CoTDog: Closed-Loop Control for Efficient LLM Reasoning

Anonymous authors
Paper under double-blind review

Abstract

Chain-of-thought (CoT) reasoning enables large language models (LLMs) to solve increasingly complex problems by allocating more computation to intermediate reasoning. However, this additional computation is not always used effectively, as reasoning models may spend excessive tokens on intermediate steps or repeatedly revisit already explored directions. In this paper, we introduce CoTDog, a closed-loop framework that guides reasoning progression and answer production within a single generation process. An external controller monitors visible trace events and resource use while maintaining persistent execution state. It guides the model to control its reasoning pace, move to the next stage when needed, and produce the final answer at the right time. We evaluate CoTDog on three open-weight reasoning models (i.e., QwQ-32B, Qwen3.5-27B, and DeepSeek-R1-Distill-Qwen-32B) across MATH-500, GPQA Diamond, and LiveCodeBench. Across the nine settings, CoTDog achieves 7.9% higher task scores and uses 23.7% fewer completion tokens, averaging the relative differences against the strongest baseline in each setting. On Qwen3.5-27B with LiveCodeBench, it improves pass@1 by 18.8 percentage points over the strongest baseline while using 22.1% fewer tokens. The results show that appropriate guidance during CoT reasoning can substantially improve reasoning performance without requiring model parameter updates.

1 Introduction

Chain-of-Thought (CoT) reasoning has become a widely adopted approach for improving the reasoning capability of large language models (LLMs). CoT allows models to produce intermediate steps and use more computation during inference. This approach improves performance on multi-step tasks such as mathematical problem solving and code generation (Wei et al., 2022; Wang et al., 2023; Guo et al., 2025). However, current reasoning models still face several challenges in their reasoning processes. For example, models may *overthink* simple problems by continuing to reason after finding a correct solution (Chen et al., 2025; Zhang et al., 2026), while *underthinking* harder problems by giving up on a promising approach prematurely (Wang et al., 2025). Longer traces often repeat the same checks, revisit earlier steps, or cycle through similar ideas (Chen et al., 2025; Duan et al., 2026). These patterns suggest that current reasoning LLMs can produce detailed reasoning but still struggle to adjust their computation during reasoning.

To address these challenges, existing approaches to improving reasoning efficiency fall into two main categories. The first controls how much computation is used through token budgets or adaptive reasoning strategies, e.g., budget-based methods such as s1, TALE, and Plan-and-Budget (Muennighoff et al., 2025; Han et al., 2025; Lin et al., 2026). However, these methods either set budgets before reasoning begins or adjust reasoning based on intermediate outputs using fixed rules, which limits their ability to respond to changing needs during reasoning. The second focuses on early stopping, such as Answer Convergence, DEER, and LYNX, which aim to identify when a model can end reasoning and produce an answer (Liu & Wang, 2025; Yang et al., 2026; Akgül et al., 2025). However, deciding when to stop does not address how to guide reasoning that is still incomplete.

These limitations suggest that efficient reasoning requires not only deciding how much computation to allocate, but also how computation should evolve during reasoning. Our key insight is that an evolving reasoning trace can be a source of feedback for controlling how reasoning proceeds. Thus, CoTDog introduces a closed-loop mechanism that continuously monitors the generated rea-

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.

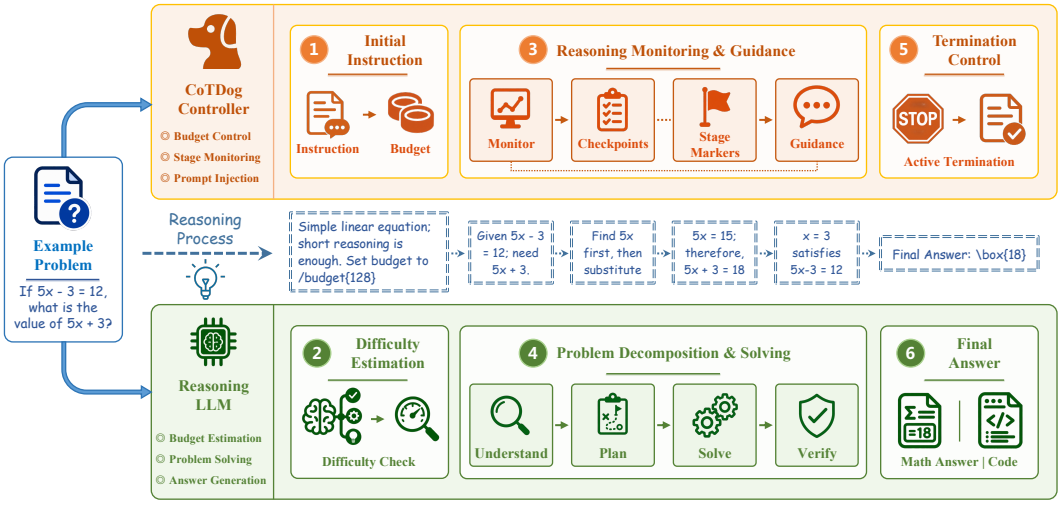


Figure 1: Overview of the CoTDog framework.

soning trace during generation and applies targeted control when needed. This enables adaptive control under a constrained computation budget without modifying model parameters or relying on an additional reasoning model.

We evaluate CoTDog on MATH-500 (Lightman et al., 2023), GPQA Diamond (Rein et al., 2023), and LiveCodeBench (Jain et al., 2024) using three open-weight reasoning models (i.e., QwQ-32B (Qwen Team, 2025), Qwen3.5-27B (Qwen Team, 2026), and DeepSeek-R1-Distill-Qwen-32B (Guo et al., 2025)). CoTDog achieves higher task scores with fewer completion tokens than both Vanilla and Structured Prompt in all nine combinations of models and benchmarks, reducing token use by 15.0% to 63.3% relative to Vanilla. It also achieves higher scores with fewer tokens than Plan-and-Budget and TALE-EP in most settings. For example, on Qwen3.5-27B with LiveCodeBench, CoTDog raises $Pass@1$ from Plan-and-Budget's 57.2% to 76.0% while reducing the average token count by 22.1%.

Our main contributions are:

- We propose CoTDog, a closed-loop method that guides LLM reasoning during generation. It follows the reasoning process, remembers earlier events, and skips guidance that is no longer needed.
- We build a controller that helps the model manage its reasoning, move forward when needed, and give the final answer at the right time. It does not change the model parameters or require another reasoning model.
- We test CoTDog on three reasoning models and three benchmarks. It achieves higher task scores than all baselines in all nine settings. Across the nine settings, CoTDog achieves 7.9% higher task scores and uses 23.7% fewer completion tokens, averaging the relative differences against the highest-scoring baseline in each setting. Our case studies show that CoTDog can keep a correct solution, finish missing steps, and stop repeated reasoning.

