THINK LESS, LOCALLY: LOCAL PREFERENCE LEARNING FOR REASONING COMPRESSION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large reasoning models achieve strong performance on challenging tasks, but their long chain-of-thought often incurs substantial inference costs by continuing to deliberate after sufficient information has been obtained. Existing reasoning compression methods primarily operate on reasoning trajectories, paying less attention to the local decision of whether further reasoning is necessary given the current state. We propose LoCo, a local preference learning method for reasoning compression. LoCo detects candidate overthinking states from the model's own traces, freezes the corresponding reasoning prefixes, regenerates shorter verified continuations from the same states, and trains on the resulting state-conditioned preference pairs with DPO. This formulation teaches the model when to continue reasoning and when to conclude, rather than simply encouraging shorter trajectories overall. We construct training data entirely from self-generated mathematical traces and evaluate LoCo on five mathematical and three code generation benchmarks. LoCo reduces completion length by 41.7%–45.4% on mathematics and 23.4%–36.3% on code, while maintaining or improving accuracy and achieving the best AUC across all code benchmarks. LoCo consistently improves the accuracy-efficiency trade-off on mathematics and transfers effectively to code generation despite math-only training, demonstrating the cross-domain generality of local preference learning for reasoning compression.

1 INTRODUCTION

Large reasoning models (LRMs) (Guo et al., 2025) have made remarkable progress on complex reasoning tasks such as mathematical problem solving and algorithmic programming. This progress is largely driven by extended chain-of-thought (CoT) reasoning, in which models generate thousands of tokens for hypothesis formation, calculation, revision, and verification before arriving at an answer (Muennighoff et al., 2025). Such extended reasoning comes at a substantial inference cost, significantly increasing decoding latency and memory consumption during deployment. More importantly, longer reasoning does not necessarily lead to higher accuracy, as models may continue reasoning beyond what is necessary to reach a correct answer (Chen et al., 2024; Sui et al., 2025). This tendency to engage in excessive reasoning is commonly referred to as *overthinking*.

Recent studies indicate that when the task is solvable by the model, overthinking is more closely related to model preference than reasoning capability (Zhang et al., 2026a; Peng et al., 2025). A model may get the correct answer early in the reasoning process yet continue generating unnecessary reasoning, reflecting a preference to keep thinking rather than to stop when sufficient reasoning has been reached. The problem therefore lies in when the model chooses to stop, rather than whether it can reach the correct answer (Chen et al., 2024; Zhang et al., 2026a). Mitigating overthinking thus calls for aligning models toward concise reasoning behaviors, which can be learned from carefully curated self-generated reasoning traces (Shen et al., 2025; Zhang et al., 2026b).

Toward this objective, existing self-generated methods derive compression supervision in two main ways, through trajectory-level signals and fine-grained operations within the reasoning process. Trajectory-level signals are derived from the reasoning trajectory as a whole, using global properties of complete solutions to establish a preference for more concise reasoning. Specifically, one line of work (Team et al., 2025; Luo et al., 2026; Shen et al., 2025; Ma et al., 2025) constructs preference pairs by comparing shorter and longer correct solutions, encouraging the model to favor

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.

the shorter one, while another (Arora & Zanette, 2026; Aggarwal & Welleck, 2025; Hu et al., 2026) incorporates length preferences into reinforcement learning by rewarding shorter correct reasoning.

Fine-grained operations provide a more localized alternative by modifying parts of an existing reasoning trajectory. Specifically, these methods remove redundant content (Xia et al., 2025), compress individual reasoning steps (Jang et al., 2024), or shorten continuations from intermediate reasoning states to construct training data for more concise reasoning (Lai et al., 2024; Jin et al., 2025). Such operations are well aligned with recent findings (Xu et al., 2026) that LLM reasoning proceeds through local and incremental state transitions rather than a globally planned process. However, existing fine-grained methods still treat the reasoning trajectory itself as the object of compression. They focus on which parts of the trajectory should be removed or shortened, rather than whether further reasoning is necessary given the model's current state. This distinction is crucial because the appropriate compression decision can vary across reasoning states: the model may need to continue reasoning at one state but should conclude at another.

Building on this observation, we propose **LoCo** (**Lo**cal **Co**mpression), a local preference learning method for reasoning compression. LoCo identifies overthinking states from the model's own reasoning process and constructs state-conditioned preferences by contrasting alternative continuations from the same reasoning state. The resulting supervision is grounded in the model's own reasoning trajectories and actual reasoning behavior. By making the compression decision conditional on the current reasoning state, LoCo encourages the model to conclude when further reasoning is unnecessary while preserving its ability to continue reasoning when additional computation is needed. In this way, LoCo avoids indiscriminate compression across the reasoning trajectory and reduces the risk of compromising the model's reasoning capability on more challenging problems.

We construct training data from self-generated traces and evaluate LoCo on five mathematical reasoning benchmarks and three coding benchmarks spanning a broad range of difficulty. The experiments show that LoCo consistently achieves the strongest efficiency–performance trade-off on mathematics, with especially clear advantages on the harder datasets where both high compression and high accuracy are required. The same trained model also transfers effectively to coding benchmarks of different difficulty, supporting the view that LoCo improves chain-of-thought compression by reshaping the model's reasoning preference rather than by injecting additional task-specific knowledge. To summarize, our main contributions are threefold:

- We show that effective reasoning compression requires state-aware decisions about when further reasoning is unnecessary, rather than simply favoring shorter trajectories.
- We introduce LoCo, a local preference learning framework that identifies overthinking states and learns continue-or-conclude preferences from verified continuations sharing the same reasoning prefix.
- We demonstrate the effectiveness of LoCo across mathematical reasoning and code generation, with consistent improvements in the efficiency-performance trade-off and strong cross-domain generalization.

2 Related Works

2.1 Trajectory-Level Compression

Several reasoning compression methods formulate compression supervision at the trajectory level, treating complete reasoning traces as the unit of comparison. Kimi k1.5 (Team et al., 2025) introduces a Long2Short stage that contrasts correct short and long solutions generated by the model itself and optimizes their relative preference with DPO. O1-Pruner (Luo et al., 2026) follows a similar paradigm by applying preference fine-tuning to pre-sampled responses, encouraging the model to favor more concise reasoning. CoT-Valve (Ma et al., 2025) identifies a parameter-space direction for controlling reasoning length, then learns from globally shortened reasoning traces. Another group of methods (Arora & Zanette, 2026; Aggarwal & Welleck, 2025; Hu et al., 2026) incorporates reasoning length into the reinforcement learning objective, jointly rewarding solution quality and efficient token usage under a prescribed length budget. Collectively, these studies show that reasoning can often be shortened without substantially affecting problem-solving performance. However, these methods generally compare or optimize complete reasoning trajectories, which provides an overall

preference for shorter solutions but offers less information about where the unnecessary reasoning occurs. Two solutions may reach the same answer through different reasoning paths, making it difficult to determine which part of the reasoning can be safely shortened from their overall length or preference alone. This motivates a more fine-grained approach that identifies unnecessary deliberation at specific points in the reasoning process.

2.2 Fine-Grained Compression

Fine-grained operations offer a more localized way to reduce reasoning by modifying selected parts of an existing trajectory. These methods remove redundant content, compress individual reasoning steps, or shorten continuations from intermediate states, in line with recent findings that LLM reasoning proceeds through local and incremental state transitions rather than a globally planned process (Xu et al., 2026). One line of work removes redundant content from the trajectory itself. TokenSkip (Xia et al., 2025) prunes verbose tokens, Verbosity-Aware Rationale Reduction (Jang et al., 2024) removes redundant sentences based on principled likelihood criteria, and Reasoning Shaping (Song et al., 2025) reweights computation-heavy segments during RLVR training. TokenSqueeze (Zhang et al., 2026b) combines depth-matched data selection with distribution-aligned token-level linguistic refinement and achieves strong performance on standard benchmarks. Another line operates at the step level. Step-DPO (Lai et al., 2024) introduces preference pairs of correct and erroneous continuations sharing the same prefix, while SSPO (Xu et al., 2025) and ReCUT (Jin et al., 2025) adapt this formulation to reasoning compression and encourage more concise step-level reasoning. Despite their finer granularity, these approaches mainly derive compression signals by contrasting or modifying existing reasoning content. Such supervision is effective for identifying unnecessary tokens, sentences, or steps, but provides less direct guidance on the decision to continue reasoning from a particular state. LoCo addresses this aspect by anchoring the comparison at a fixed reasoning state and learning preferences over its alternative continuations. This formulation captures the local decision of whether further deliberation is warranted while retaining the original state as a shared context.

