize beyond the training datasets to out-of-distribution (OOD) benchmarks spanning code, mathematics, and financial reasoning. The consistent gains across all three teachers indicate that our training-pool construction is robust to teacher choice rather than relying on a particular frontier model. At the same time, the magnitude of improvement varies across teachers and benchmarks, indicating that teacher choice affects how the student generalizes beyond the training distribution. This variation suggests that different teachers lead to different learned capability profiles, even though all three consistently improve the base student. Overall, the results show that our framework is robust across teacher models while still being sensitive to the characteristics of the selected teacher.

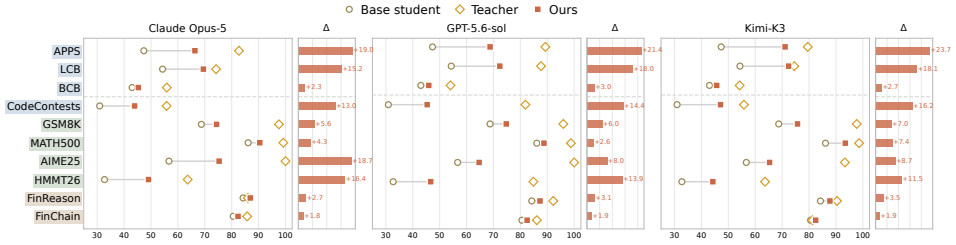


Figure 7: Cross-domain performances of base model, teachers, and trained students across code, math, and finance benchmarks. Each panel corresponds to a different teacher.

6 Conclusion

We present **CASE**, an adaptive on-policy post-training framework that combines competence-aware curriculum selection with error-targeted problem generation. Experiments show that both components provide complementary gains, outperforming generic generation and stronger adaptive sampling baselines while remaining effective across different teacher models and out-of-distribution benchmarks. These results suggest that effective on-policy post-training should adapt not only how the model learns, but also what problems it learns from.

AI USE STATEMENT

Large language models are an integral part of our experimental methodology. Frontier LLMs are accessed through their official APIs and used as teacher models for failure diagnosis and error-targeted task generation, and the student LLMs are deployed locally and optimized through our on-policy post-training pipeline. Additionally, generative AI tools were used only for writing and language polishing. All technical content were developed and verified by the authors.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or private data. All experiments are conducted using publicly available datasets and language models for research purposes. We make efforts to comply with the licenses and usage policies of the datasets, models, and APIs used in our experiments. While the proposed method may improve the reasoning capabilities of language models, such improvements could also be applied in unintended or harmful settings. We therefore encourage responsible use of the method and appropriate safeguards when deploying trained models in real-world applications.

REPRODUCIBILITY STATEMENT

We provide detailed implementation and experimental settings to facilitate reproducibility. The main training and evaluation setup is described in Section 5.1, with additional implementation details, hyperparameter configurations, and experimental settings provided in Appendix A. All comparisons use consistent training and evaluation protocols, and repeated experiments are conducted with 5 independent seeds. The source code, teacher-agent prompts, and configuration files are available in an anonymous repository `https://anonymous.4open.science/r/CASE_ICLR_3475`

REFERENCES

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*, volume 2024, pp. 21246–21263, 2024.

Anthropic. Introducing claude opus 5. `https://www.anthropic.com/news/claude-opus-5`, 2026. Accessed: Sept 2026.

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *Advances in Neural Information Processing Systems*, 38, 2026.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms. *arXiv preprint arXiv:2605.00674*, 2026.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pp. 1311–1320. Pmlr, 2017.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *International Conference on Learning Representations*, volume 2024, pp. 32694–32717, 2024.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary llms. *arXiv preprint arXiv:2305.15717*, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, et al. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 2026.

Hugging Face H4. Aime 2024. `https://huggingface.co/datasets/HuggingFaceH4/aime_2024`, 2024. Accessed: Sept 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Yuxin Jiang, Chunkit Chan, Mingyang Chen, and Wei Wang. Lion: Adversarial distillation of proprietary large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3134–3154, 2023.

Andrej Karpathy. Autoresearch. `https://github.com/karpathy/autoresearch`, 2026. Accessed: Sept 2026.

Jongwoo Ko, Sungnyun Kim, Tianyi Chen, and Se-Young Yun. Distillm: Towards streamlined distillation for large language models. *arXiv preprint arXiv:2402.03898*, 2024.

M Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. *Advances in neural information processing systems*, 23, 2010.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning. *arXiv preprint arXiv:1707.00183*, 2017.

Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. `https://ai.meta.com/blog/llama-4-multimodal-intelligence`, 2025. Accessed: Sept 2026.

OpenAI. Gpt-oss-120b & gpt-oss-20b model card. `https://openai.com/index/gpt-oss-model-card`, 2025. Accessed: Sept 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. `https://openai.com/index/gpt-5-6`, 2026. Accessed: Sept 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Erdi Sayar, Giovanni Iacca, Ozgur S Oguz, and Alois Knoll. Diffusion-based curriculum reinforcement learning. *Advances in Neural Information Processing Systems*, 37:97587–97617, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

Idan Shenfeld, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. Self-distillation enables continual learning. *arXiv preprint arXiv:2601.19897*, 2026.

Mingyang Song and Mao Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

Zichen Tang, E Haihong, Ziyan Ma, Haoyang He, Jiacheng Liu, Zhongjun Yang, Zihua Rong, Rongjin Li, Kun Ji, Qing Huang, et al. Financereasoning: Benchmarking financial numerical reasoning more credible, comprehensive and challenging. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15721–15749, 2025.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Nidhi Vakil and Hadi Amiri. Curriculum learning for graph neural networks: A multiview competence-based approach. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7036–7051, 2023.

Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):4555–4576, 2021.

Ziyi Wang, Yuxuan Lu, Yimeng Zhang, Qun Liu, Chen Luo, Jiri Gesi, Hanqing Lu, Yisi Sang, Manling Li, Jing Huang, et al. Sentinel: Failure-driven reinforcement learning for training tool-using language model agents. *arXiv preprint arXiv:2606.12908*, 2026.

Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. In *International conference on machine learning*, pp. 5238–5246. PMLR, 2018.

Zhuohan Xie, Daniil Orel, Rushil Thareja, Dhruv Sahnan, Hachem Madmoun, Fan Zhang, Debopriyo Banerjee, Georgi Nenkov Georgiev, Xueqing Peng, Lingfei Qian, et al. Finchain: A symbolic benchmark for verifiable chain-of-thought financial reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14529–14553, 2026.

Tianzhu Ye, Li Dong, Zewen Chi, Xun Wu, Shaohan Huang, and Furu Wei. Black-box on-policy distillation of large language models. *arXiv preprint arXiv:2511.10643*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Enci Zhang, Xingang Yan, Wei Lin, Tianxiang Zhang, and Lu Qianchun. Learning like humans: Advancing llm reasoning capabilities via adaptive difficulty curriculum learning and expert-guided self-reformulation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 6630–6644, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

Appendices

CONTENTS

A Experimental Details **15**

A.1 Sequence Length Analysis 15

A.2 Detailed Experiment Results 15

B Teacher Skill Structure and Generation Protocol **16**

B.1 Core Skill Specification 16

B.2 Problem-Generation Examples and Test-Type Guidance 17

B.3 Reference Specifications and Auxiliary Resources 18

B.4 Error Analysis and Diagnostic Artifacts 19

B.5 Generated Candidate Problems 22

B.5.1 Example 1: Counting-DP state design 22

B.5.2 Example 2: Implicit graph modeling 23

B.5.3 Example 3: Emerging geometry weakness 23

B.5.4 Example 4: BCB-style software engineering task 24

B.6 Verification of Generated Problems 24

C Representative Cases on OOD Test Benchmarks **26**

C.1 CodeContests 26

C.2 MATH500 28

C.3 FinReasoning 30

C.4 FinChain 31

A Experimental Details

A.1 Sequence Length Analysis

During training, the maximum prompt and response lengths are set to 4,096 and 8,192 tokens, respectively. Figure A.8 reports the mean prompt and response lengths across optimization steps for the full CASE pipeline with different teacher models. Prompt lengths remain relatively stable and substantially below the 4,096-token limit across all three teachers. Response lengths are longer but likewise remain below the 8,192-token training budget on average, with no systematic increase toward the limit over the course of optimization. These results suggest that the chosen sequence limits provide sufficient headroom for typical training trajectories. Other settings and implementation details are provided in repository `https://anonymous.4open.science/r/CASE_ICLR_3475`.

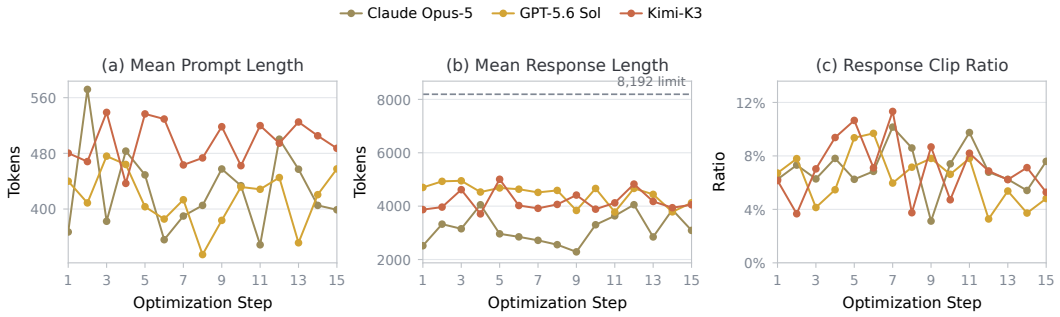


Figure A.8: Mean prompt and response lengths, together with the response clipping ratio, across optimization steps in the final training round of CASE with `Claude Opus-5`, `GPT-5.6 Sol`, and `Kimi-K3` as teachers.

A.2 Detailed Experiment Results

Table A.5: Robustness across five repeated training runs. Results are reported as mean Pass@1 ± standard deviation.