2 Related Work

Existing approaches to improving reasoning efficiency at inference time can be broadly grouped into two categories: *reasoning process control*, which adjusts how much computation the model uses and how it reasons, and *early stopping*, which determines when to stop reasoning and produce an answer.

Reasoning process control. This line of work improves efficiency by limiting the computational cost of reasoning or adjusting how the model reasons during generation. Methods such as s1 (Muennighoff et al., 2025), TALE (Han et al., 2025), and Plan-and-Budget (Lin et al., 2026) control token

usage by setting a token budget for reasoning or assigning budgets to individual questions and sub-problems. These methods control the amount of computation available for reasoning, but token budgets alone do not specify when and how to guide the model from its current reasoning stage to the next. Other methods guide reasoning based on the model's progress so far. CGRS (Huang et al., 2026) adjusts how much the model reflects on its reasoning based on its certainty, DyCon (Tu et al., 2026) adjusts how extensively it reasons based on estimated problem difficulty, and ReBalance (Li et al., 2026) uses changes in confidence to reduce repetition while allowing further exploration. Meta-Reasoner (Sui et al., 2026) assesses the model's progress to choose strategies such as revisiting earlier steps or trying a different approach. CoTDog focuses on timely transitions between reasoning stages: it guides the model to advance to the next stage before excessive deliberation in the current stage leaves insufficient computation to complete the remaining reasoning. Reducing repetitive loops is a secondary benefit rather than its primary objective.

Early stopping. The second category reduces reasoning cost by identifying when the model can produce an adequate answer without completing its full reasoning process. Answer Convergence (Liu & Wang, 2025) examines whether answers produced during reasoning agree with one another as a basis for deciding when to stop, while DEER (Yang et al., 2026) evaluates confidence in trial answers at points where the model could move from reasoning to answering. LYNX (Akgül et al., 2025) trains a small probe that reads the model's hidden states to help decide whether to stop based on confidence. These methods focus on whether to continue reasoning or produce an answer. This stopping decision, however, does not address how to advance unfinished reasoning when the model spends too much computation in a single stage. Rather than ending reasoning early, CoTDog guides timely transitions to subsequent stages so that the model can complete its reasoning within the available budget. The distinction is therefore between deciding *when to stop reasoning* and controlling *when to advance to the next reasoning stage*.

3 CoTDog: Closed-Loop Reasoning Control

3.1 Problem Formulation

CoT reasoning typically moves through several stages before reaching the final answer, and the generated trace can reveal the current stage and its progress. Standard CoT generation does not use these signals to adjust its behavior. We therefore treat CoT generation as a controllable multi-stage process and use an external controller to regulate it under a fixed output budget.

Given an input problem x, a language model with fixed parameters θ defines the next-token distribution at decoding step t as

$$p_{\theta,t}(v) = p_\theta(v \mid x, Y_{<t}), \qquad v \in \mathcal{V}, \tag{1}$$

where $\mathcal{V}$ is the vocabulary and $Y_{<t}$ is the generated prefix. Let $Y = (Y_1, \ldots, Y_T)$ denote the complete output under a total budget B. This budget covers the reasoning trace, control content, and final answer. Thus, $T \leq B$, and the remaining budget before generating Y_t is

$$r_t = B - (t-1).$$

The controller may also assign local reasoning budgets to specific actions. These budgets determine when and for how long an action is applied without changing the total budget B. Our goal is to improve answer quality and completion while reducing unnecessary reasoning.

Control decisions depend on both token position and generated content, since traces of the same length may be at different reasoning stages. The controller therefore maintains a state

$$s_t = (\ell_t, b_t, u_t),$$

where ℓ_t tracks observed reasoning events and their active status, b_t stores the total and local budget information, and u_t tracks the current action and its progress.

3.2 Stateful Feedback and Control Actions

Monitoring. Before generating the next token, the monitor combines the current prefix with the controller state:

$$m_t = M(Y_{<t}, s_t). \tag{2}$$

Depending on the policy, the monitor extracts signals such as model-reported token budgets, repeated text patterns, and stage markers.

Applying control actions. The policy selects an action $a_t = \pi(m_t)$ using the event rule in Section 3.3. Let z_t denote the model's next-token logits. The operator $\mathcal{R}$ applies the action before decoding:

$$\widetilde{z}_t = \mathcal{R}(z_t, m_t, a_t), \qquad Y_t \sim \mathcal{D}_\xi(\widetilde{z}_t), \tag{3}$$

where $\mathcal{D}_\xi$ maps the adjusted logits to a next-token distribution under the chosen decoding rule.

The identity action a_{id} leaves the logits unchanged: $\mathcal{R}(z_t, m_t, a_{\mathrm{id}}) = z_t$. Other actions guide the reasoning or prompt the model to produce the final answer. The model then continues from the resulting prefix.

State updates. After generating Y_t, the controller updates its state:

$$s_{t+1} = U(s_t, m_t, a_t, Y_t). \tag{4}$$

The update records new events, action progress, and token use. Each generated token reduces the remaining budget by one, so $r_{t+1} = r_t - 1$, while local counters follow the policy-specific accounting rules. The state u_t allows a multi-token action to finish before the controller selects another event.

3.3 Event-Driven Control Policies

Eligible events. A policy defines an ordered event set $\mathcal{E}$. Each event e has a trigger condition g_e, an action map A_e, and an active flag $\ell_t(e) \in \{0, 1\}$. When no action is active, the eligible event set is

$$\mathcal{E}_t = \{e \in \mathcal{E} : \ell_t(e) = 1 \text{ and } g_e(m_t) = 1\}. \tag{5}$$

An event can trigger only when its condition is met and it remains active. Once its goal is reached, the controller deactivates it.

Action selection and commitment. Let $\prec$ denote the declared event priority order, and let $e_t = \min_{\prec} \mathcal{E}_t$ when $\mathcal{E}_t$ is nonempty. The policy is

$$\pi(m_t) = \begin{cases} \mathrm{resume}(u_t), & u_t \neq \varnothing, \\ A_{e_t}(m_t), & u_t = \varnothing \text{ and } \mathcal{E}_t \neq \varnothing, \\ a_{\mathrm{id}}, & \text{otherwise.} \end{cases} \tag{6}$$

The first branch continues an active action to completion. Otherwise, the controller selects the highest-priority eligible event or uses the identity action if none is eligible.