3 Preliminaries

Large Reasoning Models and Reasoning States. Large reasoning models (LRMs), such as DeepSeek-R1 (Guo et al., 2025), are trained to perform extended reasoning before producing a final answer. Given a problem q, a model first generates a chain of thought (CoT) $S = (s_0, s_1, \ldots, s_n)$, consisting of intermediate hypotheses, calculations, revisions, and verifications, before arriving at the final answer. The reasoning process is typically enclosed by dedicated thinking tags, such as `<think>` and `</think>` in DeepSeek-R1 (Guo et al., 2025), with paragraph breaks naturally separating different reasoning moves. We therefore segment the CoT into reasoning steps at paragraph boundaries and denote each step by s_i. A reasoning step may correspond to establishing a subgoal, performing a computation, verifying an intermediate result, or drawing a conclusion.

We denote the input prompt associated with question q as $\mathrm{prompt}(q)$, which includes the task instruction and the question itself. For notational convenience, we define the reasoning state after step s_i as

$$P_i = \mathrm{prompt}(q) \,\|\, s_0 \,\|\, \cdots \,\|\, s_i, \tag{1}$$

where $\|$ denotes concatenation. Thus, P_i contains all information available after completing s_i and before generating s_{i+1}. Each state therefore defines a local decision point at which the model can either continue reasoning or conclude. We view overthinking as continuing to generate additional reasoning after the current state already contains sufficient information to support the answer. This state-based formulation provides the notation used throughout the method.

4 Method

4.1 Overview

LoCo converts a reasoning model's own overthinking traces into state-anchored preference supervision through three stages, as illustrated in Figure 1. First, local overthink detection screens candidate

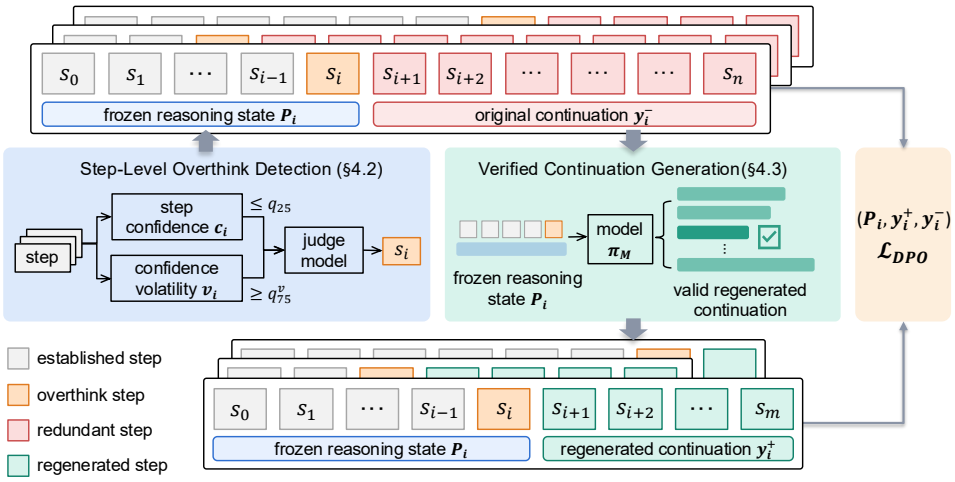


Figure 1: Overview of the LoCo pipeline. Starting from a self-generated reasoning trace, LoCo detects local overthinking states, regenerates shorter correct continuations from those states, and converts them into local preference pairs for training.

reasoning states using a two-stage filter in Section 4.2. Second, verified continuation generation samples shorter correct continuations from each detected state in Section 4.3. Third, local preference learning contrasts the regenerated continuation with the original continuation from the same state for DPO training in Section 4.4. Using the notation introduced in Section 3, this process produces a set of triples $\{(P_i, y_i^+, y_i^-)\}$, where P_i denotes the frozen reasoning state after step s_i, y_i^+ denotes a shorter correct continuation regenerated from P_i, and y_i^- denotes the original continuation that follows P_i. We denote the reasoning model being trained as M.

4.2 Step-Level Overthink Detection

We identify reasoning states after which the model begins to overthink using its predictive confidence. Since semantic heuristics can be unreliable and step-level annotation is costly, we first screen candidate states with model confidence and then verify them with a stronger judge model.

Confidence Screening. We feed the complete chain of thought back to M and score it under its own next-token distribution. Let S denote the CoT token sequence, $S_{<\tau}$ its prefix preceding position τ, and $\pi_M(\cdot \mid S_{<\tau})$ the corresponding next-token distribution. For each position, we record

$$p_\tau^{\max} = \max_{v \in \mathcal{V}} \pi_M(v \mid S_{<\tau}), \tag{2}$$

and let $\mathcal{T}_i$ denote the token positions in step s_i, with steps delimited by paragraph breaks. The step confidence is

$$c_i = \frac{1}{|\mathcal{T}_i|} \sum_{\tau \in \mathcal{T}_i} p_\tau^{\max}. \tag{3}$$

We measure confidence volatility over a sliding window of recent steps:

$$v_i = \frac{1}{|W_i|} \sum_{j \in W_i} (c_j - \bar{c}_i)^2, \tag{4}$$

where W_i contains the most recent steps ending at i. We use the minimal window $|W_i| = 2$, giving

$$v_i = \frac{(c_i - c_{i-1})^2}{4}, \tag{5}$$

with $v_0 = 0$. Thus, c_i measures the model's average predictive confidence within step s_i, while v_i captures its change across consecutive steps. Averaging over tokens makes the signal less sensitive to isolated low-probability tokens caused by punctuation or formatting.

Overthink Detection. We normalize both signals using the model's own behavior. Let q_{25} and q_{75} be the 25th and 75th percentiles of step confidence, and q^v_{25} and q^v_{75} the corresponding percentiles of volatility, computed once from CoT traces self-generated by M. Following the regime analysis of Li et al. (2026), we identify overthinking states as those whose most recent completed step has simultaneously low confidence and high volatility:

$$\mathcal{O} = \{i : \; c_i \leq q_{25} \; \wedge \; v_i \geq q^v_{75}\}. \tag{6}$$

The conjunction of the two signals distinguishes overthinking from ordinary uncertainty or state transitions. Low confidence can arise from a genuinely difficult sub-problem, while high volatility can accompany a legitimate change in reasoning. Their combination instead identifies states where the model is both uncertain and unstable. The detector is causal by construction: both c_i and v_i are computed from information available by the end of step s_i, whereas truncation begins only from s_{i+1}. Hence, the detector never uses the continuation it removes.

Judge Re-checking. Confidence alone cannot reliably distinguish redundant deliberation from genuine difficulty. We therefore re-examine each flagged state with a stronger judge model J. The two stages use different evidence. The screen identifies candidates solely from the confidence statistics (c_i, v_i), while the judge reads the current reasoning state and determines whether the next step is redundant. The next step is classified as OVERTHINK if it re-deliberates on information that has already been established, NORMAL if it makes substantive progress, or FINAL if it presents the answer. This distinction matters because truncation from P_i should remove only redundant continuation. Removing a NORMAL step discards useful progress, while removing a FINAL step may discard the answer itself. This case is particularly relevant because reasoning models often conclude their CoT by restating or boxing the final answer, which may appear repetitive but is necessary for answer extraction.