Benchmark	Base	CASE with `Claude Opus-5`			CASE with `GPT-5.6 Sol`			CASE with `Kimi-K3`		
	`Qwen-3.5-9B`	No Filtering	Keep All Wrong	Boundary Selection	No Filtering	Keep All Wrong	Boundary Selection	No Filtering	Keep All Wrong	Boundary Selection
Validation Benchmarks										
APPS	47.35 ± 0.68	60.12 ± 0.47	66.97 ± 0.37	66.33 ± 0.33	67.28 ± 0.57	66.76 ± 0.70	68.75 ± 0.73	64.59 ± 0.25	62.48 ± 0.25	71.10 ± 0.23
LCB	54.31 ± 1.89	63.37 ± 0.61	71.68 ± 0.60	69.48 ± 0.45	69.31 ± 0.62	71.28 ± 0.43	72.36 ± 0.62	69.19 ± 0.40	67.79 ± 1.09	72.40 ± 0.34
BCB	43.00 ± 1.36	44.56 ± 1.02	46.16 ± 0.45	45.32 ± 1.00	45.28 ± 0.70	45.53 ± 0.49	45.96 ± 0.55	45.12 ± 0.78	45.04 ± 0.31	45.67 ± 0.77
Out-of-Distribution Test Benchmarks										
CodeCont.	30.94 ± 2.93	41.29 ± 1.59	40.58 ± 1.43	43.91 ± 0.97	41.59 ± 1.34	41.74 ± 1.89	45.36 ± 1.24	43.45 ± 1.59	41.30 ± 0.35	47.10 ± 0.68
GSM8K	68.73 ± 0.43	71.04 ± 1.36	69.37 ± 1.28	74.37 ± 0.98	72.92 ± 0.88	73.60 ± 1.01	74.77 ± 1.31	76.09 ± 0.74	71.78 ± 0.97	75.72 ± 0.83
MATH500	86.12 ± 1.35	88.24 ± 2.15	88.04 ± 1.80	90.44 ± 1.45	94.92 ± 0.18	89.68 ± 1.49	88.76 ± 1.06	91.84 ± 1.32	90.64 ± 1.11	93.84 ± 0.78
AIME25	56.67 ± 4.71	66.00 ± 2.98	60.67 ± 4.94	75.33 ± 5.58	60.00 ± 8.50	60.67 ± 7.96	64.67 ± 5.06	62.00 ± 8.37	61.33 ± 1.83	65.33 ± 1.83
AIME24	63.33 ± 0.43	55.33 ± 2.79	67.33 ± 1.49	67.33 ± 2.79	68.00 ± 5.06	70.00 ± 4.08	72.00 ± 8.03	72.67 ± 1.49	72.00 ± 3.80	72.00 ± 6.91
HMMT26	32.73 ± 5.42	37.58 ± 3.46	40.61 ± 5.50	49.09 ± 3.95	44.85 ± 2.54	37.58 ± 4.07	46.67 ± 4.07	43.03 ± 5.42	40.61 ± 5.91	44.24 ± 5.50
HMMT25	46.67 ± 5.77	48.00 ± 5.58	46.67 ± 6.24	48.67 ± 2.98	52.00 ± 6.91	50.00 ± 7.82	56.67 ± 5.27	52.67 ± 7.23	51.33 ± 5.58	52.00 ± 4.47
FinReasoning	84.23 ± 0.43	86.56 ± 0.48	86.77 ± 0.36	86.92 ± 0.31	86.53 ± 0.28	87.53 ± 0.35	87.30 ± 0.25	86.99 ± 0.49	86.80 ± 0.22	87.75 ± 0.38
FinChain	80.58 ± 0.40	81.61 ± 0.58	81.33 ± 0.17	82.37 ± 0.37	82.62 ± 0.13	81.97 ± 0.05	82.52 ± 0.35	81.94 ± 0.18	81.67 ± 0.32	82.46 ± 0.22
Avg. Improvement (test)		+2.85	+3.49	+7.60	+5.94	+4.75	+7.64	+6.74	+5.27	+7.83
Avg. Improvement (validation + test)		+4.09	+5.96	+8.74	+7.55	+6.81	+9.26	+7.91	+6.51	+9.58
OOD Win Rate		8/9	8/9	**9/9**	8/9	**9/9**	**9/9**	**9/9**	**9/9**	**9/9**

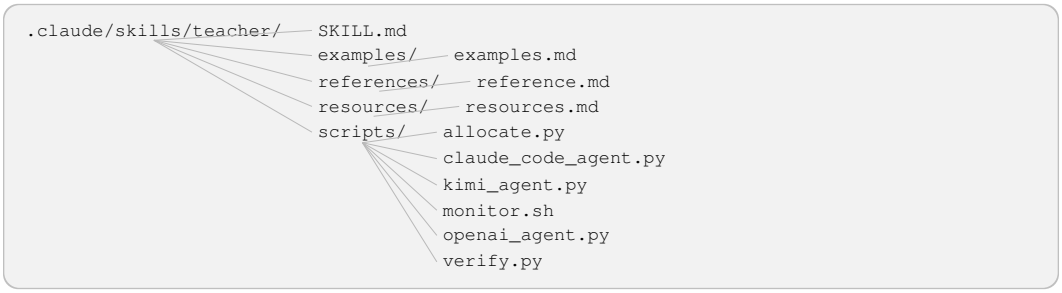
Results are reported as mean Pass@1 ± standard deviation over independent training runs.
Avg. Improvement denotes the macro-average Pass@1 improvement over Qwen-3.5-9B (base) across all twelve benchmarks.

We repeat each training configuration five times with independent random seeds, while keeping all model, optimization, and evaluation hyperparameters fixed. We report the mean and standard deviation of Pass@1 across the five runs. The results show that the gains of **CASE** are stable across repeated runs and consistent across different teacher models. Boundary Selection is the most reliable configuration, achieving the strongest overall improvement while also improving over the base model on all OOD benchmarks for every teacher. By contrast, *No Filtering* and *Keep All Wrong* also yield clear gains, but their improvements are less uniform across benchmarks. The low variance observed on most benchmarks further indicates that the improvements are not driven by isolated random runs. The larger variance on AIME and HMMT is mainly due to their small benchmark sizes, with only around 30 problems, so a few changed outcomes can noticeably affect results.

B Teacher Skill Structure and Generation Protocol

B.1 Core Skill Specification

The teacher model is instantiated with a structured skill package that specifies how training problems should be analyzed and generated. Rather than relying on a single prompt, the package separates the core generation instructions from examples, supporting references, auxiliary resources, and executable scripts. This modular organization allows the teacher to follow a consistent generation protocol while accessing additional guidance only when needed.



`SKILL.md` serves as the entry point of the teacher skill and defines the overall generation protocol, including the teacher's role, the expected workflow, output requirements, and constraints on problem construction. The remaining components provide additional information that can be consulted during different stages of this workflow.

The `examples/examples.md` file provides representative problem-generation examples for different test types. In particular, it demonstrates how the problem statement, entry point, input/output interface, and executable test cases should be organized under each evaluation format. These examples are used as protocol and formatting guidance for the teacher, rather than as evidence or source problems for error-targeted generation.

The `references/reference.md` file collects reusable reference information required by the generation procedure. Separating this material from the core skill keeps `SKILL.md` concise while allowing the teacher to consult detailed specifications when needed during problem construction.

The `resources/resources.md` file describes auxiliary resources available to the teacher during generation. These resources complement the generation instructions and references by providing additional support for carrying out the task, while remaining separate from the core generation protocol.

Finally, the `scripts/` directory operationalizes the skill package, including utilities for invoking different teacher models, allocating generation jobs, monitoring execution, and verifying generated outputs. These scripts implement the execution layer and do not themselves define the semantic content of the generation protocol.

Excerpt from `SKILL.md`: Skill Description

```
name: teacher
description: >-
   Teacher-guided adaptive curriculum loop for on-policy post-training, supporting Claude
   ↪  Opus-5, GPT-5.6 Sol, and Kimi-K3 as interchangeable teacher models. A persistent
   ↪  monitor tracks round artifacts and emits one of three events (NEED_ANALYSIS /
   ↪  NEED_PROBLEMS / NEED_LAUNCH), corresponding to analyzing student failures,
   ↪  generating and self-verifying targeted training problems, or launching the next
   ↪  training round. Each completed round produces new evaluation artifacts that
   ↪  determine the next stage of the loop, and the process continues until MAX_ROUND is
   ↪  reached.

   Trigger when the monitor fires an event, or when the user asks to
   ↪  start/continue/monitor the teacher loop.
---
```

The skill further separates the overall workflow into two roles with different responsibilities and access privileges. **Agent CLI** acts as the orchestrator: it monitors round states, analyzes student failures, and produces sanitized error reports. **Agent SDK** is the isolated problem-authoring module:

Function	Trigger	Role
Fn 1: Analyse	`NEED_ANALYSIS`	Cluster student failures by benchmark and convert them into sanitized error reports describing recurring capability gaps.
Fn 2: Author & Verify	`NEED_PROBLEMS`	Use the isolated authoring worker to generate new targeted problems from the sanitized reports and verify the resulting problems.
Fn 3: Launch	`NEED_LAUNCH`	Launch the next GRPO round, including training, evaluation, and persistence of the resulting round artifacts.

Table B.6: The three functions defined by the teacher skill.

it receives only these sanitized reports and the generation guidance, and never accesses the original failed evaluation problems. This separation establishes the *anti-leak* boundary between failure diagnosis and targeted problem generation.

Excerpt from `SKILL.md`: Module Separation

```
1. **Agent CLI (the orchestrator)** -- this skill, driven by the main CLI + the monitor.
↪  It analyses failures into sanitised error reports (Fn 1), spawns the SDK worker to
↪  author problems (Fn 2), and launches GRPO training (Fn 3). The launched round trains,
↪  evaluates, and writes `failures.json` / `results.tsv` / `best.json` itself (auto
↪  chain/revert), so there is **no separate record step**.

2. **Agent SDK (the problem author)**, Claude Code SDK by default, or `--backend openai` /
↪  `--backend kimi`. An **isolated** worker the agent spawns. It only ever receives the
↪  parameters the agent passes it -- the three **sanitised** error-report paths, the
↪  examples-template path, and the per-benchmark count -- authors + self-verifies
↪  problems in its own context, and writes `claude_problems.json`. **It must never see
↪  `failures.json`** (which holds the held-out validation problems): turning failures
↪  into category-level reports is the agent's job, so the worker only ever drills
↪  sanitised categories. This is the anti-leak boundary.
```

Three teacher functions. Based on this separation of responsibilities, the teacher workflow is organized into three functions: failure analysis, targeted problem authoring and verification, and launch of the next training round. Table B.6 summarizes the files and resources associated with each function.

State transition. These functions are not invoked independently. Instead, a persistent monitor determines which function should execute next according to the artifacts already produced in the current round. Each completed function therefore changes the observable state of the workflow and triggers the next stage.

One Full Teacher-guided Round (one loop)

```
LAUNCH a round (Fn 3)
  -> RUN run_round_mix.sh trains + evaluates + writes round_XX/{failures.json, a
     results.tsv row, best.json}, and auto chain/reverts the init model
  -> ANALYSE round_XX/failures.json -> {apps,lcb,bcb}_errors.md           (Fn 1)
  -> AUTHOR via the SDK worker      -> round_{XX+1}/claude_problems.json  (Fn 2)
  -> LOOP for round_{XX+1}, until MAX_ROUND rounds have evaluated
```

Together, these instructions define the semantic generation protocol: the teacher first identifies the capability targeted by the diagnostic artifact, constructs a new problem exercising that capability, and returns it in a format compatible with the downstream execution and verification pipeline.

B.2 Problem-Generation Examples and Test-Type Guidance

The `examples.md` file defines the problem-authoring protocol used by the teacher, specifying both a shared output schema and benchmark-specific construction rules. While all generated problems follow a common set of fields, their prompt style and test specification are adapted to the native

evaluation interface of the target benchmark. We therefore distinguish two problem families below: algorithmic problems for APPS and LCB, and library-oriented problems for BCB.

Common problem schema. Regardless of test type, every authored problem contains the common fields `test_type`, `entry_point`, `problem`, `difficulty`, `question_design_reasoning`, and `solutions`. The test-specific field is then determined by the evaluation interface.