Updating events from feedback. The controller updates event states after each token using Equation equation 4. Observed transitions can deactivate events whose goals have been reached, while local budgets and control actions can update deadlines or enable later events.

Guidance actions redirect the ongoing reasoning, while answer actions start the final output. Event conditions determine when each action is used, allowing different inputs to follow different control paths.Different tasks instantiate this framework by specifying the event set, trigger conditions, and corresponding control actions. Triggers may depend on token position, observed stage markers, or repetition patterns, while the underlying controller mechanism remains unchanged.

4 EXPERIMENTS

To evaluate COTDOG from overall performance, configuration differences, reasoning behavior, and robustness, we organize our experiments around three research questions.

Q1: Effectiveness. How effective is COTDOG across models and tasks?

Q2: Ablation study. How does each component of COTDOG affect performance?

Q3: Failure analysis. How does COTDOG handle common reasoning failures?

4.1 EXPERIMENTAL SETUP

Datasets. We evaluate COTDOG on three benchmarks covering mathematical reasoning, scientific question answering, and code generation: (1) **MATH-500** (Lightman et al., 2023), a collection of 500 mathematical problems requiring multi-step reasoning, evaluated by answer accuracy; (2) **GPQA Diamond** (Rein et al., 2023), consisting of 198 challenging multiple-choice science questions, evaluated by answer accuracy; and (3) **LiveCodeBench** (Jain et al., 2024), using the 1,055 code-generation problems in release v6, evaluated by execution-based pass@1. Together, these benchmarks assess reasoning control across tasks with different solution structures and output requirements, ranging from short numerical or multiple-choice answers to complete executable programs.

Models. We conduct experiments on three publicly available, open-weight reasoning models: **QwQ-32B** (Qwen Team, 2025), **Qwen3.5-27B** (Qwen Team, 2026), and DeepSeek-R1-Distill-Qwen-32B (Guo et al., 2025) (**DS-Qwen-32B**). We apply COTDOG to each model. The same model produces the reasoning and final answer, while the controller monitors the evolving reasoning trace and applies control actions during decoding. For the Plan-and-Budget (Lin et al., 2026) baseline, we use Llama-3.1-8B-Instruct (Grattafiori et al., 2024) as the external planner. TALE-EP (Han et al., 2025) uses the corresponding target model to estimate a per-query token budget.

Baselines. We compare COTDOG with five baselines. (1) **Vanilla** uses the task input, answer-format requirements, and a basic instruction to reason step by step. It does not control the reasoning process during decoding. (2) **Structured Prompt** adds an instruction to follow four steps: Understand, Plan, Solve, and Verify. It relies on prompting alone. (3) **Plan-and-Budget (P&B)** (ICLR 2026) uses an external planner to divide the problem into sub-questions and assign relative importance credits. We combine these credits with its polynomial allocation strategy and a task difficulty budget (Lin et al., 2026). (4) **TALE-EP** (Findings of ACL 2025) estimates a token budget for each query before solving it, then adds the budget to the reasoning prompt (Han et al., 2025). (5) **Think Token Adjustment** (EMNLP 2025) increases the logit of `</think>` during decoding to encourage the model to end reasoning earlier (Liu & Wang, 2025).

Implementation details. Unless otherwise stated, all methods use greedy decoding (temperature 0), an 8,192-token completion limit, and a 16,384-token maximum context length. The sensitivity study additionally evaluates temperature 0.6 and completion limits of 16,384 and 32,768 tokens on Qwen3.5-27B/MATH-500, with the context length increased accordingly. The main experiments use simple model- and task-specific preset local budgets. Appendix B describes how they are estimated from generation lengths and used by the task policies. Additional evaluation details, policy configurations, baseline adaptations, and control prompts and pseudocode are provided in Appendices A, B, C, and E, respectively.

Evaluation Metrics. We report three metrics. (1) **Score (%)** is accuracy on MATH-500 and GPQA Diamond, and pass@1 on LiveCodeBench. We count unfinished or truncated outputs as incorrect. (2) **AvgTok** is the average number of completion tokens per query. We count reasoning, final answers, and control tokens. We also count planner tokens for P&B and budget-estimation tokens for TALE-EP. We do not count prompt tokens. (3) $\boldsymbol{\mathcal{E}^3}$ combines score and token use. Following P&B (Lin et al., 2026), we compute it as

$$\mathcal{E}^3 = \frac{A^2}{T}, \tag{7}$$

where A is Score in percent and T is the average token count.

Table 1: Performance and generation cost of CoTDog and baseline methods.

Dataset	Method	QwQ-32B			Qwen3.5-27B			DS-Qwen-32B		
		Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑
GPQA Diamond	Vanilla	41.4	5,668	0.302	51.5	6,606	0.401	43.9	5,407	0.356
	Structured Prompt	47.0	5,514	0.401	61.6	6,016	0.631	46.5	5,130	0.421
	P&B	50.5	4,534	0.562	57.1	6,581	0.495	56.1	3,331	0.945
	TALE-EP	52.0	4,487	0.603	73.7	3,927	1.383	51.5	4,672	0.568
	Think Token Adjustment	43.9	5,538	0.349	74.2	3,749	1.470	49.0	4,770	0.503
	CoTDog	**55.1**	**2,407**	**1.259**	**79.8**	**2,630**	**2.421**	**57.1**	**1,982**	**1.643**
MATH-500	Vanilla	88.0	3,562	2.174	57.4	5,760	0.572	89.0	2,482	3.191
	Structured Prompt	90.2	3,022	2.692	60.8	5,762	0.642	90.2	2,241	3.631
	P&B	89.8	2,269	3.554	88.2	5,480	1.420	92.2	1,454	5.847
	TALE-EP	91.6	2,236	3.752	90.6	3,598	2.281	92.2	**1,427**	**5.957**
	Think Token Adjustment	88.6	3,497	2.245	86.0	**2,180**	3.393	89.2	2,247	3.540
	CoTDog	**92.6**	**2,206**	**3.887**	**96.8**	2,404	**3.898**	**92.8**	1,659	5.191
LiveCodeBench	Vanilla	53.6	5,789	0.496	23.5	7,728	0.071	49.2	5,814	0.416
	Structured Prompt	53.7	5,773	0.500	28.6	7,543	0.108	50.3	5,725	0.442
	P&B	58.4	5,665	0.602	57.2	6,068	0.539	54.2	5,416	0.542
	TALE-EP	55.1	5,632	0.539	47.3	6,547	0.342	51.3	5,557	0.474
	Think Token Adjustment	54.2	5,766	0.510	55.4	**4,217**	0.727	52.5	4,101	0.672
	CoTDog	**63.5**	**4,919**	**0.820**	**76.0**	4,726	**1.222**	**57.3**	**3,092**	**1.062**

4.2 Q1: How effective is CoTDog across models and tasks?

Overall performance. CoTDog improves solution quality while reducing token use across all nine model and dataset settings relative to both Vanilla and Structured Prompt (Table 1). Relative to Vanilla, it lowers the average token count in every setting, with reductions ranging from 15.0% to 63.3%, and raises Score by 3.8 to 52.5 percentage points. Across all compared methods, CoTDog achieves the highest Score in all nine settings, the highest $\mathcal{E}^3$ in eight, and the lowest token use in six. The consistent gains over Structured Prompt indicate that the initial reasoning structure alone does not account for the performance of the full controller.