For each $i \in \mathcal{O}$, the judge receives the problem q, the frozen state P_i, the next step s_{i+1}, and the screen prediction $\hat{\ell}_i$. It returns a corrected role and a rationale:

$$\ell_i = J\big(q, \; P_i, \; s_{i+1}, \; \hat{\ell}_i\big) \in \mathcal{L}, \qquad \mathcal{L} = \big\{\text{OVERTHINK}, \text{NORMAL}, \text{FINAL}\big\}, \tag{7}$$

where $\hat{\ell}_i = $ OVERTHINK for all screened candidates. We retain only the states whose next step is confirmed as redundant by the judge:

$$\mathcal{O} \leftarrow \big\{i \in \mathcal{O} : \ell_i = \text{OVERTHINK}\big\}. \tag{8}$$

4.3 Verified Continuation Generation

For each $i \in \mathcal{O}$, we freeze the reasoning state P_i and sample k independent continuations from M conditioned on that state:

$$y_i^{(1)}, \ldots, y_i^{(k)} \sim \pi_M\big(\cdot \mid P_i\big), \tag{9}$$

A continuation is considered valid if it completes the reasoning template and produces an answer symbolically equivalent to the gold answer z, denoted by $y \models z$. Among all valid continuations, we select the shortest one:

$$\mathcal{V}_i = \big\{j : y_i^{(j)} \models z\big\}, \qquad y_i^+ = \arg\min_{j \in \mathcal{V}_i} \big|y_i^{(j)}\big|, \tag{10}$$

where $|\cdot|$ denotes the number of tokens. If $\mathcal{V}_i$ is empty, we discard the detection point. Otherwise, we define the original continuation from the same state as $y_i^- = (s_{i+1}, s_{i+2}, \ldots)$ and retain the point only when the selected continuation is strictly shorter:

$$\mathcal{O} \leftarrow \big\{i \in \mathcal{O} : |y_i^+| < |y_i^-|\big\}. \tag{11}$$

4.4 Local Preference Learning

For each surviving detection point $i \in \mathcal{O}$, we construct the preference triple $(P_i, \; y_i^+, \; y_i^-)$: the two continuations complete the identical frozen state P_i, the chosen y_i^+ is the shortest verified continuation, and the rejected y_i^- is the original continuation. Writing $g_i = |y_i^-| - |y_i^+|$ for the token savings

of the pair, we retain a pair only when the savings reach a minimum δ, the prompt fits within c_p tokens, and each continuation fits within c_r tokens. Since larger savings indicate clearer compression signals, at most K pairs are kept per question, preventing questions with many detection points from dominating the training data:

$$\mathcal{D} = \left\{ \left(P_i,\ y_i^+,\ y_i^-\right) \ :\ i \in \mathcal{S} \right\}, \qquad \mathcal{S} = \left\{ i \in \mathcal{O} :\ g_i \geq \delta,\ \left|P_i\right| \leq c_p,\ \left|y_i^{\pm}\right| \leq c_r \right\}, \tag{12}$$

where $\mathcal{S}$ is truncated within each question to its K largest-g_i elements.

The shared prefix provides a natural state-conditional preference objective. With $x = P_i$, we optimize the DPO objective

$$\mathcal{L}_{\mathrm{DPO}} = -\mathbb{E}_{(x,\, y^+,\, y^-)\sim\mathcal{D}} \left[\log \sigma \left(\beta \left[\log \frac{\pi_\theta(y^+|x)}{\pi_{\mathrm{ref}}(y^+|x)} - \log \frac{\pi_\theta(y^-|x)}{\pi_{\mathrm{ref}}(y^-|x)} \right] \right) \right], \tag{13}$$

where π_{ref} is the frozen reference policy, and each log-likelihood is computed as the sum of token log-probabilities over the corresponding continuation, $\log \pi(y|x) = \sum_t \log \pi\big(y_t \,\big|\, x,\, y_{<t}\big)$. Since both branches share the same prefix P_i, the preference comparison is conditioned on the identical reasoning state and directly supervises which continuation the model should prefer after reaching that state. In this way, LoCo learns a local continue-or-conclude preference: once the model has reached state P_i, it learns when to conclude with a shorter valid continuation instead of continuing with the original redundant reasoning.

5 Experiments

5.1 Experimental Setup

Data Construction and Training Configurations. The preference data are constructed from self-generated traces of DeepSeek-R1-Distill-Qwen-7B on the math-only training set used by Zhang et al. (2026b). For each query, we sample three rollouts with temperature 0.6 and top-p 0.95, and use the shortest correct rollout as the original trajectory. We then re-examine each flagged reasoning step with Qwen3-30B as the judge. For each detected step, we regenerate $k = 8$ continuations using the same temperature and top-p, and retain preference pairs with at least $\delta = 300$ tokens of savings, while limiting each question to at most $K = 3$ pairs. The prompt and continuation lengths are capped at $c_p = 6$k and $c_r = 12$k tokens, respectively.

We train DeepSeek-R1-Distill-Qwen-7B with LoCo for two epochs using a LoRA adapter. The adapter has rank 256, scaling factor $\alpha = 512$, and dropout 0.05 on the q/k/v/o attention projections, while the base model remains frozen. We use an effective batch size of 64 and a learning rate of 1×10^{-5}, with a cosine schedule and a warmup ratio of 0.1. Optimization uses AdamW with a weight decay of 0.01 and gradient clipping of 1.0. The DPO temperature is set to $\beta = 0.1$, with the base model serving as the reference policy. Training is conducted on four H20 GPUs with a maximum sequence length of 14,336 tokens.

Evaluation. We evaluate LoCo on mathematical reasoning and code generation. For mathematical reasoning, we use five benchmarks spanning different difficulty levels: MATH-500 (Lightman et al., 2024) with 500 problems, AMC 2023 with 40 problems, Olympiad (He et al., 2024) with 675 problems, AIME 2024 with 30 problems, and AIME 2025 with 30 problems. For code generation, we use MBPP (Austin et al., 2021) sanitized with 257 problems, LiveCodeBench (Jain et al., 2025) release v5 with 167 problems, and CodeContests (Li et al., 2022) with 165 problems. We sample eight completions per problem for all benchmarks.

Evaluation is performed with vLLM using the LoRA adapter and batched generation, with temperature 0.6 and top-p 0.95. We allow up to 32,768 newly generated tokens with a context length of 36,864, and generate until EOS without truncating at `</think>`. For mathematical reasoning, we extract the final answer from the last `\boxed{}` expression in the complete model output, and check it for symbolic equivalence using `math_verify`. For code generation, we extract the program from the last Python code block and evaluate it by sandboxed execution, following the official protocols for LiveCodeBench and MBPP and the AlphaCode judging protocol for CodeContests with a numerical tolerance of 10^{-5}. We report Accuracy, Tokens, and AUC following Zhang et al. (2026b), where AUC denotes the area under the accuracy versus token budget curve obtained by varying the maximum allowed reasoning length. All baselines (Guo et al., 2025; Shen et al., 2025; Zhang et al., 2026b) are evaluated from their released full weights without modification.

Table 1: Mathematical reasoning results across five benchmarks of increasing difficulty. The table reports absolute performance and base-relative changes under a shared evaluation protocol. LoCo achieves the best overall trade-off, combining the strongest compression with consistently comparable or improved accuracy across difficulty levels.