- **Algorithmic problems (for APPS and LCB).** These benchmarks follow competitive- or interview-style problem formulations. APPS uses stdin-based evaluation, where the generated solution is a complete program that reads from standard input and prints the answer. LCB supports both stdin and functional formats; the latter evaluates a single top-level function with a fixed entry point.
  - **stdin:** each problem provides `inputs` as a list of raw input strings. The expected outputs are automatically obtained by executing the reference solution.
  - **functional:** provides `inputs` as argument lists rather than raw stdin strings. The verifier calls the reference function on each argument list to obtain the expected outputs.
- **Library-oriented software engineering problems (for BCB).** BCB follows a more rigorous docstring-style specification, including explicit arguments, return behavior, imports, and a fixed function signature. It uses the `unittest` format with `task_func` as the entry point.
  - **unittest:** provides a complete `unittest.TestCase` source in the `test` field. The test suite may check output values, return types, error behavior, numerical tolerance, determinism, side effects, and edge cases.

Problem design guidelines. Beyond matching the native format of each benchmark, generated problems are required to target concrete failure modes while remaining diverse in both structure and solution strategy. In particular, the teacher avoids template-like variant cloning, duplicate problems, trivial solutions that reduce to direct library calls, ambiguous specifications, and tasks with multiple entry points. Problems are constructed from category-level diagnoses rather than individual held-out prompts, ensuring that targeted generation addresses recurring capability gaps without paraphrasing evaluation examples.

Test construction guidelines. The test design is adapted to the corresponding evaluation interface. For `stdin` problems, tests include representative edge cases and require deterministic outputs. For `functional` problems, several diverse function calls are used with stable output ordering. For `unittest` problems, the test suite additionally checks both types and values, covers empty and degenerate inputs, and explicitly controls randomness through seeding or mocking when needed. These constraints help ensure that generated problems are not only syntactically valid, but also sufficiently specified and reliably executable by the downstream grader.

B.3 Reference Specifications and Auxiliary Resources

Beyond the core skill instructions and problem-authoring examples, the teacher is provided with reference specifications that define the artifacts exchanged across training rounds. These references describe the structure and role of the teacher's inputs, intermediate diagnostic reports, generated outputs, and round-level evaluation records. They therefore serve as an interface contract between failure analysis, problem generation, verification, and the surrounding training pipeline.

Round-level artifact specification. The reference file first specifies the artifacts maintained for each round. The evaluated student produces `failures.json`, which serves as the diagnostic input to Agent CLI. Failure analysis then produces three benchmark-specific reports, `apps_errors.md`, `lcb_errors.md`, and `bcb_errors.md`. These sanitized intermediate artifacts are subsequently consumed by Agent SDK to construct the candidate problems for the next round. In parallel, round-level evaluation statistics are persisted in `results.tsv` and associated checkpoint metadata.

Diagnostic report specification. The reference specification also defines the structure of the benchmark-specific error reports produced by Agent CLI. Each report groups repeated failures into capability-level clusters and records four types of diagnostic information: the recurring pattern, its

likely root cause, concrete failure modes, and the capabilities that should be targeted by subsequent practice. An overall assessment summarizes the dominant weaknesses observed in the round.

A key part of this specification is the sanitisation rule. Diagnostic reports must describe algorithmic or reasoning categories rather than copying the domain, wording, or narrative of individual held-out problems. The resulting report should therefore remain meaningful as a general diagnosis of a class of failures while being non-traceable to a specific evaluation item. This constraint is what allows the report to be passed to the isolated Agent SDK without exposing the original failed problems.

Round-state and result references. The reference file further documents the round-level evaluation records used by the orchestration logic. In particular, `results.tsv` records benchmark performance, model initialization, and whether a round is retained as the current best checkpoint or discarded in favor of a previous round. These records are written automatically by the evaluation pipeline and are consulted by the teacher loop rather than manually modified by the teacher.

Auxiliary execution resources. Complementing these reference specifications, `resources.md` provides the execution-side information required to connect the teacher workflow to the training pipeline, such as the relevant training entry points, runtime configuration, and round-launch procedure. We separate these operational resources from the semantic authoring instructions so that the latter remain independent of implementation details.

B.4 Error Analysis and Diagnostic Artifacts

Having introduced the diagnostic schema and sanitisation requirements, we now show a concrete error-analysis artifact produced during training. The following report was generated by `GPT-5.6 Sol` for the APPS failures in Round 03, illustrating how individual failures are abstracted into category-level patterns, root causes, and targeted practice directions without exposing the original evaluation prompts.

Error Analysis Report Generated by GPT-5.6 Sol — Round 03

```
# APPS Error Analysis -- Round 03

**Failures: 337 / 1000 evaluated**

---

## Cluster 1: Counting DP & Modular Combinatorics -- State-Design Failures

**Patterns observed**: The single largest and most concentrated bucket (~30% of
↪  failures). Tasks ask for the number of objects satisfying a structural constraint,
↪  answered modulo a prime: tilings and covering recurrences, permutations counted by a
↪  statistic (inversions, run-length, displacement/"oddness", square-of-permutation
↪  preimages), bracket/trie enumeration, knapsack-style "number of plans" counts,
↪  inclusion-exclusion over subsets, and expected values that must be returned as a
↪  modular rational P·Q^{--1}. The common shape is a recurrence whose transition
↪  compresses an exponential object space into a small carried invariant.

**Root cause**: Conceptual -- failure to design the DP *state*, not to code a known one.
↪  The model reaches for the obvious index (position/prefix) but omits the auxiliary
↪  dimension that makes transitions Markovian (remaining "debt", parity/carry,
↪  count-of-open-groups, last-boundary color, rank within equivalence class). It also
↪  conflates *counting distinct outcomes* with *summing a quantity*, and treats
↪  expectation as a float instead of arithmetic in the modular field.

**Key pitfalls**:
- Missing a second state dimension, so overlapping subproblems are recounted or
↪  under-counted.
- Applying modulo to numerators/denominators without modular inverse; dividing under a
↪  modulus.
- Off-by-one on empty-object base cases (empty string/sequence/subset counted twice or
↪  zero times).
- No overflow control before the mod, or taking mod too late so intermediate products
↪  wrap.
- Double-counting symmetric configurations that inclusion-exclusion was supposed to
↪  cancel.
```

```
**What to drill**: linear tiling/recurrence counts; digit DP (count integers with a
↪  bit/sum property under a bound); permutation-statistic DP (inversions, runs,
↪  displacement); interval/segment repaint DP; subset-sum "number of ways" with a mod;
↪  expected-value-as-modular-rational and Grundy-free probability DP.

---

## Cluster 2: Latent-Graph Modeling & Traversal -- Recognizing the Implicit Graph

**Patterns observed**: The second stubborn core (~25% of failures). No adjacency list is
↪  given; the graph must be *inferred* from a relation -- same-value or
↪  arithmetic-successor edges, a doubling/modular successor map, a lowest-prime-divisor
↪  parent structure, a time/state-expanded layer graph, or an
↪  edge-orientation/reachability construction. Once built, the task is a standard
↪  traversal, SCC, Eulerian/Hamiltonian existence, bridge/2-edge-connectivity, matching,
↪  tree-DP-on-inferred-tree, or shortest/widest path.

**Root cause**: Reasoning -- the model does not perform the modeling leap that turns
↪  entities+relations into vertices+edges. It attempts direct simulation or ad-hoc case
↪  analysis on the surface objects instead of recognizing an isomorphic graph problem,
↪  so it never invokes the right algorithm.

**Key pitfalls**:
- Building the wrong node set: using raw values as nodes when the state is (position,
↪  phase) or (node, residue).
- Missing implicit edges (e.g. all equal-valued items mutually adjacent) → wrong
↪  connectivity/shortest path.
- Confusing directed vs. undirected, and orientation choices that must differ per
↪  objective (max vs. min reachability).
- Eulerian/Hamiltonian existence handled by greedy walk instead of degree-parity /
↪  construction criteria.
- Tree-DP recursion depth on large inferred trees; no iterative/explicit-stack fallback.

**What to drill**: BFS/DFS on a graph defined by a successor function or same-key edges;
↪  time-expanded/layered BFS (state = node × phase); edge-orientation for
↪  SCC/reachability; bridge and 2-edge-connectivity construction; Eulerian-path
↪  existence on a doubling map; rooted-tree DP where the tree is reconstructed from
↪  distance/parent constraints.

---

## Cluster 3: Number Theory & Modular/Digit Arithmetic

**Patterns observed**: gcd-driven recursions, divisibility and simultaneous-remainder
↪  (CRT/LCM) constructions, representations in mixed or non-decimal bases,
↪  sum/product-of-digits functions, coprime-k-th-element queries, factoring into
↪  distinct parts, and "smallest number with property X". Answers hinge on an arithmetic
↪  insight (a closed form or a fast recurrence), not on data structures.

**Root cause**: Conceptual/Reasoning -- the model uses brute enumeration where a
↪  divisor-structure or digit-position argument is required, and misjudges magnitudes
↪  (values up to 10^{18} / 10^6 digits) so a per-unit loop is impossible.

**Key pitfalls**:
- Iterating to N when only $O(\sqrt{N})$ divisors or $O(log N)$ digit positions matter →
↪  timeout.
- Forgetting that "distinct remainders 1..k" forces a factorial/LCM condition;
↪  brute-checking instead.
- Big-integer handling for arbitrary-length inputs done with fixed-width ints.
- Greedy digit/base decomposition without proving the greedy choice; corner value 0/1 and
↪  single-digit cases.

**What to drill**: gcd/step-count recurrences with fast-forward; CRT and LCM-condition
↪  construction; base-B / distinct-power-of-b representations; digit-sum and
↪  digit-product inverse problems; $O(\sqrt{N})$ factorization and divisor sieves.

---

## Cluster 4: String Reconstruction, Parsing & Suffix Structures

**Patterns observed**: Restore a hidden string/sequence from indirect evidence (sorted
↪  permutations, per-index distance sums, overlapping triples, swap witnesses,
↪  question-mark templates); palindrome partitioning / equal-length palindrome cutting;
↪  prefix=suffix occurrence counting (KMP/Z); prefix-suffix merge/compression; grammar
↪  parsing and tiny interpreters. Many are constructive ("output any valid").

**Root cause**: Implementation/Reasoning -- the model knows the primitives (KMP, Z,
↪  greedy folding) but mis-specifies the reconstruction invariant, or handles the
↪  constructive existence check incorrectly (declares impossible when a witness exists,
↪  or emits an invalid witness).
```

```
**Key pitfalls**:
- Failing feasibility/parity checks before construction (character-count parity for
↪  palindromes).
- Greedy assignment of ambiguous positions without backtracking, producing locally-valid
↪  but globally-invalid strings.
- Prefix-function/Z edge cases at length-1 and full-length overlaps.
- Lexicographically-minimal-answer requirement ignored (any-valid produced instead of
↪  smallest).

**What to drill**: KMP/Z prefix-function occurrence counting; palindrome-partition
↪  existence + reconstruction; restore-permutation-from-k-tuples; template/question-mark
↪  completion with feasibility proof; recursive-descent parsing of small grammars.

---

## Cluster 5: Greedy + Exchange-Argument Scheduling & Binary-Search-on-Answer

**Patterns observed**: Order/select to optimize a cost (assign-to-minimize,
↪  sort-by-ratio, sliding-window removals), interleaved with "minimize the maximum /
↪  maximize the minimum" tasks solved by binary search on the answer plus a feasibility
↪  check. Includes shuttle/transport timing, coverage-with-minimum-cost, and pick-order
↪  optimizations.

**Root cause**: Reasoning -- the model picks a plausible-but-unproven greedy key, or
↪  fails to recognize a monotone predicate that licenses binary search, defaulting to
↪  exponential search or a wrong sort criterion.

**Key pitfalls**:
- Sorting by the wrong comparator (absolute value vs. ratio vs. deadline); no
↪  exchange-argument check.
- Binary search bounds/precision wrong; feasibility predicate not actually monotone as
↪  assumed.
- Two-pointer/sliding-window index management off-by-one on shrink/grow.
- Tie-breaking and "cannot be done" branches unhandled.

**What to drill**: exchange-argument sort proofs; parametric "min max / max min" via
↪  binary search + greedy feasibility; sliding-window minimal-removal; interval covering
↪  with fewest increments.

---

## Cluster 6: Computational Geometry & Continuous Simulation -- Emerging Residual Family

**Patterns observed**: A now-prominent block (~12% of failures) that stands out as APPS
↪  overall improved: ray reflections in mirrors/tubes, circle-intersection area, 3D
↪  bounding-box / minimal-enclosing-cylinder, rotating-motion and physics stepping,
↪  agglomeration/collision of moving bodies, polygon covering, and Euclidean-radius
↪  reachability. These are numeric, formula-heavy, tolerance-checked.

**Root cause**: Conceptual/Edge-case -- weakest area on *derivation* of the closed-form
↪  geometry and on floating-point discipline. The model either cannot set up the
↪  analytic relation (reflection unfolding, law-of-cosines segment area, projection onto
↪  an arbitrary axis) or mishandles precision and degenerate configurations.

**Key pitfalls**:
- Reflection problems not solved by coordinate unfolding; instead simulated step-by-step
↪  and diverging.
- Circle/segment area via wrong branch (containment vs. disjoint vs. partial overlap not
↪  separated).
- Floating comparisons without epsilon; answers just outside 1e-6 tolerance.
- Degeneracies: collinear points, zero-radius, tangency, division by near-zero slopes.
- Continuous-time event simulation ordered incorrectly (next-collision time not taken as a
↪  min-heap of events).

**What to drill**: reflection-by-unfolding; two-circle intersection-area case split;
↪  projection/dot-cross primitives and orientation tests; event-driven continuous
↪  simulation with a time-priority queue; epsilon-robust comparisons and degenerate-case
↪  checklists.

---

## Cluster 7: Adversarial Game Theory & Sprague-Grundy

**Patterns observed**: Two-player optimal-play outcomes: take-away/arithmetic-move games,
↪  prime-multiple and powers games, Nim-like decompositions on graphs/heaps,
↪  tournament-bribing and elimination brackets, and minimax over reachable states.
↪  Answer is a winner label, an optimal value, or a guaranteed count.
```

```
**Root cause**: Conceptual -- the model does not reduce to win/lose (Grundy) states or a
↪  minimax recurrence with memoization; it substitutes heuristic or single-ply reasoning
↪  and misses parity/nimber structure.

**Key pitfalls**:
- No memoized game-state DP; recomputation blows up or gives inconsistent outcomes.
- XOR-of-Grundy-values not recognized for independent-subgame sums.
- Misidentifying terminal/losing positions and who-moves parity.
- Optimal *value* games conflated with win/lose games (needs alpha-beta / DP over values,
↪  not just a boolean).

**What to drill**: Grundy-number computation and Nim-sum over independent games; win/lose
↪  backward induction with memoization; minimax-value DP on small state spaces;
↪  parity-argument shortcuts for take-away games.

---

## Overall Assessment
The two long-standing stubborn cores remain the residual center of gravity: **counting-DP
↪  state design (Cluster 1)** and **latent-graph modeling (Cluster 2)** together account
↪  for roughly half of all failures and are the highest-leverage drill targets -- the
↪  deficit is the *modeling/state-invention* step, not coding known algorithms. Failures
↪  are concentrated, not spread: the top three clusters (counting-DP, latent-graph,
↪  number-theory) dominate. The clearest **new emerging family surfacing now that APPS
↪  improved is computational geometry / continuous physical simulation (Cluster 6)** --
↪  analytic setup and floating-point/degeneracy discipline are the weakest non-core
↪  skills and were previously masked by the larger algorithmic-reasoning gaps.
↪  Prioritize state-design and graph-modeling drills first, then add a dedicated
↪  geometry-with-tolerance track.
```