Comparison with reasoning controls. The gains remain substantial against methods that already regulate reasoning length, particularly for Qwen3.5-27B. On this model, CoTDog reaches 79.8% on GPQA Diamond, 96.8% on MATH-500, and 76.0% on LiveCodeBench; the strongest baseline scores are 74.2%, 90.6%, and 57.2%, respectively. On LiveCodeBench, it improves upon P&B by 18.8 points while reducing the average token count from 6,068 to 4,726. For QwQ-32B, CoTDog also improves all three scores over the strongest baseline, including an increase from 58.4% to 63.5% on LiveCodeBench with fewer tokens. On DS-Qwen-32B GPQA, CoTDog reaches 57.1% versus 56.1% for P&B while using 40.5% fewer completion tokens. These results support controlling both the progress of reasoning and its transition to the final output.

Observed trade-offs. The highest Score does not always coincide with the shortest output. On DS-Qwen-32B MATH, TALE-EP has higher $\mathcal{E}^3$ (5.957 versus 5.191), reflecting its shorter outputs. Think Token Adjustment also uses fewer tokens on Qwen3.5-27B MATH and LiveCodeBench, with lower scores of 86.0% and 55.4%. The main comparison measures completion-token efficiency. The runtime study uses fixed subsets and does not establish deployment-scale performance.

4.3 Q2: How does each component of CoTDog affect performance?

Ablation design. We compare six ablation configurations that differ in guidance, budget control, and trace-based feedback (Table 2). **Vanilla** and **Prompt Only** match Vanilla and Structured Prompt in the main comparison. **Prompt & Budget** retains the structured prompt and the controls for moving to the answer and stopping generation, but removes guidance during reasoning. **Prompt & Guidance** retains CoTDog's guidance text and trigger points, but removes forced reasoning closure, timed finishing instructions, and controller-forced termination. The GPQA verbal repetition reminder remains. The shared completion limit remains in force for every variant. **Open-Loop Control** uses the same initial prompt, action text, and budget thresholds as CoTDog. After the

Table 2: Ablation study of CoTDog.

Dataset	Method	QwQ-32B			Qwen3.5-27B			DS-Qwen-32B		
		Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑
GPQA Diamond	Vanilla	41.4	5,668	0.302	51.5	6,606	0.401	43.9	5,407	0.356
	Prompt Only	47.0	5,514	0.401	61.6	6,016	0.631	46.5	5,130	0.421
	Prompt & Budget	49.0	3,803	0.631	75.3	4,594	1.233	54.5	2,198	1.353
	Prompt & Guidance	54.0	3,513	0.831	74.2	4,054	1.360	45.5	4,614	0.448
	Open-Loop Control	53.0	2,698	1.042	76.3	3,092	1.881	**57.6**	1,995	**1.661**
	CoTDog	**55.1**	**2,407**	**1.259**	**79.8**	**2,630**	**2.421**	57.1	**1,982**	1.643
MATH-500	Vanilla	88.0	3,562	2.174	57.4	5,760	0.572	89.0	2,482	3.191
	Prompt Only	90.2	3,022	2.692	60.8	5,762	0.642	90.2	2,241	3.631
	Prompt & Budget	80.0	**1,936**	3.306	90.8	3,687	2.236	92.4	2,073	4.118
	Prompt & Guidance	91.0	2,422	3.419	93.4	2,552	3.419	90.6	1,749	4.692
	Open-Loop Control	92.2	2,381	3.570	95.8	2,641	3.475	**93.0**	1,751	4.940
	CoTDog	**92.6**	2,206	**3.887**	**96.8**	**2,404**	**3.898**	92.8	**1,659**	**5.191**
LiveCodeBench	Vanilla	53.6	5,789	0.496	23.5	7,728	0.071	49.2	5,814	0.416
	Prompt Only	53.7	5,773	0.500	28.6	7,543	0.108	50.3	5,725	0.442
	Prompt & Budget	62.6	4,857	0.806	71.3	4,982	1.020	**58.7**	3,286	1.048
	Prompt & Guidance	55.5	5,628	0.546	35.3	7,204	0.173	53.9	5,017	0.580
	Open-Loop Control	50.5	**4,158**	0.614	49.7	**4,540**	0.543	47.2	**2,830**	0.787
	CoTDog	**63.5**	4,919	**0.820**	**76.0**	4,726	**1.222**	57.3	3,092	**1.062**

reasoning budget is set, the token position of each action remains fixed. Each action runs once when generation reaches its token threshold. **CoTDog** includes all controller components, using trace-based feedback to update control decisions. On GPQA, this includes Final Commit, which ends generation after a supported explicit final choice appears outside the thinking section (Appendix B). Appendix C defines each configuration and its task-specific control behavior.

Output length distributions. Figure 2 shows that the token reductions are not limited to the averages in Table 2. Across the nine settings, CoTDog's median output length is 2,998 tokens, compared with 6,959 for Vanilla, 4,123 for Prompt & Guidance, and 3,202 for Prompt & Budget. The share of outputs using at least 8,000 tokens falls from 45.7% for Vanilla and 32.9% for Prompt & Guidance to 11.3% for CoTDog. Compared with Vanilla, the full controller shortens typical outputs and reduces the number of outputs near the token limit. Notably, Open-Loop Control has a lower average token count than CoTDog (3,250 and 3,415, respectively), but achieves a lower Score in seven of the nine settings. Appendix D separates the nine settings to show how these patterns vary across models and datasets.