Method	MATH-500			AMC23			Olympiad		
	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)
Base	**91.88**±1.20	4015±177	82.64±1.12	90.31±2.48	6285±516	76.95±2.07	61.33±0.41	8972±130	51.28±0.37
DAST	91.38±0.79 (-0.50)	3720±119 (↓7.4%)	83.95±0.66 (+1.31)	88.12±4.17 (-2.19)	5981±304 (↓4.8%)	76.89±3.22 (-0.06)	59.91±0.73 (-1.42)	9155±313 (↑2.0%)	51.86±0.59 (+0.58)
TokenSqueeze	91.18±0.89 (-0.70)	2442±52 (↓39.2%)	85.92±0.73 (+3.28)	88.44±4.62 (-1.87)	4350±582 (↓30.8%)	79.42±4.01 (+2.47)	60.98±0.95 (-0.35)	6149±105 (↓31.5%)	54.29±0.78 (+3.01)
LoCo (Ours)	91.80±0.88 (-0.08)	**2243**±83 **(↓44.1%)**	**86.56**±0.76 **(+3.92)**	**91.56**±2.65 **(+1.25)**	**3663**±270 **(↓41.7%)**	**83.44**±2.12 **(+6.49)**	**62.78**±0.64 **(+1.45)**	**4968**±133 **(↓44.6%)**	**56.61**±0.61 **(+5.33)**

Method	AIME24			AIME25		
	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)
Base	50.83±6.61	13493±1367	40.33±5.00	37.92±5.62	15134±920	30.29±3.82
DAST	50.83±6.36 (±0.00)	14423±2075 (↑6.9%)	41.40±4.81 (+1.07)	37.92±4.34 (±0.00)	15652±1637 (↑3.4%)	31.66±3.79 (+1.37)
TokenSqueeze	48.75±4.69 (-2.08)	10457±831 (↓22.5%)	40.14±4.05 (-0.19)	35.83±4.96 (-2.09)	11055±585 (↓27.0%)	31.20±4.12 (+0.91)
LoCo (Ours)	**54.17**±4.27 **(+3.34)**	**7364**±399 **(↓45.4%)**	**46.82**±3.89 **(+6.49)**	**40.42**±3.75 **(+2.50)**	**8291**±250 **(↓45.2%)**	**34.55**±3.13 **(+4.26)**

5.2 Evaluation on Mathematical Reasoning Benchmarks

We first evaluate LoCo on mathematical reasoning benchmarks that span a wide difficulty range, from MATH-500 and AMC23 to Olympiad, AIME24, and AIME25. This setting tests whether the method can compress reasoning aggressively without sacrificing performance as problem difficulty increases.

Table 1 shows a consistent pattern across difficulty levels. On the easier benchmarks, LoCo already matches or improves upon the base model while removing 41.7%–44.1% of the generated tokens. More importantly, this advantage becomes even clearer on the harder benchmarks: on Olympiad, AIME24, and AIME25, LoCo simultaneously attains the highest accuracy, the highest AUC, and a 44.6%–45.4% reduction in completion length. This behavior distinguishes LoCo from both baselines. DAST provides only mild compression and can even increase completion length on harder benchmarks, while TokenSqueeze compresses aggressively but loses more accuracy as difficulty rises. Overall, the results indicate that LoCo preserves reasoning quality under substantial compression and remains particularly strong exactly where preserving difficult reasoning matters most.

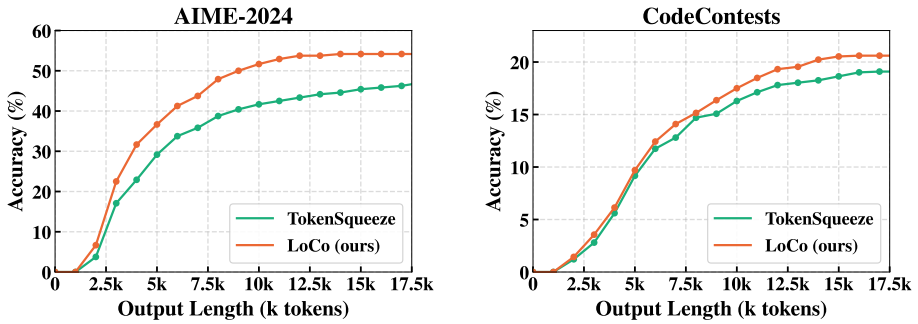
5.3 Evaluation on Coding Benchmarks

We next evaluate LoCo on code generation to test whether the learned compression behavior transfers beyond the math-only training queries described above. This provides a strict generalization setting. If the method still improves performance across coding benchmarks of different difficulty, the gain is more plausibly explained by a change in the model's reasoning preference, specifically when to continue and when to conclude, rather than by injecting additional domain knowledge.

Table 2 shows that this transfer is consistent across coding benchmarks of different difficulty. On MBPP, LoCo matches the base model in accuracy while reducing tokens by 36.3% and improving AUC by 1.53 points. On the more challenging LiveCodeBench and CodeContests benchmarks, it not only produces substantially shorter completions, but also improves both accuracy and AUC over all baselines. Since the training queries are mathematical rather than coding tasks, these gains

Table 2: Code generation results on three benchmarks of increasing difficulty, evaluated after training on a math-only query pool. The table reports absolute performance and base-relative changes under a shared evaluation protocol. LoCo retains the strongest overall advantage, showing that the learned compression preference generalizes beyond the training domain.

Method	MBPP			LiveCodeBench			CodeContests		
	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)
Base	**84.05**±0.91	3403±234	78.42±0.83	41.77±1.10	12319±352	32.86±0.87	18.48±2.17	14900±223	13.87±1.50
DAST	81.13±1.87 (-2.92)	4097±444 (↑20.4%)	76.32±1.68 (-2.10)	39.82±2.50 (-1.95)	14158±476 (↑14.9%)	31.88±1.66 (-0.98)	16.06±2.42 (-2.42)	17757±623 (↑19.2%)	12.61±1.79 (-1.26)
TokenSqueeze	81.37±2.18 (-2.68)	2196±131 (↓35.5%)	77.40±1.90 (-1.02)	41.69±2.22 (-0.08)	9670±200 (↓21.5%)	35.03±1.38 (+2.17)	19.09±0.97 (+0.61)	11677±305 (↓21.6%)	15.54±0.81 (+1.67)
LoCo (Ours)	**84.05**±1.94 (±0.00)	**2167**±119 **(↓36.3%)**	**79.95**±1.95 **(+1.53)**	**42.51**±2.81 **(+0.74)**	**9278**±204 **(↓24.7%)**	**35.64**±2.15 **(+2.78)**	**20.61**±1.07 **(+2.13)**	**11413**±243 **(↓23.4%)**	**16.78**±0.98 **(+2.91)**



(a) AIME24 **(b)** CodeContests

Figure 2: Accuracy versus token budget on AIME24 and CodeContests. LoCo achieves a better accuracy and efficiency trade-off than TokenSqueeze on both benchmarks.

support the interpretation that LoCo works by reshaping the model's preference over reasoning length, instead of relying on additional task-specific knowledge injected during training.

Figure 2 further illustrates the AUC gains in Tables 1 and 2. On both benchmarks, LoCo stays above TokenSqueeze over most token budgets, indicating a better overall accuracy–efficiency frontier rather than an advantage at a single cutoff.

5.4 Ablation Study

To understand which components are responsible for LoCo's gains, we keep the training setting fixed and vary one design choice at a time. Table 3 compares alternative preference sources and cutpoint strategies against the full method. As in the results table, we report absolute values for non-base variants together with their changes relative to the base model.

Local versus global preferences. In the *Global* ablation setting, we keep the same query set, pair-selection thresholds, and training setting as LoCo, but remove the state structure entirely. For each query, we form a trajectory-level preference pair between the shortest and longest correct full rollouts, so the prompt contains only the original query rather than a frozen intermediate state. Compared with this global construction, LoCo concentrates the supervision signal at the specific state where compression becomes appropriate. This difference matters most for compression. Although *Global* reaches similar accuracy on some benchmarks, it reduces tokens by only 5.2% on AMC23, 11.1% on AIME24, and 15.8% on AIME25, far below LoCo's 41.7%, 45.4%, and 45.2%. These results suggest that effective reasoning compression depends not only on preferring shorter reasoning, but also on identifying where compression should occur.