This example illustrates how the diagnostic artifact converts a large set of individual failures into a compact curriculum signal. Rather than treating all 337 failures uniformly, the report identifies a small number of recurring capability gaps and distinguishes their underlying causes. In particular, counting-DP state design and latent-graph modeling emerge as the two dominant weaknesses, together accounting for roughly half of the observed failures. The report further identifies computational geometry and continuous simulation as an emerging residual weakness, suggesting that the failure distribution itself evolves as previously dominant deficiencies are reduced. These diagnoses are then translated into concrete **what to drill** directions, providing Agent SDK with targeted generation objectives rather than access to the original evaluation problems.

B.5 Generated Candidate Problems

Following error analysis, Agent SDK converts the sanitized diagnostic reports into candidate training problems for the subsequent round. To make this diagnosis-to-generation process concrete, we show three candidate problems generated by `GPT-5.6 Sol` from the Round 03 APPS error report. The examples correspond to two dominant persistent failure clusters and one emerging residual weakness identified in the preceding analysis.

For stdin-style candidates, Agent SDK generates a set of test inputs together with a reference solution. Expected outputs are obtained automatically by executing the reference solution, rather than being manually authored by the teacher. We therefore show the generated test inputs together with each candidate below.

B.5.1 Example 1: Counting-DP state design

Corresponding diagnosis. Cluster 1 identifies a persistent state-design weakness in counting DP: the student often fails to introduce the auxiliary state required to capture the relevant combinatorial statistic. Permutation-statistic DP is explicitly identified as a high-value practice direction.

Generated problem. Count the permutations of $1, \ldots, n$ having exactly k inversions, modulo $10^9 + 7$.
Input. Two integers n and k, where $n \leq 200$ and $k \leq 20000$.
Output. The number of such permutations modulo $10^9 + 7$.

Generated test inputs.

```
3 1
4 0
5 10
```

```
6 7
```

Difficulty. Hard

Why this problem. Solving the problem requires an inversion-count state and an efficient sliding-window transition, directly exercising the state-design step highlighted in the diagnosis rather than simple implementation of a fixed recurrence.

B.5.2 Example 2: Implicit graph modeling

Corresponding diagnosis. Cluster 2 shows that the student frequently misses the modeling step that converts entities and relations into vertices and edges, instead attempting direct simulation or case analysis on the surface problem.

Generated problem. Consider a directed graph whose vertices are the residues $0, \ldots, M-1$. From every vertex x, there are directed edges to

$$2x \bmod M \qquad \text{and} \qquad (x+1) \bmod M.$$

Given M, A, and B, compute the shortest distance from A to B.

Generated test inputs.

```
10 3 7
1 0 0
17 0 16
```

Difficulty. Medium

Why this problem. The arithmetic transition rules must first be recognized as an implicit directed graph before standard shortest-path traversal can be applied, directly targeting the missing modeling step identified in the diagnosis.

B.5.3 Example 3: Emerging geometry weakness

Corresponding diagnosis. Cluster 6 identifies computational geometry as an emerging residual weakness, with errors concentrated in analytic case derivation, floating-point handling, and degenerate configurations. Circle intersection is specifically highlighted as one of the problematic geometric structures.

Generated problem. Given two circles with centers (x_1, y_1) and (x_2, y_2) and radii r_1 and r_2, compute their intersection area and print the result to ten decimal places. Radii are nonnegative.

Generated test inputs.

```
0 0 1 3 0 1
0 0 2 0 0 1
0 0 1 1 0 1
0 0 0 0 0 0
```

Difficulty. Hard

Why this problem. The solution must explicitly distinguish disjoint, containment, and partial-overlap cases while handling floating-point clamping and degenerate configurations, directly targeting the geometry-specific reasoning and numerical robustness issues identified in the diagnosis.

Taken together, these examples illustrate the mapping from diagnostic signals to targeted practice. Agent CLI specifies the capability gap and the failure mechanism at the category level, while Agent SDK independently instantiates that diagnosis as a new problem with a different surface form and executable specification. The resulting candidate set therefore preserves the training signal identified from student failures without exposing or paraphrasing the original evaluation instances.

B.5.4 EXAMPLE 4: BCB-STYLE SOFTWARE ENGINEERING TASK

The preceding examples use the **stdin** interface native to APPS. To illustrate that the authoring protocol also adapts the executable test specification to the target benchmark, we additionally show one BCB candidate generated in the **unittest** format.

Corresponding diagnosis. In contrast to APPS-style algorithmic failures, BCB problems emphasize precise software contracts, including return types, output schemas, deterministic ordering, and explicit error behavior.

Generated problem. Given a list of item names and their corresponding revenues, return a Pandas DataFrame with columns `Item` and `Total Revenue`. Rows with the same item should be aggregated by summing their revenues, and the resulting DataFrame should be sorted by `Item` in ascending order. Unequal input lengths should raise a `ValueError`.

Test type. `unittest`

Entry point. `task_func`

Generated test cases.

```
class TestCases(unittest.TestCase):

    def test_type(self):
        self.assertIsInstance(task_func([], []), pd.DataFrame)

    def test_columns(self):
        self.assertEqual(
            list(task_func([], []).columns),
            ['Item', 'Total Revenue']
        )

    def test_values(self):
        self.assertEqual(
            task_func(
                ['b', 'a', 'b'],
                [2, 3, 4]
            ).to_dict('records'),
            [
                {'Item': 'a', 'Total Revenue': 3},
                {'Item': 'b', 'Total Revenue': 6}
            ]
        )

    def test_index(self):
        self.assertEqual(
            list(task_func(['b', 'a'], [1, 2]).index),
            [0, 1]
        )

    def test_error(self):
        with self.assertRaises(ValueError):
            task_func(['a'], [])
```

Difficulty. Easy

Why this problem. Unlike the preceding `stdin`-based algorithmic candidates, this problem is evaluated through an explicit behavioral contract. The test suite jointly checks return type, DataFrame schema, numerical correctness, deterministic ordering, index normalization, and error handling, reflecting the library-oriented software-engineering skills targeted by BCB.