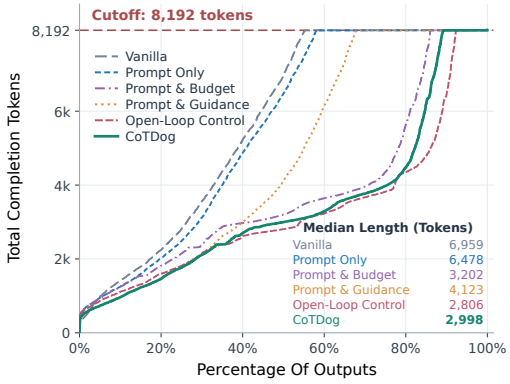


Figure 2: Completion lengths for six variants. Each method has 5,259 equally weighted outputs. Details and per-setting curves are shown in Appendix D.

Guidance helps, but termination control is still needed. Prompt & Guidance is effective relative to Prompt Only. It reduces the average token count in all nine settings, improves Score in eight, and achieves higher $\mathcal{E}^3$ in all nine. Adding termination control produces further gains. CoTDog achieves a higher Score and a lower average token count in all nine settings. The largest difference occurs on Qwen3.5-27B LiveCodeBench, where Score increases from 35.3% to 76.0% and the average token count falls from 7,204 to 4,726. The number of unfinished outputs also decreases from 672 to 142 out of 1,055 problems. This shows that guidance can improve the reasoning process, but termination control is needed to turn that reasoning into complete answers within the token budget.

Budget control alone is not enough. Prompt & Budget is effective relative to Prompt Only. It reduces the average token count in all nine settings, improves Score in eight, and achieves higher $\mathcal{E}^3$ in all nine. The complete controller provides additional benefits. Compared with Prompt & Budget, CoTDog produces more correct answers in eight of the nine settings and achieves higher $\mathcal{E}^3$ in all nine. Across the 5,259 matched examples, CoTDog achieves a net gain of 166 correct answers over Prompt & Budget (Table 10). This gain consists of 68 from changes between completed wrong and correct outputs and 98 from changes between unfinished and correct outputs. This shows that the full system improves not only the recovery of unfinished outputs but also the correctness of completed answers.

Trace-based feedback improves upon fixed scheduling. Open-Loop Control improves upon Prompt & Budget by adding guidance at fixed token positions. It achieves a higher Score in all six GPQA and MATH-500 settings, with gains of up to 12.2 percentage points, although it performs worse in all three LiveCodeBench settings. Making the schedule responsive to the evolving reasoning trace improves performance further. CoTDog achieves a higher Score in seven of the nine settings and higher $\mathcal{E}^3$ in eight. The largest differences appear on LiveCodeBench, where CoTDog improves Score by 13.0, 26.3, and 10.1 percentage points across the three models. This shows that a fixed schedule can strengthen budget control, but trace-based feedback is needed to adapt control actions to the current reasoning progress.

4.4 Q3: How does CoTDog handle common reasoning failures?

We compare three matched Vanilla and CoTDog outputs from DS-Qwen-32B on MATH-500, with full details in Appendix F. The examples cover overthinking, underthinking, and repetition. They show three different effects of reasoning control: keeping a correct result, completing missing work, and ending repeated reasoning. Because these are selected cases, they do not show how often each failure occurs or separate the effect of each control action. Table 3 summarizes their final answers and completion lengths.

Overthinking: reversing a correct answer. For circular seating (ID 32; Figure 4), Vanilla correctly obtains 720 with a block-based solution but continues reasoning. It later rejects the correct method and changes the answer to 240. CoTDog checks the internal order of the block and keeps the correct result. This case shows that unnecessary reasoning after a complete solution can turn a correct answer into a wrong one.

Table 3: Final answers and completion lengths.

Case	Vanilla	**CoTDog**
Overthinking ID 32	$Answer: 240$ × 4,123 tokens	$Answer: 720$ ✓ 872 tokens
Underthinking ID 462	$Answer: 60$ × 469 tokens	$Answer: 59$ ✓ 1,266 tokens
Repetition ID 153	Unfinished 8,192 tokens	$Answer: 2/21$ ✓ 3,420 tokens

Selected DS-Qwen-32B / MATH-500 cases at temperature 0. ✓ correct; × incorrect.

Underthinking: stopping before the calculation. For pooled money (ID 462; Figure 5), Vanilla describes the correct plan but ends reasoning before adding the four amounts. It then uses the wrong total of \$126.60 and returns 60 cents. CoTDog continues the calculation, checks the total of \$126.59, and returns 59 cents. Unlike the overthinking case, solving this problem correctly requires more reasoning rather than less.

Repetition: breaking a reasoning loop. For grid collinearity (ID 153; Figure 6), Vanilla repeats the correct probability sentence 91 times and reaches 8,192 tokens without a completed answer. CoTDog also starts to repeat the sentence, but a loop-control action stops the repetition after ten occurrences and moves the model to the answer. It then returns the correct $2/21$. This case shows that the controller can recover from a loop even when it does not prevent the loop from starting.

Together, these cases show that CoTDog does not simply shorten every output. It reduces extra work after a correct solution, allows more work when reasoning is incomplete, and stops repeated text when the model no longer makes progress. This behavior supports controlling reasoning based on its current state rather than applying the same fixed limit to every problem. Appendix F gives the evidence checks for these three pairs and a compact residual-repetition example; complete generation traces are provided in the supplementary artifact.

4.5 Robustness and Sensitivity

We conduct additional studies to assess CoTDog's runtime overhead, adaptive budget initialization, and sensitivity to decoding settings (Appendix D.3).

Runtime overhead. CoTDog is implemented with an external controller. Therefore, latency is an important practical concern. We compare the runtime overhead of Vanilla and CoTDog on all three models and datasets. Across the nine settings, the per-token throughput (TPS) between the two approaches differs by at most 0.53%. Over all 90 timed problems, CoTDog reduces total generation time from 18,488.8 to 9,626.1 seconds through shorter outputs. The detailed results are shown in Table 11.

Adaptive budget initialization. The main experiments use the simple preset budget setting described in Appendix B. To examine how much the results depend on these presets, we also test an adaptive variant in which the target model estimates a local budget for each query. This variant shows that the controller can be paired with a different budget initialization, although the choice still affects accuracy and token use. Across the nine settings, adaptive initialization reduces the average token count in four and improves $\mathcal{E}^3$ in three. For example, on Qwen3.5-27B/LiveCodeBench, the average token count falls from 4,726 to 3,347 (i.e., 29.2%), while Score decreases from 76.0% to 72.6%. Full results are shown in Table 12.