Table 3: Ablation results on three representative mathematical reasoning benchmarks under matched training settings. The table compares alternative preference sources and cutpoint choices against the full LoCo design. LoCo yields the strongest overall compression–accuracy trade-off, indicating that both local preference construction and detector-selected cutpoints are important.

Method	AMC23			AIME24			AIME25		
	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)	Acc (%)	Token	AUC (%)
Base	90.31±2.48	6285±516	76.95±2.07	50.83±6.61	13493±1367	40.33±5.00	37.92±5.62	15134±920	30.29±3.82
Preference source									
Global	90.94±2.65 (+0.63)	5957±562 (↓5.2%)	77.66±2.42 (+0.71)	**54.17**±5.56 **(+3.34)**	12002±870 (↓11.1%)	42.57±4.23 (+2.24)	**40.42**±5.47 **(+2.50)**	12740±1079 (↓15.8%)	32.46±3.90 (+2.17)
GPT-5.4 Cont.	90.62±2.59 (+0.31)	4083±261 (↓35.0%)	81.46±2.17 (+4.51)	52.92±6.53 (+2.09)	9204±823 (↓31.8%)	44.11±5.54 (+3.78)	39.58±6.53 (+1.66)	9690±709 (↓36.0%)	33.56±4.78 (+3.27)
Cutpoint design									
RandCut	88.44±2.97 (-1.87)	3679±243 (↓41.5%)	80.59±2.40 (+3.64)	48.33±5.35 (-2.50)	7880±642 (↓41.6%)	41.64±4.57 (+1.31)	38.75±5.62 (+0.83)	9151±968 (↓39.5%)	34.06±4.39 (+3.77)
UniCut	89.38±2.59 (-0.93)	**3573**±382 **(↓43.2%)**	81.17±2.81 (+4.22)	48.75±5.02 (-2.08)	7895±482 (↓41.5%)	42.58±4.08 (+2.25)	36.67±6.17 (-1.25)	8627±751 (↓43.0%)	31.76±4.73 (+1.47)
LoCo (Ours)	**91.56**±2.65 **(+1.25)**	3663±270 (↓41.7%)	**83.44**±2.12 **(+6.49)**	**54.17**±4.27 **(+3.34)**	**7364**±399 **(↓45.4%)**	**46.82**±3.89 **(+6.49)**	**40.42**±3.75 **(+2.50)**	**8291**±250 **(↓45.2%)**	**34.55**±3.13 **(+4.26)**

Self-generated versus external continuations. In the *GPT-5.4 Cont.* ablation setting, we keep the same cutpoints, frozen prefixes, rejected continuations, pair-selection thresholds, and training setting as LoCo, and change only the chosen continuation. GPT-5.4 regenerates from the same frozen prefix and replaces the response only when it is both correct and shorter. Compared with this external-continuation variant, LoCo performs better overall in compression and AUC, especially on the harder AIME benchmarks. A plausible explanation is distributional alignment. The preference pairs in LoCo are built from continuations generated by the same model being trained, so the supervision remains closer to the model's native reasoning distribution. In contrast, external continuations provide useful signals, but they are less matched to the model's own CoT style and therefore offer a weaker training signal for learning when to stop.

Detector-selected versus heuristic cutpoints. In the *RandCut* and *UniCut* ablations, we keep the regeneration process, pair-selection thresholds, and training configuration identical to LoCo, and only change how the cutpoints are selected. Each trace is assigned the same number of cutpoints as in LoCo. Specifically, *RandCut* selects cutpoints uniformly at random from the trajectory, while *UniCut* selects them at evenly spaced positions. The shorter continuations are then regenerated and paired following the same procedure as LoCo. This comparison isolates the effect of cutpoint selection and evaluates whether identifying appropriate compression points is important for effective reasoning compression. Both variants can still shorten CoT substantially, but they incur larger accuracy losses because they often compress reasoning at positions where the model still needs additional computation. This effect is especially clear on AIME24 and AIME25, where LoCo preserves the strongest accuracy while matching or exceeding the heuristic variants in overall AUC. These comparisons show that LoCo's advantage comes not only from preferring shorter continuations, but from learning that preference at the right reasoning states.

6 Conclusion

We introduced LoCo, a local preference learning method for reasoning compression that constructs supervision directly at reasoning states where overthinking occurs. Instead of treating compression as a global preference over complete trajectories, LoCo freezes the current reasoning prefix, regenerates verified shorter continuations from the same state, and learns state-conditioned continue-or-conclude preferences with DPO. Empirically, LoCo shows consistent gains across both mathematical reasoning and code generation. The cross-domain improvements suggest that LoCo primarily reshapes the model's preference over reasoning length rather than adding task-specific knowledge. This indicates that effective reasoning compression requires learning when further reasoning is unnecessary at the appropriate reasoning states, rather than simply favoring shorter trajectories.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic data sets, cleaning and reformatting datasets, and assisting with translation. We have not used generative AI tools for helping develop theoretical models or conceptual frameworks, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, implementing methods, supporting qualitative and thematic data analysis, interpreting results, and formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools for editing the paper to improve readability. We have reviewed all AI-assisted work. Generating and cleaning data using large language models is a primary method of our study design, for which we conducted detailed experiments, and we checked all AI-edited text. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the methodological and experimental details required to reproduce our results in the main text and appendices. Section 4 describes the LoCo method, including local overthink detection, verified continuation generation, and local preference learning. Section 5 specifies the training setup, evaluation protocol, benchmark composition, decoding settings, and ablation design used in our experiments. Appendix A provides additional implementation details, while Sections A.2 and A.3 specify the evaluation prompts and the full construction of the ablation variants.

REFERENCES

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning. *arXiv preprint arXiv:2503.04697*, 2025.

Daman Arora and Andrea Zanette. Training language models to reason efficiently. *Advances in Neural Information Processing Systems*, 38:60770–60808, 2026.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, et al. Do not think that much for 2+ 3=? on the overthinking of o1-like llms. *arXiv preprint arXiv:2412.21187*, 2024.

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.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Chenzhi Hu, Qinzhe Hu, Yuhang Xu, Junyi Chen, Ruijie Wang, Shengzhong Liu, Jianxin Li, Fan Wu, and Guihai Chen. Smartthinker: Progressive chain-of-thought length calibration for efficient large language model reasoning. *arXiv preprint arXiv:2603.08000*, 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.

Joonwon Jang, Jaehee Kim, Wonbin Kweon, Seonghyeon Lee, and Hwanjo Yu. Verbosity-aware rationale reduction: Effective reduction of redundant rationale via principled criteria. *arXiv preprint arXiv:2412.21006*, 2024.

Zhensheng Jin, Xinze Li, Yifan Ji, Chunyi Peng, Zhenghao Liu, Qi Shi, Yukun Yan, Shuo Wang, Furong Peng, and Ge Yu. Recut: Balancing reasoning length and accuracy in llms via stepwise trails and preference optimization. In *EMNLP (Findings)*, pp. 14269–14282, 2025.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv preprint arXiv:2406.18629*, 2024.

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.

Yulin Li, Tengyao Tu, Li Ding, Junjie Wang, Huiling Zhen, Yixin Chen, Yong Li, and Zhuotao Tian. Efficient reasoning with balanced thinking. *arXiv preprint arXiv:2603.12372*, 2026.

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.

Haotian Luo, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Xiaochun Cao, Dacheng Tao, Naiqiang Tan, and Li Shen. O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 14242–14257, 2026.

Xinyin Ma, Guangnian Wan, Runpeng Yu, Gongfan Fang, and Xinchao Wang. Cot-valve: Length-compressible chain-of-thought tuning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6025–6035, 2025.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.

Keqin Peng, Liang Ding, Yuanxin Ouyang, Meng Fang, and Dacheng Tao. Revisiting overthinking in long chain-of-thought from the perspective of self-doubt. *arXiv preprint arXiv:2505.23480*, 2025.