B.6 VERIFICATION OF GENERATED PROBLEMS

The candidate problems described above are not inserted directly into the next training round. Instead, every candidate undergoes a mandatory self-verification stage that checks whether its reference solution, generated tests, and executable grading specification are mutually consistent. Importantly, verification invokes the same grader implementation used during training, so candidates are tested under the same evaluation interface that will later determine their training reward. The purpose

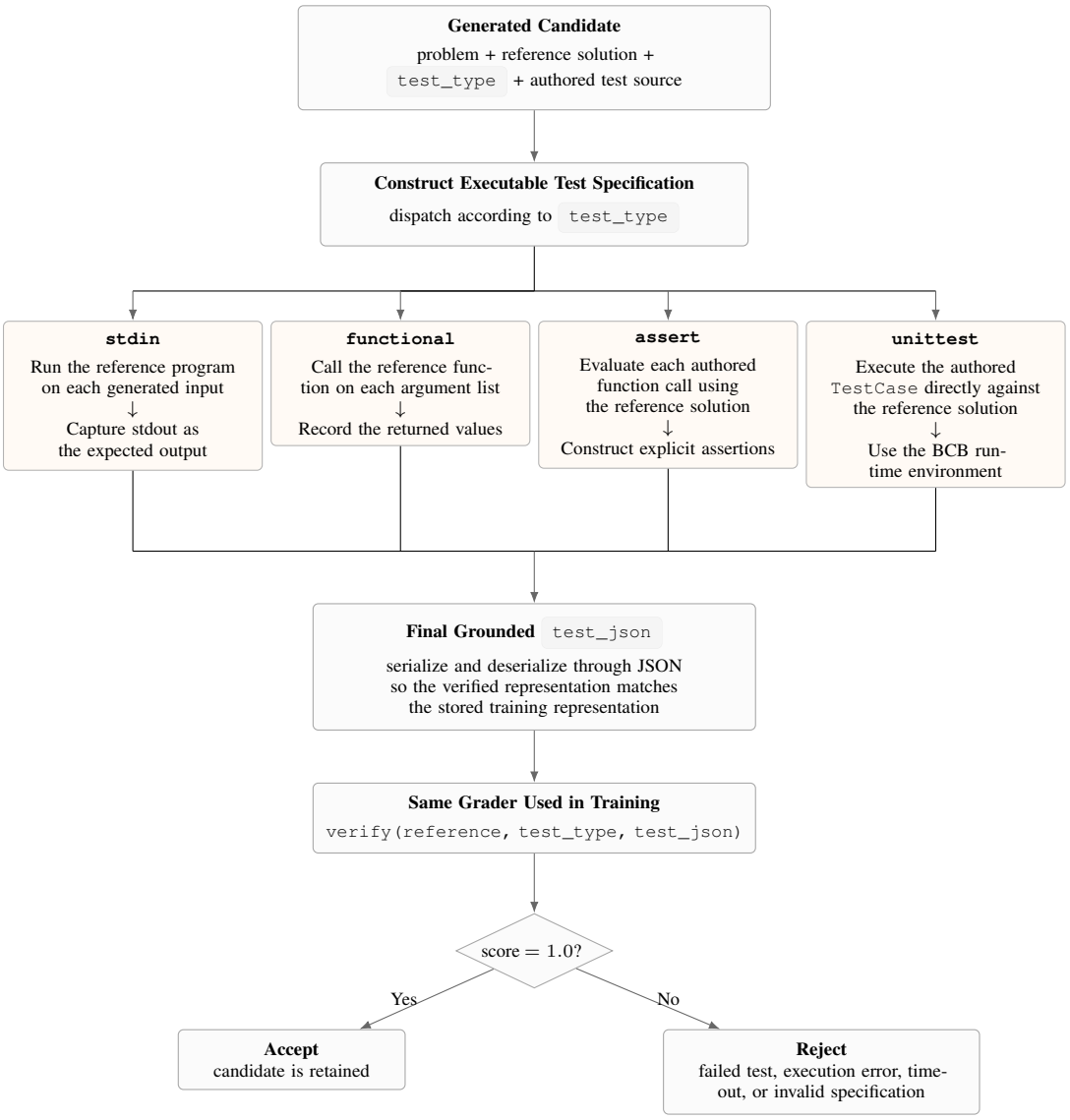


Figure B.9: Self-verification of generated candidate problems.

of this stage is therefore to filter out candidates that are non-executable, internally inconsistent, or incompatible with the downstream grader.

Self-verification procedure. Each generated candidate contains a problem statement, a reference solution, a test type, and the corresponding test source. Verification proceeds in **two main stages**. First, the verifier constructs an executable test specification from the authored tests and the reference solution. For test types requiring explicit expected outputs, these outputs are grounded by actually executing the reference solution rather than being independently authored by the teacher. Second, the resulting specification is passed together with the reference solution to the training grader. A candidate is retained only when the reference solution receives a score of exactly 1.0. Any failed test, execution error, timeout, or malformed specification causes the candidate to be rejected.

Test grounding across evaluation interfaces. Figure B.9 shows that the grounding procedure depends on the evaluation interface. For `stdin` problems, each generated input string is executed against the reference program and its stdout becomes the expected output. For `functional` problems, the reference function is invoked on each generated argument list and its return value is

recorded as ground truth. For `assert` problems, the authored function calls are evaluated under the reference implementation and converted into explicit assertions. For `unittest` problems, the test suite already specifies the expected behavior, so the complete `unittest.TestCase` is executed directly against the reference implementation without a separate output-grounding step. If a fully constructed `test_json` is already provided, the verifier uses that specification directly rather than reconstructing it from the convenience fields.

For BCB-style `unittest` candidates, verification is additionally performed with the same dedicated Python environment used by BCB training and evaluation. This is important for library-dependent tasks involving packages such as Pandas, Matplotlib, NumPy, or SciPy, whose behavior may depend on the execution environment.

Consistency with downstream grading. After grounding, the complete test specification is serialized and deserialized through JSON before it is graded. Consequently, the object passed to the verifier has the same serialized structure as the `test_json` subsequently stored and consumed by the training pipeline. Non-serializable reference outputs are rejected at this stage, preventing candidates from passing verification under an in-memory representation that cannot later be reproduced during training.

The grounded specification and reference solution are then evaluated by the same `grade` function used by the training grader. Verification therefore checks the complete executable path from the generated tests, through their stored representation, to the downstream grading interface, rather than relying on a separate lightweight checker.

Acceptance and post-processing. Only candidates whose reference solution obtains a perfect grader score of 1.0 are retained. Exceptions raised during grounding or execution, timeouts, malformed test specifications, and any score below 1.0 result in rejection. After this verification step, the surviving candidates are de-duplicated by their normalized problem statements to remove repeated generations across authoring batches. The remaining problems, together with their grounded `test_json`, reference solutions, difficulty labels, and generation rationales, form the verified problem set used by the subsequent training round.

C Representative Cases on OOD Test Benchmarks

We examine representative examples from the out-of-distribution test benchmarks to characterize both the capabilities acquired by the trained student and its remaining limitations. For each benchmark, we summarize the task format, evaluation protocol, and aggregate performance, then analyze one or two diagnostically informative cases, including both representative failures and, when appropriate, successfully solved challenging problems. These cases highlight transferable capabilities as well as recurring weaknesses in reasoning, implementation, and self-verification.

C.1 CodeContests

CodeContests (Li et al., 2022) evaluates competitive-programming ability through problems that require algorithm design, complexity-aware reasoning, and executable implementation. Each instance provides a **natural-language problem statement** together with a standard `input/output` specification and a set of executable tests. The model is required to generate a complete program that reads from standard input and writes the required answer to standard output. A problem is counted as solved only when the generated program passes the associated execution tests.

Table C.7 further breaks down the performance of the three CASE students by Codeforces problem index and problem rating. Across all three teachers, performance generally decreases as problem difficulty increases, with substantially higher Pass@1 on lower-rated problems and considerably lower accuracy in the highest-rating group. The rating-based breakdown exhibits a clearer monotonic trend than the contest-specific problem index, whose higher-index categories contain relatively few instances and are not globally calibrated across contests. Despite differences in absolute performance, the three teacher configurations show broadly similar difficulty-dependent behavior, suggesting that the remaining errors are concentrated on more demanding competitive-programming problems.

Table C.7: CodeContests performance across Codeforces problem indices and rating ranges.

Grouping	Range	# Tasks	**CASE** with Claude Opus-5		**CASE** with GPT-5.6 Sol		**CASE** with Kimi-K3	
			# Solved	Pass@1	# Solved	Pass@1	# Solved	Pass@1
CF Index[†]	A	39	32	82.05	34	87.18	33	84.62
	B	41	27	65.85	27	65.85	27	65.85
	C	42	21	50.00	22	52.38	21	50.00
	D	45	15	33.33	12	26.67	16	35.56
	E	41	13	31.71	11	26.83	15	36.59
	F	34	5	14.71	6	17.65	5	14.71
	G	16	4	25.00	4	25.00	5	31.25
	H+	18	6	33.33	10	55.56	8	44.44
CF Rating[‡]	≤ 1200	76	61	80.26	60	78.95	60	78.95
	$1200 \sim 1600$	38	22	57.89	22	57.89	24	63.16
	$1601 \sim 2000$	47	19	40.43	18	38.30	19	40.43
	$2001 \sim 2400$	47	12	25.53	17	36.17	14	29.79
	≥ 2400	68	9	13.24	9	13.24	13	19.12
Total		276	123	44.57	126	45.65	**130**	**47.10**

For each teacher, we report one randomly selected run from the repeated training experiments. Shaded cells indicate the best Pass@1 within each difficulty group; ties are all highlighted.

[†]**CF Index** denotes the Codeforces problem index (`cf_index`) within the original contest (A, B, C, ...). Later indices generally correspond to harder problems, but the scale is contest-specific and is not globally calibrated. **H+** aggregates all problems with index H or later because these categories contain only a small number of instances.

[‡]**CF Rating** denotes the Codeforces problem rating (`cf_rating`), a numerical estimate of problem difficulty that is more comparable across contests. Larger values indicate more difficult problems.

The following case is drawn from the student trained with CASE using `GPT-5.6 Sol` as the teacher. We select *Arena* as a representative failure because the model develops several correct local observations but fails to derive the general combinatorial structure. Instead, it replaces the unresolved derivation with a closed-form expression inferred from the provided examples, revealing a failure of generalization and self-verification.

Case 1: Arena

Problem. There are n heroes fighting in the arena. Initially, the i-th hero has a_i health points. The fight in the arena takes place in several rounds. At the beginning of each round, each alive hero deals 1 damage to all other heroes. Hits of all heroes occur simultaneously. Heroes whose health is less than 1 at the end of the round are considered killed.
If exactly 1 hero remains alive after a certain round, then he is declared the winner. Otherwise, there is no winner.
Your task is to calculate the number of ways to choose the initial health points for each hero a_i, where $1 \leq a_i \leq x$, so that there is no winner of the fight. The number of ways can be very large, so print it modulo 998244353. Two ways are considered different if at least one hero has a different amount of health. For example, $[1, 2, 1]$ and $[2, 1, 1]$ are different.
Input. The only line contains two integers n and x ($2 \leq n \leq 500$; $1 \leq x \leq 500$).
Output. Print one integer. The number of ways to choose the initial health points for each hero a_i, where $1 \leq a_i \leq x$, so that there is no winner of the fight, taken modulo 998244353.