Temperature and output-limit sensitivity. We test temperatures $\{0, 0.6\}$ and completion limits $\{8{,}192, 16{,}384, 32{,}768\}$ using Qwen3.5-27B on MATH-500. CoTDog achieves the highest Score, lowest average token count, and highest $\mathcal{E}^3$ in all six settings, while increasing the limit from 16,384 to 32,768 reduces $\mathcal{E}^3$ in five of the six comparisons. For example, at temperature 0.6 and the 32,768-token limit, CoTDog reaches 98.0% Score with an average token count of 2,876. Thus, larger limits provide little additional efficiency benefit, while CoTDog remains effective across decoding settings (Table 13).

5 Conclusion

We presented CoTDog, a closed-loop framework for controlling reasoning progression and answer production. Its controller combines visible trace observations with persistent execution state to guide continued reasoning, retire unnecessary interventions, and initiate answer production within a single generation process. This design requires neither model parameter updates nor an auxiliary reasoning model. Across three models and three benchmarks, CoTDog achieves a higher Score than the strongest baseline in all nine settings and the highest $\mathcal{E}^3$ in eight. Paired reasoning-trace analyses illustrate shorter correct solutions, completion of omitted computation, and interruption of repetitive reasoning. Overall, our results suggest that efficient reasoning is not only a question of when to stop, but also of how to guide ongoing computation toward a complete answer.

Limitations. Our study has several limitations. First, the main evaluation covers three open-weight models and three benchmarks, with most experiments using greedy decoding and an 8,192-token completion limit, which may limit the generalizability of our findings to other models and decoding settings. To mitigate this issue, we additionally evaluate temperature 0.6 and completion limits up to 32,768 tokens. Second, the current policies use task-specific observation rules, thresholds, and control text, which may require additional design when applying CoTDog to new tasks. To reduce this burden, we estimate preset budgets from generation lengths on a random 5% sample without using reference answers or evaluation scores, and further evaluate adaptive per-query budget initialization across all nine settings. Third, the controller relies on visible reasoning patterns rather than a correctness verifier, so some observed signals may be incomplete or misleading. To reduce the effect of such signals, the controller maintains persistent event state and removes actions for reasoning stages that have already been reached. Finally, most main results are based on a single run, while the runtime study uses ten examples per benchmark and model under one serving configuration, which limits conclusions about execution stability and system-level performance. To mitigate this limitation, we report runtime results across all nine settings and three-run results for the experiments under the temperature of 0.6.

AI USE STATEMENT

Generative AI tools were used solely to assist with language editing and polishing of the manuscript. Their role was limited to improving the wording and readability of text prepared by the authors and did not involve substantive contributions to the research. All AI-assisted edits were reviewed by the authors to ensure that the intended meaning and technical content were preserved. The authors take full responsibility for the final manuscript.

REPRODUCIBILITY STATEMENT

We provide the implementation of CoTDog in an anonymous repository at `https://anonymous.4open.science/r/CoTDog`, including instructions and configurations for reproducing the experiments. The proposed framework and control mechanism are described in the main text, with additional implementation details and experimental settings provided in the appendix.

REFERENCES

Ömer Faruk Akgül, Yusuf Hakan Kalaycı, Rajgopal Kannan, Willie Neiswanger, and Viktor Prasanna. LYNX: Learning dynamic exits for confidence-controlled reasoning. *arXiv preprint arXiv:2512.05325*, 2025.

Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Do NOT think that much for 2+3=? on the overthinking of long reasoning models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 9487–9499. PMLR, 2025.

Zenghao Duan, Liang Pang, Zihao Wei, Wenbin Duan, Yuxin Tian, Shicheng Xu, Jingcheng Deng, Zhiyi Yin, and Xueqi Cheng. Circular reasoning: Understanding self-reinforcing loops in large reasoning models. *arXiv preprint arXiv:2601.05693*, 2026.

Aaron Grattafiori et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24842–24855. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1274.

Jiameng Huang, Baijiong Lin, Guhao Feng, Jierun Chen, Di He, and Lu Hou. Efficient reasoning for large reasoning language models via certainty-guided reflection suppression. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(37):31176–31184, 2026. doi: 10.1609/aaai.v40i37.40379.

Naman Jain, King Han, 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. *arXiv preprint arXiv:2403.07974*, 2024.

Yulin Li, Tengyao Tu, Li Ding, Junjie Wang, Huiling Zhen, Yixin Chen, Yong Li, and Zhuotao Tian. Efficient reasoning with balanced thinking. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=cJseWJJ5IM`.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Junhong Lin, Xinyue Zeng, Jie Zhu, Song Wang, Julian Shun, Jun Wu, and Dawei Zhou. Plan and budget: Effective and efficient test-time scaling on reasoning large language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ctspw4CqbS.

Xin Liu and Lu Wang. Answer convergence as a signal for early stopping in reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17896–17907. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.904.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20275–20321. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1025.

Qwen Team. QwQ-32B: Embracing the power of reinforcement learning. Qwen Blog, 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof Q&A benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Yuan Sui, Yufei He, Tri Cao, Sophia Simeng Han, Yulin Chen, and Bryan Hooi. Meta-reasoner: Dynamic guidance for optimized inference-time reasoning in large language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 13268–13286. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.649.

Tengyao Tu, Yulin Li, Hui-Ling Zhen, Libo Qin, Zhoujun Wei, Jinghua Piao, Zhuotao Tian, Yong Li, and Min Zhang. DyCon: Dynamic reasoning control via evolving difficulty modeling. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2606.07108.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yue Wang, Qiuzhi Liu, Jiahao Xu, Tian Liang, Xingyu Chen, Zhiwei He, Linfeng Song, Dian Yu, Juntao Li, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Thoughts are all over the place: On the underthinking of long reasoning models. In *Advances in Neural Information Processing Systems*, volume 38, pp. 34514–34534, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Chenxu Yang, Qingyi Si, Yongjie Duan, Zheliang Zhu, Chenyu Zhu, Qiaowei Li, Minghui Chen, Zheng Lin, and Weiping Wang. Dynamic early exit in reasoning models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=NpU7ZXafRi.

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. Association for Computational Linguistics, 2026.

A Experimental Details and Evaluation

Evaluation data. We evaluate all 500 MATH-500 problems, all 198 GPQA Diamond questions, and all 1,055 code-generation problems in LiveCodeBench release v6. Each method produces one completion for every problem under matched model and dataset settings. This gives 5,259 completions per method across the nine settings.

Table 4: Benchmarks and evaluation procedures.

Benchmark	Requested Output	Evaluation
MATH-500	A boxed mathematical answer.	Compare normalized answers, with symbolic verification when available.
GPQA Diamond	One choice from A, B, C, or D.	Compare the extracted letter with the label under a fixed option order.
LiveCodeBench	One complete Python program.	Use the official release-v6 execution checker with one candidate per problem.