Yi Shen, Jian Zhang, Jieyun Huang, Shuming Shi, Wenjing Zhang, Jiangze Yan, Ning Wang, Kai Wang, Zhaoxiang Liu, and Shiguo Lian. Dast: Difficulty-adaptive slow-thinking for large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 2322–2331, 2025.

Feifan Song, Shaohang Wei, Bofei Gao, Yejie Wang, Wen Luo, Wei Li, Linli Yao, Weimin Xiong, Liang Chen, Tianyu Liu, et al. Mitigating overthinking through reasoning shaping. *arXiv preprint arXiv:2510.09535*, 2025.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Na Zou, et al. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419*, 2025.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. Tokenskip: Controllable chain-of-thought compression in llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 3351–3363, 2025.

Liyan Xu, Mo Yu, Fandong Meng, and Jie Zhou. How far ahead do llms plan? uncovering the latent horizon in chain-of-thought reasoning. *arXiv preprint arXiv:2602.02103*, 2026.

Yuyang Xu, Yi Cheng, Haochao Ying, Zhuoyun Du, Renjun Hu, Xing Shi, Wei Lin, and Jian Wu. Sspo: Self-traced step-wise preference optimization for process supervision and reasoning compression. *arXiv preprint arXiv:2508.12604*, 2025.

Xinliang Frederick Zhang, Anhad Mohananey, Alexandra Chronopoulou, Pinelopi Papalampidi, Somit Gupta, Tsendsuren Munkhdalai, Lu Wang, and Shyam Upadhyay. Do llms really need 10+ thoughts for "find the time 1000 days later"? towards structural understanding of llm overthinking. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17005–17030, 2026a.

Yuxiang Zhang, Zhengxu Yu, Weihang Pan, Zhongming Jin, Qiang Fu, Deng Cai, Binbin Lin, and Jieping Ye. Tokensqueeze: Performance-preserving compression for reasoning llms. *Advances in Neural Information Processing Systems*, 38:91516–91539, 2026b.

A Implementation Details

A.1 Overthink Detection

Source traces. We generate self-sampled reasoning traces from DeepSeek-R1-Distill-Qwen-7B on the math14k training pool, the math-only training set of Zhang et al. (2026b) containing 14,027 questions. We sample three rollouts per question at temperature 0.6 and top-p 0.95, and retain the shortest correct rollout as the source trajectory for each question. Detection, judging, and pair construction all operate on this trajectory, so each question contributes at most one trace. A total of 6,182 questions yield at least one surviving pair.

Confidence screening. The complete CoT is fed back to the model as a prompt, and we record the top-1 token probability $p_\tau^{\max}$ at every position via batched offline vLLM prompt logprobs. Steps are delimited by paragraph breaks. Step confidence c_i is the arithmetic mean of $p_\tau^{\max}$ over the step's token span, and volatility is $v_i = (c_i - c_{i-1})^2/4$ as in Section 4.2. The four quantiles $(q_{25c}, q_{75c}, q_{25v}, q_{75v})$ are calibrated once on the model's self-generated trace corpus. For DeepSeek-R1-Distill-Qwen-7B, they are 0.666, 0.928, 0.00049, and 0.0099, respectively.

Judge re-checking. Every confidence-screened Overthink step is re-checked by Qwen3-30B using vLLM. The judge sees the problem, the preceding steps, the candidate step, and the predicted label, and returns a corrected label plus a short rationale in JSON. The template is:

```
You are evaluating whether an existing overthink label
for one reasoning step is correct.

Problem:
{question}

Previous reasoning steps:
{previous_steps}

Current reasoning step:
{current_step}

Predicted label:
{predicted_label}

Label definitions:
- OVERTHINK: this step is wasteful or redundant;
  removing it would not break the path to the answer.
- NORMAL: this step is necessary reasoning;
  removing it would break or weaken the path.
- FINAL_ANSWER_WRAPUP: this step mainly wraps up
  or restates the final answer.

Respond in this exact JSON format and nothing else:
{
  "gold_label": "OVERTHINK|NORMAL|FINAL_ANSWER_WRAPUP",
```

```
  "reason": "short reason"
}
```

Only judge-confirmed OVERTHINK steps survive. A trace with many flagged steps could otherwise dominate regeneration, so each trace keeps at most 10 candidate states, selected by uniform sampling. Across the corpus this leaves 64,087 candidate states.

A.2 Evaluation Detail

Evaluation prompts. Each benchmark appends a short instruction to the raw problem text, and the result is wrapped by each model's native chat template. The five mathematical benchmarks share the instruction

```
Please reason step by step, and put your final answer within
\boxed{}.
```

The three code benchmarks share the following instruction:

```
Please write a solution in Python.  Wrap your final code in
a ```python code block.
```

Following their official protocols, MBPP additionally includes the public tests in the prompt (`Your code should pass these tests:  tests`), while LiveCodeBench appends the provided starter code when applicable. We extract the last Python code block from the complete output, which naturally excludes any code blocks generated within the thinking process, and execute it in a sandbox with per-problem timeouts of 10 seconds for MBPP and 6 seconds for LiveCodeBench. CodeContests follows the AlphaCode judging protocol, where outputs are split by whitespace and compared token-wise case-insensitively, with exact matching for integers and a tolerance of 10^{-5} for floating-point values.

A.3 Ablation Construction

This subsection provides the full descriptions of the ablation variants introduced in the main text. All variants use the same question set, pair-selection thresholds, and training recipe as LoCo. Each variant modifies only one construction choice, while differences in pair counts arise naturally from the resulting construction.

Global. The Global variant pairs the shortest and longest correct rollouts of the same question. To avoid disadvantaging this variant, its pool is expanded from three source rollouts to eight per question, allowing the selected responses to draw from a larger sample of short solutions than the source traces provide. The same savings threshold and top-K selection as LoCo are applied, and the prompt contains only the question, without a frozen state.

GPT-5.4 Cont. GPT-5.4 continues from the same frozen prefix, provided as an assistant prefill with a system instruction to continue the reasoning in the same first-person style without restarting the problem. The continuation is generated directly from the frozen prefix and replaces the self-generated chosen continuation only when it is correct and shorter than the rejected continuation. Thus, this variant differs from LoCo only on the chosen side.

RandCut and UniCut. Each trace receives the same number of cutpoints as selected by the detector. Cutpoints are sampled from all eligible steps after excluding the first step and the final two steps. RandCut samples uniformly at random. UniCut places the cutpoints on an evenly spaced grid over the trace and snaps each to the nearest eligible step in its segment. Downstream regeneration and pair construction remain unchanged. The detector's cutpoints concentrate late in the trace, with a mean relative position of 0.57 and about half occurring in the final 40% of steps. In contrast, RandCut and UniCut cover the trace more uniformly, with mean relative positions of 0.47 and 0.50, respectively. Thus, the heuristic variants tend to compress at earlier, less converged states.

B QUALITATIVE EXAMPLES

We present preference triples from the training data in full. Each triple contains the frozen prefix P, the chosen continuation y^+, which is the shortest verified regeneration from the same state, and the rejected original continuation y^-, reproduced verbatim from the stored pairs. The model's LaTeX and Markdown are rendered as-is, with only leading and trailing whitespace trimmed. Each continuation ends its reasoning block with the closing `</think>` tag, followed by the final answer block outside the reasoning region.

Example 1 **Question:** If July 3, 2030 is a Wednesday, what day of the week is July 14, 2030?

The detector flags the state 28 steps into the original trace. By this point, the answer (Sunday) has already been derived arithmetically, confirmed by a day-by-day listing, and re-confirmed using modular arithmetic. The preference pair contrasts the model's original continuation from this state, which is rejected, with the shortest verified regeneration, which is chosen.

Frozen prefix P (the first 28 steps).