Student response (selected excerpt).
The model first reasons correctly about several special cases. In particular, it observes that when $n > x$, every hero dies in the first round, and verifies the first three public examples individually. However, after repeatedly failing to derive a general counting rule, it begins to infer a pattern directly from these examples:

```
From the examples:
- n=2, x=5: 5 = x
- n=3, x=3: 15 = 3 * 5
- n=5, x=4: 1024 = 4^5

...

For n <= x: x * (x+2)^(n-2)
For n > x: x^n
```

It then implements the guessed rule as

```
if n > x:
    ans = pow(x, n, MOD)
else:
    ans = (x * pow(x + 2, n - 2, MOD)) % MOD
```

Finally, the response claims that the same formula reproduces the public case $n = 13, x = 37$:

```
# n=13, x=37: 976890680
# (n <= x, formula: 37 * 39^11 mod 998244353)
```

Ground truth.
The correct answer for $n = 13$ and $x = 37$ is 976890680. Therefore, the proposed closed-form expression does not describe the general elimination process.

Failure analysis.
The model makes several correct local observations about the dynamics, including the damage received by each surviving hero and the special case in which all heroes are eliminated in the first round. However, it does not derive the combinatorial structure required to count the general case. Instead, after repeatedly noting that it lacks a general formula, it fits the expression $x(x+2)^{n-2}$ to a small number of sample cases and adopts it without a proof or independent validation. Crucially, the response states that the formula matches the examples without actually checking the final sample, for which its own formula gives a different result. The failure therefore arises from replacing an unresolved counting problem with an unsupported pattern inferred from insufficient evidence.

Failure type.
Unsupported generalization from examples and insufficient self-verification.

C.2 MATH500

MATH-500 (Lightman et al., 2024) is a 500-problem subset of the test set of the MATH benchmark (Hendrycks et al., 2021), covering competition-style mathematics across topics such as algebra, geometry, number theory, and counting and probability. Each instance consists of a natural-language mathematical problem together with a reference solution and final answer. The model is prompted to generate a complete reasoning process and provide its final answer in a designated `\boxed{}` format. For evaluation, we extract the final boxed answer, normalize equivalent formatting such as whitespace, LaTeX notation, and numerical representations, and compare it against the reference answer.

We select two representative cases for closer inspection. The correct one illustrates successful transfer of structured mathematical reasoning, the wrong one reveals a remaining failure mode in which an initially correct formulation is overturned by subsequent misinterpretation.

Case 1: Distances to Three Orthogonal Planes

Problem. For a point P, let d_1, d_2 and d_3 represents the distances from P to the planes $x-z=0$, $x-2y+z=0$, and $x+y+z=0$. Let S be the set of points P such that

$$d_1^2 + d_2^2 + d_3^2 = 36.$$

Find the region of the volume enclosed by S.

Student response.
The model first writes the distances to the three planes as

$$d_1 = \frac{|x-z|}{\sqrt{2}}, \qquad d_2 = \frac{|x-2y+z|}{\sqrt{6}}, \qquad d_3 = \frac{|x+y+z|}{\sqrt{3}}.$$

It then observes that the corresponding unit normal vectors are mutually orthogonal. Consequently, the squared distances are exactly the squared projections of (x, y, z) onto an orthonormal basis, giving

$$d_1^2 + d_2^2 + d_3^2 = x^2 + y^2 + z^2.$$
$$x^2 + y^2 + z^2 = 36.$$

The model therefore identifies S as a sphere of radius 6 and computes the enclosed volume as

$$V = \frac{4}{3}\pi(6)^3 = 288\pi.$$

Ground truth. $\boxed{288\pi}$

Capability analysis.
The model does more than directly expand the three distance formulas. It recognizes that the normalized plane normals form an orthonormal basis and interprets the distances as orthogonal projections. This reduces an apparently complicated quadratic constraint to the standard sphere equation $x^2 + y^2 + z^2 = 36$. The reasoning correctly connects analytic geometry, vector orthogonality, and the geometric interpretation of the resulting quadratic form.

Capability demonstrated.
Structural geometric reasoning and cross-representation abstraction.

Case 2: Cars in Three Lanes

Problem. Six cars pull up to a red light, one at a time. At the light, there are three lanes, one left-turn lane, one straight-going lane, and one right-turn lane. How many ways can the cars stack up so that all three lanes are occupied?
Note that if the first car turns left and the second goes straight, this is considered different from the first car going straight and the second car turning left. In other words, the cars are distinguishable, but pull up to the intersection in a fixed order.

Student response.
The model initially interprets the problem correctly as assigning each of the six cars to one of three lanes while requiring every lane to be used. It applies inclusion–exclusion and obtains

$$3^6 - 3 \cdot 2^6 + 3 \cdot 1^6 = 729 - 192 + 3 = 540.$$

However, it subsequently reinterprets the problem as allowing the cars to be independently reordered within each lane. Under this alternative interpretation, it arranges all six cars and inserts two dividers:

$$6!\binom{5}{2} = 720 \times 10 = 7200.$$

The model ultimately discards its earlier result and outputs **7200**.

Ground truth. The correct answer is $\boxed{540}$. Because the cars arrive in a fixed order, once each car is assigned to a lane, its relative position within that lane is already determined. Hence no additional permutation of cars within lanes should be counted. The valid assignments are exactly the surjections from six distinguishable cars to three distinct lanes:

$$3^6 - 3 \cdot 2^6 + 3 = 540.$$

Failure analysis.
The model reaches the correct combinatorial formulation and explicitly derives the correct value 540, but then overrides it after introducing an additional ordering degree of freedom that is excluded by the problem statement. In particular, the fixed arrival order already determines the

relative order of cars sharing a lane, so multiplying by within-lane permutations double-counts configurations. This case is therefore notable not for an inability to derive the correct solution, but for failing to preserve an already correct interpretation when reconsidering the problem.

Failure type.

Constraint misinterpretation and combinatorial overcounting.

C.3 FINREASONING

FinanceReasoning (Tang et al., 2025) evaluates financial numerical reasoning across a broad range of financial concepts and formulas. It extends and refines questions from four existing financial datasets, including 908 newly annotated problems with detailed Python solutions and stricter evaluation criteria. Its more challenging problems require models to select and compose multiple financial formulas while maintaining numerical precision across intermediate calculations. In our evaluation, we adopt the **program-of-Thought** setting, where the model generates a self-contained Python program from the provided context and question, and the executed result is compared with the reference answer. This setting evaluates both the formulation of the required financial computation and its executable realization.

Case 1: Portfolio Variance with Covariance

Problem. An investor is considering forming a portfolio consisting of two stocks, Stock A and Stock B. The investor aims to find the variance of this new portfolio to understand its risk level. Stock A has an expected variance of 0.04, while Stock B has an expected variance of 0.09. The covariance between Stock A and Stock B is 0.02. The investor decides to allocate 40% of their investment in Stock A and the remaining 60% in Stock B.

What is the variance of the portfolio consisting of Stock A and Stock B? Provide the answer as a decimal rounded to four decimal places.

Student response.

```
Let me solve this step by step.

The variance of a two-stock portfolio is calculated using the formula:

sigma_p^2 = w_A^2 x sigma_A^2 + w_B^2 x sigma_B^2 + 2 x w_A x w_B x Cov(A,B)

Where:
- w_A = weight of Stock A = 0.40
- w_B = weight of Stock B = 0.60
- sigma_A^2 = variance of Stock A = 0.04
- sigma_B^2 = variance of Stock B = 0.09
- Cov(A,B) = covariance between Stock A and Stock B = 0.02

Let me calculate:
- w_A^2 x sigma_A^2 = 0.40^2 x 0.04 = 0.16 x 0.04 = 0.0064
- w_B^2 x sigma_B^2 = 0.60^2 x 0.09 = 0.36 x 0.09 = 0.0324
- 2 x w_A x w_B x Cov(A,B) = 2 x 0.40 x 0.60 x 0.02 = 2 x 0.24 x 0.02 = 0.0096

Total variance = 0.0064 + 0.0324 + 0.0096 = 0.0484

Rounded to four decimal places: 0.0484

Let me write the Python program to compute this.
</think>
```

```
# Portfolio variance calculation
w_A = 0.40
w_B = 0.60
var_A = 0.04
var_B = 0.09
cov_AB = 0.02

# Portfolio variance formula: w_A^2 * var_A + w_B^2 * var_B + 2 * w_A * w_B * cov_AB
portfolio_variance = (w_A ** 2) * var_A + (w_B ** 2) * var_B + 2 * w_A * w_B * cov_AB
```

```
# Round to four decimal places
result = round(portfolio_variance, 4)

print(result)
```

Ground truth. $\boxed{0.0484}$

Capability analysis.
The model correctly identifies the two-asset portfolio variance formula and incorporates all three required components: the weighted variance of Stock A, the weighted variance of Stock B, and their covariance term. It then translates the formulation into an executable Python program and obtains the reference value of 0.0484. The case illustrates successful transfer from financial formula selection to precise executable computation under the PoT evaluation setting.

Capability demonstrated.
Financial formula composition and executable quantitative reasoning.

C.4 FINCHAIN

FinChain (Xie et al., 2026) evaluates verifiable multi-step financial reasoning across 58 topics in 12 financial domains. Its problems are generated from parameterized symbolic templates grounded in domain-specific financial formulas, with five templates per topic spanning basic, intermediate, and advanced reasoning levels. Each template is instantiated with randomly sampled variables and paired with executable logic for generating intermediate steps and final answers. This construction supports deterministic verification while primarily testing symbolic financial reasoning rather than retrieval or open-ended analysis.

FinChain was originally designed to evaluate both final-answer correctness and step-level reasoning consistency through CHAINEVAL. In our experiments, to consistent with the other OOD benchmarks, we report Pass@1 based on the correctness of the extracted final answer. We additionally inspect representative model responses to understand the reasoning behavior underlying these aggregate results. The following two cases: a successful example requiring correct financial-concept grounding and multi-step computation, and a failed example in which the local arithmetic is correct but relevant variables are omitted when constructing the overall calculation.

Case 1: Deferred Tax Asset from Warranty Expense

Problem. NVIDIA reported a warranty expense of $102610 in its financial statements under accrual accounting. However, only $39809 of that is deductible for tax purposes in the current year. The applicable tax rate is 28%. What is the resulting Deferred Tax Asset (DTA) from this temporary difference?