MATH prompts contain the original problem without the stored solution or answer. GPQA option order is fixed before inference and shared by all methods. LiveCodeBench tests and reference answers are available only to the evaluator. We do not sample subsets, use majority voting, or select among multiple candidates.

Inference protocol. We use each model's base checkpoint and tokenizer without training or adapters. The main experiments use greedy decoding and an 8,192-token completion limit for every method. CoTDog uses the preset model- and task-specific local budgets reported in Appendix B. These budgets determine local control points but do not change the common completion limit.

Answer extraction and completion. For MATH, we extract the final boxed answer and apply normalized and symbolic comparison. For GPQA, we extract the final answer letter after normalizing formatting and reasoning markers. For LiveCodeBench, we extract one program and pass it to the official evaluator. A length-limited output is counted as incorrect even if it contains a partial answer. The earlier P&B records for MATH and GPQA do not include finish reasons, so outputs that reach the solver limit are treated as unfinished.

Token accounting. Let s_i be the solver completion tokens for problem i and h_i its auxiliary completion tokens. We report

$$T = \frac{1}{N}\sum_{i=1}^{N}(s_i + h_i), \qquad A = \frac{100}{N}\sum_{i=1}^{N}\mathbf{1}[\text{completed and correct}], \qquad \mathcal{E}^3 = \frac{A^2}{T}.$$

Incorrect and unfinished outputs remain in the denominator of both Score and the average token count.

Table 5: Completion tokens included in AvgTok.

Cost Component	Included?
Solver reasoning and final answer	Yes
Injected CoTDog control text	Yes
P&B planner completion	Yes
TALE-EP estimator completion	Yes
Input and system prompts	No
Evaluation and program execution	No

The runtime comparison in Appendix D.3 reports both per-token throughput and total generation time.

A.1 Runs and reproducibility

The main tables report one execution for each setting. The temperature-0.6 sensitivity results report the mean and standard deviation over three runs. The supplementary artifact provides the run manifest, configurations, raw outputs, and scripts used to reconstruct the reported results.

B CoTDog Policy Instantiations

B.1 Preset Budget Setting

For each benchmark, we randomly sample 5% of the inputs and run Structured Prompt with each target model on this sample. For each model and benchmark, we keep only outputs whose reasoning completes normally. We exclude unfinished outputs and outputs truncated by the token limit because their lengths do not represent a complete CoT. Let $\bar{\ell}$ be the mean generated CoT length over the remaining outputs. The preset local budget is

$$b = 256 \left\lceil \frac{\bar{\ell}}{256} \right\rceil .$$

Thus, b is the mean completed CoT length rounded up to the nearest multiple of 256. Apart from the completion filter above, this process uses only generation length. It does not evaluate the answers or use reference answers, output correctness, or evaluation scores, and no other filtering is applied. The sampled 5% of inputs remain in the final benchmark evaluation. Because reasoning requirements may vary across models and tasks, our goal is only to obtain a rough model- and task-specific budget estimate. This setting does not leak answer or score information. To address concerns that this preset procedure could affect evaluation fairness, we also test an adaptive variant that estimates a local budget independently for each query across all nine settings (Appendix D.3).

The relative positions of control events are still determined by the fixed task-specific policy rules below.

Table 6: Preset local budgets b. Control event positions are calculated from b using the task-specific policy rules below.

Dataset	QwQ-32B	Qwen3.5-27B	DS-Qwen-32B
MATH-500	1,536	2,816	1,024
GPQA Diamond	2,816	3,072	2,560
LiveCodeBench	2,816	3,840	2,816

The main completion limit is $B = 8{,}192$. The sensitivity study also considers $B \in \{16{,}384, 32{,}768\}$ while retaining the policy associated with each decoding setting.

B.2 Events and token counts

Each request maintains an ordered set of events and its own controller state. Events can respond to token position, visible output patterns, or repetition. The first active matching event applies after any current action finishes. An action may insert guidance, update later events, start the final answer, or end generation. The controller observes only the generated text and its saved state. It does not consult the reference answer.

Let g count ordinary generation tokens and c count control-action tokens. Both are produced through the model decoding interface and contribute to the generated prefix. The total completion length is $g + c$, and both types of tokens count toward the completion limit and AvgTok. Policy thresholds are measured using g. A *checkpoint* adds guidance without ending reasoning, whereas a *handoff* closes reasoning and begins the required answer format.

B.3 TASK POLICIES

MATH-500. An initial Understand action is followed by Plan, Solve, Focus, Verify, and Finish checkpoints at

$$(\lceil 0.12b \rceil, \lceil 0.25b \rceil, \lceil 0.50b \rceil, \lceil 0.70b \rceil, \lceil 0.90b \rceil).$$

Observed stage headings remove checkpoints for stages that the model has already reached. For QwQ-32B and Qwen3.5-27B, all five actions are guidance checkpoints and do not force an answer at b. For DS-Qwen-32B, a late repetition event can start the answer after $3b$, and a hard handoff remains at $6b$. After either transition, the controller ends generation after the boxed answer line.

GPQA Diamond. The policy begins with a compact reasoning instruction. A repetition reminder becomes active at $\lceil 0.35b \rceil$, a checkpoint acts at $\lceil 0.60b \rceil$, and an answer handoff acts at $\lceil 0.90b \rceil$ if it is still needed. After reasoning closes, the controller ends generation when it observes a complete answer line containing one choice from A, B, C, or D. All choices are treated symmetrically.

LiveCodeBench. At the first newline after $\lceil 0.75b \rceil$, the policy asks the model to check the solution plan. At the first newline after $\lceil 0.90b \rceil$, it asks the model to produce the program. A hard handoff remains at b if this transition has not occurred. The controller can end generation after a complete closing code fence.

C ABLATIONS AND BASELINE ADAPTATIONS

C.1 ABLATION CONFIGURATIONS

All configurations in Table 2 use the same target model, problems, decoding rule, and solver completion limit within each setting.

Prompt & Budget. This configuration keeps the structured prompt and task-specific answer transitions but removes runtime reasoning guidance. On GPQA, LiveCodeBench, and MATH-500, the transitions follow the corresponding COTDOG policy.

Prompt & Guidance. This configuration retains runtime guidance and removes forced answer transitions and output stopping. MATH keeps the stage guidance, GPQA keeps the compact checkpoint and repetition reminder, and LiveCodeBench keeps the verification action. Generation otherwise ends naturally or at the common limit.