> Okay, so I need to figure out what day of the week July 14, 2030 is, given that July 3, 2030 is a Wednesday. Hmm, let's see. I remember that each week has 7 days, so the days of the week repeat every 7 days. That means if I can find out how many days are between July 3 and July 14, I can add that number of days to Wednesday and find the corresponding day.
>
> First, let me count the number of days between July 3 and July 14. Let's see, July has 31 days, right? So from July 3 to July 14, how many days is that? Hmm, if I subtract 3 from 14, that would give me 11 days, but wait, does that include both the start and end dates? I think not. So if I start counting from July 3 as day 0, then July 4 is day 1, and so on. So to get from July 3 to July 14, I need to count 11 days. Let me verify that: July 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14. Yeah, that's 11 days. So the difference is 11 days.
>
> Wait, but sometimes when calculating the number of days between two dates, you have to be careful whether to include both the start and end dates. Let me think. If I have two consecutive days, like July 3 and July 4, that's 1 day difference, right? So subtracting 3 from 4 gives 1, which is correct. So in this case, July 3 to July 14 is 11 days. So 14 minus 3 is 11, so that's correct.
>
> Now, since each week is 7 days, I can figure out how many weeks and extra days are in 11 days. So, 11 divided by 7 is 1 week and 4 days. So 11 days is equal to 1 week plus 4 days. That means that 11 days from a Wednesday would be Wednesday plus 4 days. Let me count: Wednesday plus 1 day is Thursday, plus 2 is Friday, plus 3 is Saturday, plus 4 is Sunday. So, is July 14, 2030 a Sunday?
>
> Wait, let me double-check that. Sometimes when I'm adding days, I can get confused. Maybe I should list out the dates and their corresponding days.
>
> Starting from July 3, which is Wednesday. Let's list each day:
>
> July 3: Wednesday
>
> July 4: Thursday
>
> July 5: Friday
>
> July 6: Saturday
>
> July 7: Sunday
>
> July 8: Monday
>
> July 9: Tuesday
>
> July 10: Wednesday
>
> July 11: Thursday
>
> July 12: Friday
>
> July 13: Saturday
>
> July 14: Sunday
>
> Wait, so July 14 is indeed a Sunday. Hmm, okay, so my initial calculation was correct. 11 days from Wednesday is Sunday.
>
> But hold on, sometimes people might get confused whether to add 11 days or 12 days because of the way we count dates. Let me see: If July 3 is day 0, then July 14 is day 11. So adding 11 days to Wednesday. 7 days later is the same day of the week, which would be Wednesday again, and then 4 more days would bring it to Sunday.

Alternatively, another way to think about it is using modular arithmetic. Since the days cycle every 7 days, the day of the week repeats every 7 days. So, if I have a certain number of days, I can find the remainder when divided by 7, and that will tell me how many days ahead it is from the starting day.

So, 11 divided by 7 is 1 with a remainder of 4. So, the day of the week would be Wednesday plus 4 days, which is Sunday. So that matches my previous result.

But just to be thorough, let me check another method. Maybe using the concept of odd days. Each month, the number of days can affect the day of the week for the next month. Wait, but in this case, we're just dealing with the same month, July. So maybe that's complicating things. Alternatively, I can use the concept of counting weeks and extra days.

Wait, another way: If July 3 is Wednesday, then July 10 would also be a Wednesday because that's exactly 7 days later. Then, July 10 is Wednesday, so July 11 is Thursday, July 12 is Friday, July 13 is Saturday, July 14 is Sunday. So that also gets me to Sunday. Yeah, that seems consistent.

Alternatively, I could use a reference point. I know that July 4, 2023, is a Monday. Wait, no, that's not relevant. Maybe I can find a known date in 2030. Wait, 2030 is not a leap year, so February has 28 days. But that might not help directly with July.

Alternatively, maybe I can use Zeller's Congruence or some algorithm to calculate the day of the week for July 14, 2030, given that July 3 is a Wednesday. But that might be overcomplicating, since I already have two methods confirming it's Sunday.

Wait, let me also just consider the number of weeks and days. From July 3 to July 14 is 11 days. 11 divided by 7 is 1 week and 4 days. So, starting from Wednesday, adding 4 days: Wednesday + 4 days = Sunday.

Alternatively, if I consider that 7 days later is Wednesday, then adding another 4 days: Thursday, Friday, Saturday, Sunday. So that also gives me Sunday.

Chosen y^+ (212 tokens).

So, multiple methods confirm that July 14, 2030 is a Sunday.

I think I'm confident now that the answer is Sunday.

Final Answer $\boxed{Sunday}$ </think>

Given that July 3, 2030 is a Wednesday, we need to determine the day of the week for July 14, 2030.

First, we calculate the number of days between July 3 and July 14. This is 11 days.

Next, we determine how many weeks and extra days are in 11 days. Dividing 11 by 7 gives 1 week and 4 extra days.

Starting from Wednesday, we add 4 days: - Wednesday + 1 day = Thursday - Thursday + 1 day = Friday - Friday + 1 day = Saturday - Saturday + 1 day = Sunday

Thus, July 14, 2030 is a Sunday.

$$\boxed{Sunday}$$

Rejected y^- (986 tokens, pair gap 774).

Wait, another way: Maybe using the concept of reference dates. For example, I know that January 1, 2030, was a Wednesday. Wait, no, actually, let me check. Wait, no, I don't know that off the top of my head. Maybe it's better not to go that route.

Alternatively, I can use the fact that each year, the same date moves one day forward in the week, unless it's a leap year, in which case it moves two days. Wait, 2030 is not a leap year, so the next year, 2031, same date would be one day later in the week. But that might not help here.

Wait, perhaps I can count the number of days between July 3, 2030, and July 14, 2030, and then see how many weeks and extra days that is.

Wait, so July 3 to July 14 is 11 days, as established earlier. So 11 days later. 11 divided by 7 is 1 week and 4 days. So, Wednesday plus 4 days is Sunday.

Alternatively, maybe using the concept of counting weeks: 7 days is one week, so 14 days would be two weeks, which brings us back to Wednesday. But 11 days is less than 14, so it's 1 week and 4 days.

Alternatively, perhaps thinking in terms of modulo 7 arithmetic. If I assign numbers to days: Sunday=0, Monday=1, Tuesday=2, Wednesday=3, Thursday=4, Friday=5, Saturday=6. Then, adding 11 days to Wednesday (which is 3) would be 3 + 11 = 14. Then, 14 mod 7 is 0, which is Sunday. So that also works.

Wait, so 3 (Wednesday) + 11 days = 14, which is 0 mod 7, which is Sunday. So that confirms it as well.

Alternatively, if I use the formula for the day of the week: (start_day + number_of_days) mod 7. So, if start_day is Wednesday, which is 3, number_of_days is 11, so (3 + 11) mod 7 = 14 mod 7 = 0, which is Sunday.

Alternatively, if I use another numbering system where Sunday is 1, Monday is 2, etc., but that might complicate things. The key is that modulo 7 will give the correct offset.

So, given that, I think I'm confident that July 14, 2030 is a Sunday.

But just to be absolutely sure, let me recount the days manually:

Starting from July 3, which is Wednesday.

July 3: Wed

July 4: Thu

July 5: Fri

July 6: Sat

July 7: Sun

July 8: Mon

July 9: Tue

July 10: Wed

July 11: Thu

July 12: Fri

July 13: Sat

July 14: Sun

Yes, so each week, it cycles back. So from July 3 to July 10 is exactly one week, which is 7 days, so that brings us back to Wednesday. Then, July 11 is Thursday, 12 is Friday, 13 is Saturday, and 14 is Sunday. So that's four days after Wednesday, which is Sunday.

Therefore, I can conclude that July 14, 2030 is a Sunday.

Final Answer $\boxed{Sunday}$ </think>

Given that July 3, 2030 is a Wednesday, we need to determine the day of the week for July 14, 2030.

First, we calculate the number of days between July 3 and July 14, 2030. This is 11 days (from July 3 to July 14 inclusive).