Student response.

```
Here's a thinking process that leads to the solution:

1. **Analyze the Request:**

    **Role:** Expert financial analyst.

    **Task:** Calculate the Deferred Tax Asset (DTA) based on provided financial data.

    **Input Data:**
        Warranty Expense (Book): $102,610
        Warranty Expense (Tax Deductible): $39,809
        Tax Rate: 28%

    **Output Format:** Step-by-step calculation, final answer in `\boxed{...}` (bare
     number, no symbols).

2. **Understand the Concept:**
```

```
    *Deferred Tax Asset (DTA)* arises when there is a temporary difference that results
      in taxable income in the future (meaning expenses are recognized for book
     purposes *before* they are recognized for tax purposes, or vice versa).

    Specifically, for a warranty expense:

        **Book (Financial Statements):** Expense is recognized when incurred (accrual
     basis). This reduces Book Income.

        **Tax (Income Tax Return):** Deduction is usually taken when the warranty is
     actually paid/settled (cash basis or specific tax rules). This means the tax
     deduction happens *later* than the book expense.

    *Temporary Difference:* The difference between the Book Expense and the Tax
     Deduction.

    *Calculation Logic:*

        1. Determine the temporary difference (the amount of expense recognized for
     books but not yet for tax).

        2. Multiply this difference by the applicable tax rate.

        3. Since the expense is recognized now for books (reducing taxable income *now*
      for books) but *not* for tax (reducing taxable income *later* for tax), this
     creates a future tax deduction. A future tax deduction is a Deferred Tax Asset.

3. **Step-by-Step Calculation:**

    **Step 1: Identify the Book Value vs. Tax Base (or simply the difference).**

        Book Warranty Expense = $102,610
        Tax Deductible Warranty Expense = $39,809
        The temporary difference is the portion of the expense recognized for financial
      reporting but not yet deductible for tax purposes.
        Temporary Difference = Book Expense - Tax Deduction
        Calculation: $102,610 - $39,809

    **Step 2: Perform the Subtraction.**

        $102,610 - $39,809 = $62,801

    **Step 3: Apply the Tax Rate.**

        Tax Rate = 28% (or 0.28)
        DTA = Temporary Difference $\times$ Tax Rate
        Calculation: $62,801 \times 0.28$

    **Step 4: Perform the Multiplication.**

        $62,801 \times 0.28$
        $62,801 \times 20 = 1,256,020$
        $62,801 \times 8 = 502,408$
        $1,256,020 + 502,408 = 1,758,428$ (Wait, let's do standard multiplication).
        $62,801 \times 0.28$
        $62801 \times 28$
        $62801 \times 8 = 502408$
        $62801 \times 20 = 1256020$
        $1256020 + 502408 = 1758428$
        Adjust decimals (2 places): $17,584.28$

4. **Verification of Logic:**

    Book Expense > Tax Deduction.

    Book Income < Taxable Income (in the current year).

    This creates a "Net Operating Loss" effect or simply a timing difference.

    In the future, when the warranty is paid, the tax deduction will occur.

    Tax savings in the future = (Future Deduction) * Tax Rate.

    The "Future Deduction" is the temporary difference calculated above ($62,801).

    Therefore, DTA = $17,584.28.

5. **Formatting the Output:**
```

```
    Show steps clearly.

    Final answer: `17584.28` inside the box.

    Check constraints: "no currency symbol, no thousands separators, no percent sign,
     no units".

6. **Drafting the Response:**

    *Step 1: Calculate the Temporary Difference.*

        Book Expense: $102,610
        Tax Deductible Amount: $39,809
        Difference: $102,610 - $39,809 = $62,801.

    *Step 2: Calculate the Deferred Tax Asset.*

        The difference represents a future deductible amount.
        DTA = Temporary Difference $\times$ Tax Rate.
        DTA = $62,801 \times 0.28$.

    *Step 3: Final Calculation.*

        $62,801 \times 0.28 = 17,584.28$.

7. **Final Review:**

    Did I interpret "warranty expense" correctly? Yes, warranty expense is accrued on
     books but often deducted on tax returns when paid (or based on actuals). If the
     book expense is higher than the tax deduction, it creates a DTA.

    Is the arithmetic correct?

        $102610 - 39809 = 62801$. Correct.
        $62801 \times 0.28 = 17584.28$. Correct.

    Are formatting constraints met? Yes.

8. **Construct Final Output.** (Proceed to generate output based on these steps).

</think>

To calculate the Deferred Tax Asset (DTA), we need to determine the temporary
     difference between the book value (financial accounting) and the tax base (tax
     accounting) regarding the warranty expense.

**Step 1: Calculate the Temporary Difference**

The temporary difference is the amount of the warranty expense recognized for financial
      reporting purposes (accrual) that has not yet been recognized for tax purposes (
     deductible).

**Book Warranty Expense:** $102,610

**Tax Deductible Warranty Expense:** $39,809

$$ \text{Temporary Difference} = \text{Book Expense} - \text{Tax Deduction} $$

$$ \text{Temporary Difference} = 102,610 - 39,809 $$

$$ \text{Temporary Difference} = 62,801 $$

This $62,801 represents an expense that reduced book income in the current year but
     will reduce taxable income in future years when the warranty claims are actually
     paid.

**Step 2: Calculate the Deferred Tax Asset**

A future tax deduction creates a Deferred Tax Asset. We multiply the temporary
     difference by the applicable tax rate.

**Temporary Difference:** $62,801

**Tax Rate:** 28% (0.28)

$$ \text{Deferred Tax Asset} = \text{Temporary Difference} \times \text{Tax Rate} $$
```

```
$$ \text{Deferred Tax Asset} = 62,801 \times 0.28 $$

**Calculation:**

$$ 62,801 \times 0.28 = 17,584.28 $$

\boxed{17584.28}
```

Ground truth.

```
Step 1: Determine the portion of warranty expense not deductible in the current tax
     year.
        Temporary Difference = Reported Expense - Tax Deductible Now
                             = $102610 - $39809 = $62801

Step 2: Apply the tax rate to the temporary difference to compute the Deferred Tax
     Asset.
        DTA = Temporary Difference $\times$ Tax Rate
            = $62801 $\times$ (28/100)

Step 3: Perform the calculation.
        DTA = $17584.28

Step 4: Conclusion
        The Deferred Tax Asset arising from the temporary difference in warranty
     expense is $17584.28.
```

Capability analysis.

The model correctly identifies the temporary difference between the book-recognized warranty expense and the amount currently deductible for tax purposes, obtaining 62,801. It then correctly interprets this difference as a future tax deduction and applies the 28% tax rate to obtain the Deferred Tax Asset of 17,584.28. The success therefore reflects both correct financial-accounting interpretation and accurate multi-step computation.

Capability demonstrated.

Accounting concept grounding and multi-step quantitative reasoning.

Case 2: Multi-Factor Crypto Sentiment Score

Problem. For a crypto asset, sentiment is computed from multiple factors. It received 998 tweets with an average polarity of 0.87 and an engagement factor of 0.13. Additionally, there were 39 news articles with an average sentiment of 1.26 and a news influence of 0.14. Finally, the asset had a price change of $6.73 adjusted by a factor of 0.30. If the overall sentiment score is the sum of the tweet-based, news-based, and price-based sentiment scores, what is the overall sentiment score?