Next, we determine how many weeks and extra days are in 11 days: - 11 days is equivalent to 1 week (7 days) and 4 extra days.

Starting from Wednesday, we add 4 days: - Wednesday + 1 day = Thursday - Thursday + 1 day = Friday - Friday + 1 day = Saturday - Saturday + 1 day = Sunday

Thus, 11 days after Wednesday is Sunday.

Therefore, July 14, 2030 is a Sunday.

$$\boxed{Sunday}$$

The rejected suffix adds no new information once the answer has been derived and confirmed. It restarts from a reference-date idea, abandons it, re-derives the offset through counting, modulo-7 arithmetic, and a day-of-week formula, and finally recounts the calendar by hand before restating Sunday. At this state, LoCo learns a simple preference to conclude rather than continue.

Example 2 **Question:** What fraction of $\frac{1}{2}$ is $\frac{1}{7}$?

The detector flags the state 11 steps into the trace, after the equation $x \cdot \frac{1}{2} = \frac{1}{7}$ has been set up, solved, and verified twice.

Frozen prefix P (the first 11 steps).

Alright, so I have this math problem here: "What fraction of 1/2 is 1/7?" Hmm, okay. I need to figure out what fraction, when multiplied by 1/2, gives me 1/7. Let me think about how to approach this.

First, let's translate the question into an equation. If I let the unknown fraction be x, then the problem is asking: x times 1/2 equals 1/7. So, mathematically, that would be:

x * (1/2) = 1/7

Now, to solve for x, I need to isolate it. Since it's multiplied by 1/2, I can do the opposite operation, which is multiplication by the reciprocal. The reciprocal of 1/2 is 2/1, or just 2. So, if I multiply both sides of the equation by 2, I should get:

x = (1/7) * 2

Let me compute that. 1/7 multiplied by 2 is 2/7. So, x equals 2/7. That means 2/7 of 1/2 is 1/7.

Wait, let me double-check that. If I take 2/7 and multiply it by 1/2, does that give me 1/7?

Calculating that: (2/7) * (1/2) = (2*1)/(7*2) = 2/14 = 1/7. Yep, that checks out.

But just to make sure I didn't make a mistake in setting up the equation, let me rephrase the question. It's asking, "What fraction of 1/2 is 1/7?" So, it's like saying 1/7 is what portion of 1/2? That portion is x.

Another way to think about it is: if 1/2 is the whole, then 1/7 is some fraction of that whole. So, x = (1/7) / (1/2). Dividing by a fraction is the same as multiplying by its reciprocal, so x = (1/7) * (2/1) = 2/7.

Yes, that makes sense. So, the fraction is 2/7.

Chosen y^+ (296 tokens).

I think that's the correct answer.

Just to recap, the steps are:

1. Set up the equation: x * (1/2) = 1/7. 2. Solve for x by multiplying both sides by 2: x = (1/7) * 2 = 2/7. 3. Verify by plugging back in: (2/7) * (1/2) = 1/7.

Everything seems to add up, so I'm confident the answer is 2/7.

Final Answer The fraction is $\boxed{\frac{2}{7}}$. </think>

To determine what fraction of $\frac{1}{2}$ is $\frac{1}{7}$, we set up the equation:

$$x \times \frac{1}{2} = \frac{1}{7}$$

To solve for x, we multiply both sides by the reciprocal of $\frac{1}{2}$, which is 2:

$$x = \frac{1}{7} \times 2$$

This simplifies to:

$$x = \frac{2}{7}$$

We verify this by checking that $\frac{2}{7} \times \frac{1}{2} = \frac{1}{7}$, which is correct.

Thus, the fraction is $\boxed{\frac{2}{7}}$.

Rejected y^- (1,247 tokens, pair gap 951).

Wait, let me try another approach just to be thorough. Let's use fractions in words. If I have half of something, and I want to find out how much 1/7 is of that half, I can set up a proportion.

Let me denote the unknown fraction as x. So, x of 1/2 is 1/7. So, x * (1/2) = 1/7. Solving for x, as before, gives me x = (1/7) / (1/2) = (1/7) * (2/1) = 2/7.

Alternatively, if I visualize this, imagine a pizza cut into 2 equal slices. Each slice is 1/2 of the pizza. Now, if I take 1/7 of a slice, how much of the whole pizza is that? It would be 1/7 of 1/2, which is 1/14 of the whole pizza. But wait, that seems different.

Wait, hold on. If I have 1/2 of a pizza, and I take 1/7 of that, it's 1/7 * 1/2 = 1/14 of the whole pizza. But the question isn't asking for that. The question is, what fraction of the 1/2 is the 1/7. So, actually, it's like, if I have 1/2, and 1/7 is some part of that 1/2. So, it's like, 1/7 divided by 1/2, which is 2/7, as we had before.

I think my confusion came from misinterpreting the visualization. So, the key is, it's not 1/7 of the whole pizza, but rather 1/7 is what fraction of the half. So, it's 1/7 divided by 1/2, which is 2/7. So, 1/7 is 2/7 of 1/2.

Another way to think about it is using ratios. If I have two numbers, a and b, and I want to find what fraction a is of b, it's a/b. So, in this case, a is 1/7 and b is 1/2. So, the fraction is (1/7)/(1/2) = 2/7.

Let me think of another way to confirm. If I have 1/2, and I want to know how many times 1/7 fits into it, it's like dividing 1/2 by 1/7. So, that would be (1/2) / (1/7) = 7/2 = 3.5. But that's not directly helpful because the question isn't asking how many times 1/7 fits into 1/2, but rather what fraction of 1/2 is 1/7.

Wait, so if I have 1/2 and 1/7 is a fraction of that, then it's 1/7 divided by 1/2, which is 2/7. So, 1/7 is 2/7 of 1/2.

Alternatively, using decimal equivalents might help. Let me convert the fractions to decimals.

1/2 is 0.5, and 1/7 is approximately 0.142857. So, what fraction of 0.5 is 0.142857? So, 0.142857 divided by 0.5 is approximately 0.285714, which is roughly 2/7, since 2 divided by 7 is approximately 0.285714. So, that aligns with our earlier result.

Okay, so all these methods point to the same answer: 2/7.

Just to recap, the steps are:

1. Let the unknown fraction be x. 2. Set up the equation: x * (1/2) = 1/7. 3. Solve for x by multiplying both sides by the reciprocal of 1/2, which is 2. 4. x = (1/7) * 2 = 2/7. 5. Verify by plugging back into the original equation: (2/7) * (1/2) = 1/7, which is correct. 6. Cross-checked using different methods: visualizing, ratios, decimal conversion—all confirm that 2/7 is the correct answer.

So, after going through multiple approaches and verifying each step, I feel confident that the fraction we're looking for is 2/7.

Final Answer The fraction is $\boxed{\frac{2}{7}}$. </think>

To determine what fraction of $\frac{1}{2}$ is $\frac{1}{7}$, we set up the equation where x is the unknown fraction:

$$x \times \frac{1}{2} = \frac{1}{7}$$

To solve for x, we multiply both sides by the reciprocal of $\frac{1}{2}$, which is 2:

$$x = \frac{1}{7} \times 2$$

This simplifies to:

$$x = \frac{2}{7}$$

We verify this by checking that $\frac{2}{7}$ multiplied by $\frac{1}{2}$ equals $\frac{1}{7}$:

$$\left(\frac{2}{7}\right) \times \left(\frac{1}{2}\right) = \frac{2 \times 1}{7 \times 2} = \frac{2}{14} = \frac{1}{7}$$

Thus, the fraction is $\boxed{\frac{2}{7}}$.

The rejected suffix re-derives the same result through a proportion, a pizza-slice visualization with a detour caused by a misinterpretation, a ratio argument, a dead-end attempt to count how many times the quantity fits, and a decimal-conversion check. It then closes with a step-by-step recap, adding four redundant derivations after the answer had already been verified in the prefix. LoCo anchors the preference at the state where the existing solution is already sufficient.