Student response.

```
Here's a thinking process that leads to the solution:

1. **Analyze the Request:**
    **Role:** Expert financial analyst.
    **Task:** Calculate the overall sentiment score for a crypto asset based on
     provided data.
    **Method:** Step-by-step calculation showing intermediate steps.
    **Output Format:** Final answer in `\boxed{...}`. No currency symbols, no commas,
     no percent signs, no units.

2. **Deconstruct the Problem:**
    **Goal:** Calculate `Overall Sentiment Score`.
    **Formula:** `Overall Sentiment Score` = `Tweet-based Score` + `News-based Score` +
      `Price-based Score`.

3. **Analyze the Components:**

    **Component 1: Tweet-based Score**

        Data provided:
```

```
            Number of tweets = 998 (Likely irrelevant for the calculation unless
      specified as a multiplier, but usually sentiment scores are weighted by engagement
      /polarity).
            Average polarity = 0.87.
            Engagement factor = 0.13.

         *Interpretation:* How is the score derived? Usually, it's a weighted average or
       a product of the factors. Given the phrasing "sentiment is computed from multiple
       factors", and the specific mention of "average polarity" and "engagement factor",
       a standard approach in these types of logic puzzles is to multiply the sentiment
      metric (polarity) by the weight/impact metric (engagement factor). Alternatively,
      it could be a sum, but "factor" usually implies multiplication in this context.
      Let's look at the other components to see if a pattern emerges.

         *Hypothesis 1 (Multiplication):* $Score_{tweet} = Polarity \times Engagement$

         *Hypothesis 2 (Sum):* $Score_{tweet} = Polarity + Engagement$ (Unlikely for "
      factors").

         *Hypothesis 3 (Weighted Sum):* $Score_{tweet} = (Polarity \times Weight) + ...$
       (No weights given other than the factor).

         *Decision:* The most logical interpretation of "score computed from... average
      polarity... and engagement factor" is the product: $0.87 \times 0.13$. *Self-
      Correction/Refinement:* Sometimes "factor" implies a weight applied to the value.
      Let's calculate $0.87 \times 0.13$.

         Calculation: $0.87 \times 0.13 = 0.1131$.

    **Component 2: News-based Score**

         Data provided:
             Number of articles = 39 (Likely irrelevant for the score calculation itself
      , just volume).
             Average sentiment = 1.26.
             News influence = 0.14.

         *Interpretation:* Similar to tweets, the score is likely the sentiment
      multiplied by the influence factor.

         Calculation: $1.26 \times 0.14$.
         $1.26 \times 0.10 = 0.126$
         $1.26 \times 0.04 = 0.0504$
         $0.126 + 0.0504 = 0.1764$.

    **Component 3: Price-based Score**

         Data provided:
             Price change = $6.73.
             Adjustment factor = 0.30.

         *Interpretation:* The score is the price change adjusted by the factor.

         Calculation: $6.73 \times 0.30$.
         $6.73 \times 3 = 20.19$
         $20.19 / 10 = 2.019$.

4. **Synthesize and Calculate:**

    **Tweet-based Score:** $0.87 \times 0.13 = 0.1131$
    **News-based Score:** $1.26 \times 0.14 = 0.1764$
    **Price-based Score:** $6.73 \times 0.30 = 2.019$

    **Total:** $0.1131 + 0.1764 + 2.019$

    *Step-by-step Addition:*
        $0.1131 + 0.1764 = 0.2895$
        $0.2895 + 2.019 = 2.3085$

5. **Review and Verification:**

    Does the logic hold?

         "Tweet-based sentiment score" = Polarity * Engagement? Yes, this is standard
      for sentiment analysis weighting.
         "News-based sentiment score" = Sentiment * Influence? Yes.
         "Price-based sentiment score" = Price Change * Factor? Yes.
         "Overall" = Sum of components? Yes, explicitly stated in the prompt.
```

```
    Are there any alternative interpretations?

       What if the number of tweets/articles matters? Usually, in these specific types
     of prompt-based math problems, extraneous data (like "998 tweets") is provided to
     test if the solver can filter relevant information. If I were to weight by volume
     (e.g., $0.87 \times 0.13 \times 998$), the numbers would explode and likely not
    make sense as a "sentiment score" (which usually stays within a normalized range
    or a specific financial metric range). The prompt asks for "sentiment scores"
    derived from the factors. The most direct mapping is Value $\times$ Factor.

       Let's check the wording again: "received 998 tweets... Additionally, there were
     39 news articles... Finally, the asset had a price change...".

       If the calculation required volume, it would usually say "weighted by volume"
    or "total score". Since it asks for the "tweet-based sentiment score" derived from
     "average polarity" and "engagement factor", the volume (998) is context, not a
    calculation variable.

       Let's double-check the math.

           $87 \times 13$: $80 \times 13 = 1040$; $7 \times 13 = 91$; $1040 + 91 =
    1131$. Decimal places: 4. -> $0.1131$. Correct.

           $126 \times 14$: $126 \times 10 = 1260$; $126 \times 4 = 504$; $1260 + 504
    = 1764$. Decimal places: 4. -> $0.1764$. Correct.

           $673 \times 3$: $600 \times 3 = 1800$; $70 \times 3 = 210$; $3 \times 3 = 9
    $; $1800+210+9 = 2019$. Decimal places: 3 (since $0.30$ is effectively 2 decimal
    places, $6.73$ is 2, total 4? No, $6.73 \times 0.30$. $6.73 \times 3 = 20.19$. $20
    .19 \times 0.1 = 2.019$. Correct.

           Sum:
               $0.1131$
             $+ 0.1764$
             $+ 2.0190$
             -----------
               $0.2895$
             $+ 2.0190$
             -----------
               $2.3085$

6. **Formatting:**

    The prompt asks for the final answer inside `\boxed{...}`.
    No currency symbol ($).
    No thousands separators (commas).
    No percent sign (%).
    No units.
    Just the number.

    Result: 2.3085

7. **Final Polish:**

    Write down the steps clearly as requested.
    State the formula used for each component.
    Sum them up.
    Box the final number.

    *Self-Correction on "Price Change":* Is it possible the price change is a
     percentage? It says "$6.73". That's a dollar amount. The factor is 0.30. So $6.73
     * 0.30 = 2.019$. This seems consistent.

    *Self-Correction on "Sentiment":* Sentiment scores can be negative. Here all inputs
      are positive.

    *Self-Correction on "Tweets/Articles count":* I will explicitly mention that these
     counts are not used in the calculation to justify the exclusion of 998 and 39.

    *Wait, is there any ambiguity about "engagement factor"?* Could it be added? "
     Average polarity of 0.87 and an engagement factor of 0.13". If I add them: $0.87 +
      0.13 = 1.0$. If I multiply: $0.1131$. In financial modeling contexts (like the
     one implied by "financial analyst"), "factors" are almost exclusively multipliers.
      If it were a sum, it would usually say "plus". I will proceed with multiplication.
```

```
*Is there any ambiguity about "price change adjusted by a factor"?* "Adjusted by"
 implies multiplication (scaling).

*Is there any ambiguity about the sum?* "Overall sentiment score is the sum of..."
 - Explicit.

*Final check of the calculation:*
    Tweet: $0.87 \times 0.13 = 0.1131$
    News: $1.26 \times 0.14 = 0.1764$
    Price: $6.73 \times 0.30 = 2.019$
    Sum: $0.1131 + 0.1764 + 2.019 = 2.3085$

The prompt asks for "the overall sentiment score".
I will output 2.3085.

Wait, could "price change" be treated differently? If the price change is $6.73,
 and the factor is 0.30.
Is it possible the user expects me to treat the count (998, 39) as weights?
"received 998 tweets...". If I calculate $0.87 \times 0.13 \times 998$, that equals
  1131.
If I calculate $1.26 \times 0.14 \times 39$, that equals 6.9444.
If I calculate $6.73 \times 0.30$, that equals 2.019.
Sum = $1131 + 6.94 + 2.019 \approx 1140$.
Which makes more sense? A "Sentiment Score" of 1140 is high, but possible. A "
 Sentiment Score" of 2.3 is also possible.
However, usually, "factors" (engagement factor, news influence, adjustment factor)
 are designed to normalize or weight the raw sentiment (polarity/sentiment). The
 raw counts (998, 39) are usually volume metrics. If the prompt wanted volume
 weighting, it would likely say "weighted by the number of tweets".
Furthermore, "Engagement factor" usually implies a metric like 0.13 (low to high).
 "News influence" 0.14. "Adjustment factor" 0.30. These look like weights.
If I use the counts, the "Engagement factor" becomes redundant if the count is used
  as the weight.
Therefore, the most robust interpretation is: Score = (Metric) * (Factor). The
 counts are distractors.

Let's check the constraints again. "Expert financial analyst". In analyst reports,
 you aggregate signals.
Signal 1: Polarity (0.87) * Engagement (0.13).
Signal 2: Sentiment (1.26) * Influence (0.14).
Signal 3: Price ($6.73) * Factor (0.30).
This seems the most consistent with the provided variables.

Let's check if there's any other interpretation of "adjusted by".
If price change is $6.73, and adjusted by 0.30.
Maybe it's $6.73 / 0.30$? No, "adjusted by" usually means scaling.
Maybe it's $6.73 + 0.30$? No, factors are multipliers.

Okay, I'm confident in the multiplication approach.

One more check: "average sentiment of 1.26". Sentiment scores often range -1 to 1,
 or 0 to 1, or 0 to 100. 1.26 is slightly above 1. This is fine.
Polarity 0.87 is within 0-1.
The resulting score 2.3085 is the sum of these weighted components.

Final check on formatting: "no currency symbol, no thousands separators, no percent
  sign, no units, no extra words".
So, `2.3085`.

Wait, is it possible the "price change" is a percentage? "$6.73". The dollar sign
 indicates currency, not percentage. So it is an absolute value.

Let's write out the steps.
Step 1: Calculate Tweet-based score.
Step 2: Calculate News-based score.
Step 3: Calculate Price-based score.
Step 4: Sum them.

Wait, I need to make sure I don't box the unit "2.3085" with a label. Just the
 number.

Double check the math one last time.

$0.87 \times 0.13$:
    $87 \times 13 = 87 \times (10 + 3) = 870 + 261 = 1131$.
    Decimals: 2 + 2 = 4.
    $0.1131$.
```

```
    $1.26 \times 0.14$:
        $126 \times 14 = 126 \times (10 + 4) = 1260 + 504 = 1764$.
        Decimals: 2 + 2 = 4.
        $0.1764$.

    $6.73 \times 0.30$:
        $673 \times 30 = 20190$.
        Decimals: 2 + 2 = 4.
        $2.0190$.

    Sum:
        $0.1131$
        $0.1764$
        $2.0190$
        -------
        $1+4+0 = 5$
        $3+6+9 = 18$ (carry 1)
        $1+7+1+1 = 10$ (carry 1)
        $1+1+0+1 = 3$
        $0+0+2 = 2$
        Result: $2.3085$.

    Looks solid.

    One nuance: "price change of $6.73 adjusted by a factor of 0.30".
    Does "adjusted by" mean added to? No.
    Does it mean divided by? No.
    Does it mean multiplied? Yes.

    Another nuance: "average polarity of 0.87 and an engagement factor of 0.13".
    Could it be $0.87 + 0.13$?
    If I add them: $1.0$.
    If I add news: $1.26 + 0.14 = 1.4$.
    If I add price: $6.73 + 0.30 = 7.03$.
    Total: $1.0 + 1.4 + 7.03 = 9.43$.
    However, "factor" strongly implies multiplication in mathematical modeling contexts
      (e.g. "influence factor", "adjustment factor"). A "factor" modifies the base
     value. An "engagement factor" of 0.13 suggests that the engagement is low (13%),
     so it should dampen the polarity score. If it were additive, a factor of 0.13 adds
      13 points to a 0.87 polarity? That would make the score 1.0. While possible,
     multiplication is the standard interpretation of "factor" in weighted scoring
     models. I will stick with multiplication.

    Wait, looking at the price change: $6.73. If this is a dollar amount, and the
     factor is 0.30.
    If I multiply, I get ~2.
    If I add, I get ~7.
    Given the other scores are small (0.11, 0.17), having a price score of 2.019 makes
     the total 2.3.
    If I added, the total would be 9.43.
    Usually, sentiment scores are normalized. 2.3 is a plausible "score". 9.43 is less
     "normalized" but possible.
    However, "adjusted by a factor" is the key phrase. If I have a value of 100 and
     adjust it by a factor of 0.5, I get 50. I do not get 100.5.
    So multiplication is the only linguistically correct interpretation of "adjusted by
      a factor".

    Final check on "998 tweets" and "39 news articles".
    If these were meant to be used, the problem would likely ask for a "volume-weighted
      sentiment". The prompt asks for "sentiment score... computed from multiple
     factors". The factors listed are Polarity, Engagement, Sentiment, Influence, Price
     , Factor. The counts are not labeled as "factors".
    Therefore, counts are distractors.

    Plan:
    1. Define the formula for each component (Metric * Factor).
    2. Calculate Tweet component.
    3. Calculate News component.
    4. Calculate Price component.
    5. Sum them.
    6. Box result.

    Wait, is "price change" a metric? Yes. "Adjusted by a factor". Yes.
    Is "news influence" a factor? Yes.
    Is "engagement factor" a factor? Yes.

    Ready to write.
</think>
```

```
To calculate the overall sentiment score, we need to compute the individual scores for
     the tweet-based, news-based, and price-based components, and then sum them up. The
      problem provides specific metrics and factors for each component. We interpret "
     factors" (engagement factor, news influence, adjustment factor) as multipliers
     applied to the base metrics (average polarity, average sentiment, price change).
     The volume counts (998 tweets, 39 news articles) are treated as context and not
     included in the calculation, as the factors provided serve as the weights.

**Step 1: Calculate the Tweet-based Sentiment Score**

**Base Metric:** Average Polarity = 0.87

**Factor:** Engagement Factor = 0.13

**Calculation:** $0.87 \times 0.13$

    $0.87 \times 0.10 = 0.087$
    $0.87 \times 0.03 = 0.0261$
    $0.087 + 0.0261 = 0.1131$

**Tweet-based Score:** 0.1131

**Step 2: Calculate the News-based Sentiment Score**

**Base Metric:** Average Sentiment = 1.26

**Factor:** News Influence = 0.14

**Calculation:** $1.26 \times 0.14$

    $1.26 \times 0.10 = 0.126$
    $1.26 \times 0.04 = 0.0504$
    $0.126 + 0.0504 = 0.1764$

**News-based Score:** 0.1764

**Step 3: Calculate the Price-based Sentiment Score**

**Base Metric:** Price Change = $6.73

**Factor:** Adjustment Factor = 0.30

**Calculation:** $6.73 \times 0.30$

    $6.73 \times 0.3 = 2.019$

**Price-based Score:** 2.019

**Step 4: Calculate the Overall Sentiment Score**

**Formula:** Tweet-based Score + News-based Score + Price-based Score

**Calculation:** $0.1131 + 0.1764 + 2.019$

    $0.1131 + 0.1764 = 0.2895$
    $0.2895 + 2.019 = 2.3085$

The overall sentiment score is 2.3085.

\boxed{2.3085}
```

Ground truth.

```
Step 1: Compute tweet-based sentiment: 998 * 0.87 * 0.13 = 112.87.
Step 2: Compute news-based sentiment: 39 * 1.26 * 0.14 = 6.88.
Step 3: Compute price-based sentiment: 6.73 * 0.30 = 2.02.
Step 4: Sum all components: 112.87 + 6.88 + 2.02 = 121.77.
```

Failure analysis.

The arithmetic is internally consistent with the formula chosen by the model, but the formula itself omits two relevant quantities from the problem. In particular, the model explicitly treats the numbers of tweets and news articles, 998 and 39, as contextual distractors. Under the benchmark specification, these quantities scale the corresponding average sentiment signals,

yielding $998 \times 0.87 \times 0.13 = 112.87$ and $39 \times 1.26 \times 0.14 = 6.88$. Omitting these volume terms reduces the prediction to 2.3085 instead of the reference value 121.77. The error therefore arises before arithmetic execution, from an incorrect composition of the variables specified in the problem.

Failure type.

Relevant-variable omission and incorrect factor composition.