Open-Loop Control. This configuration is designed as the fixed-schedule counterpart of COTDOG. It uses the same initial prompt, control-action text, preset local budget, task-specific answer-format safeguards, and final-output completion rules. Reasoning-side interventions are only applicable before final-answer generation begins. The key difference is that Open-Loop Control does not update pending control events from the observed reasoning trace. Once the local budget is initialized, retained reasoning and transition actions follow predetermined token thresholds rather than being canceled or adapted according to stage progress.

C.2 BASELINE ADAPTATIONS

Plan-and-Budget. We reproduce the P&B variant that allocates a global budget across planned subproblems (Lin et al., 2026). Llama-3.1-8B-Instruct generates two to five subproblems with importance credits that sum to 100. For normalized credit c_i and zero-based position i among m steps, the allocation is

$$b_i = \left\lfloor B_\ell \frac{c_i(m-i)^2}{\sum_{j=0}^{m-1} c_j(m-j)^2} \right\rfloor .$$

The level budgets are $(200, 250, 350, 450, 600)$. MATH uses its annotated level, GPQA uses level five, and LiveCodeBench maps easy, medium, and hard problems to levels two, three, and five. The allocations are soft word limits in the solver prompt. Planner completion tokens are included in the average token count.

Table 7: Adaptations of the comparison methods. Auxiliary completions are included in AvgTok.

Method	Adaptation	Auxiliary Completion
Vanilla and Structured	Common task wrappers with either a basic reasoning instruction or the four-stage prompt.	None
P&B	Task adapters and the published allocation rule. Allocations remain soft prompt constraints.	Llama-3.1-8B-Instruct planner
TALE-EP	A target-model budget estimate added as a soft prompt constraint.	One target-model estimator call
Think Token Adjustment	The published logit adjustment applied to each model's reasoning-end token.	None

TALE-EP. We use TALE's training-free estimate-and-prompt variant (Han et al., 2025). The target model first estimates a token budget, which is clipped to the common output limit and added to the solver prompt as a soft constraint. The solver may continue to the common completion limit. Estimator completion tokens are included in the average token count.

Think Token Adjustment. Before the first reasoning-end marker, we apply the inference-time adjustment from Liu and Wang (Liu & Wang, 2025)

$$z_{\text{end}} \leftarrow z_{\text{end}} + 0.6 \left(\max_{v \in V} z_v - \frac{1}{|V|} \sum_{v \in V} z_v \right).$$

The adjustment is disabled after that marker and does not insert an answer. We evaluate it on the same models and benchmark collections as the other methods.

All methods use the same task and target model within each setting. The supplementary artifact contains the complete prompt templates and configurations.

D Additional Quantitative Results

D.1 Pooled Length and Outcome Analyses

The length analyses use the same executions as Table 2 and include correct, incorrect, and unfinished outputs. Each method contributes 5,259 outputs, so pooled results weight benchmark examples rather than datasets equally.

Table 8: Pooled completion statistics over 5,259 outputs per method. Length statistics include all outcomes. Near-limit denotes at least 8,000 tokens.

Method	AvgTok	Median Tokens	Near-Limit (%)	Unfinished (%)
Vanilla	5,666	6,959	45.7	44.7
Prompt Only	5,495	6,478	42.6	41.9
Prompt & Budget	3,764	3,202	14.1	14.0
Prompt & Guidance	4,678	4,123	32.9	32.3
Open-Loop Control	3,250	2,806	7.9	7.7
CoTDog	3,415	2,998	11.3	10.8

Open-Loop Control has a shorter pooled median and fewer near-limit outputs than CoTDog, but a lower Score in seven of nine settings. Completion length therefore does not by itself indicate answer quality or completion.

Across the 5,259 outputs, CoTDog leaves fewer unfinished outputs than Vanilla, Prompt Only, Prompt & Budget, and Prompt & Guidance, with 569 unfinished in total (Table 9). Open-Loop Control leaves even fewer, at 407, and therefore completes 162 more outputs than CoTDog (i.e., a 3.1-percentage-point increase in completion rate). However, its mean Score is 6.2 percentage

Table 9: Unfinished outputs by setting. GPQA, MATH, and LCB contain 198, 500, and 1,055 problems. PO, PB, PG, and OL denote the four ablation variants.

Dataset	Model	Vanilla	PO	PB	PG	OL	CoTDoG
GPQA Diamond	QwQ-32B	86	71	26	27	10	4
	Qwen3.5-27B	90	68	30	27	5	0
	DS-Qwen-32B	81	74	2	80	1	0
MATH-500	QwQ-32B	52	45	19	32	31	23
	Qwen3.5-27B	213	195	41	25	15	10
	DS-Qwen-32B	38	37	2	27	3	5
LiveCodeBench	QwQ-32B	474	469	323	440	227	323
	Qwen3.5-27B	806	750	199	672	93	142
	DS-Qwen-32B	509	496	93	367	22	62
Total		**2,349**	**2,205**	**735**	**1,697**	**407**	**569**

points lower (i.e., 68.4% compared with 74.6%). This shows that completing more outputs does not guarantee better answers and that trace-based feedback is important for balancing completion and correctness.

Table 10: Paired outcome changes from Prompt & Budget to CoTDoG. U, W, and C denote unfinished, completed but wrong, and completed and correct outputs, respectively.

Dataset	Model	U→C	W→C	C→U	C→W	W/C Net Gain	U/C Net Gain	Total Net Gain
GPQA Diamond	QwQ-32B	12	15	0	15	0	+12	+12
	Qwen3.5-27B	9	6	0	6	0	+9	+9
	DS-Qwen-32B	0	17	0	12	+5	0	+5
MATH-500	QwQ-32B	6	64	6	1	+63	0	+63
	Qwen3.5-27B	30	1	1	0	+1	+29	+30
	DS-Qwen-32B	1	12	0	11	+1	+1	+2
LiveCodeBench	QwQ-32B	37	18	27	18	0	+10	+10
	Qwen3.5-27B	50	41	17	24	+17	+33	+50
	DS-Qwen-32B	6	48	2	67	−19	+4	−15
Total		**151**	**222**	**53**	**154**	**+68**	**+98**	**+166**

The paired analysis matches problem identifiers between Prompt & Budget and CoTDoG. We decompose the change in the number of correct answers as

$$G_{\mathrm{W/C}} = N_{\mathrm{W}\to\mathrm{C}} - N_{\mathrm{C}\to\mathrm{W}}, \qquad G_{\mathrm{U/C}} = N_{\mathrm{U}\to\mathrm{C}} - N_{\mathrm{C}\to\mathrm{U}}, \qquad \Delta N_{\mathrm{correct}} = G_{\mathrm{W/C}} + G_{\mathrm{U/C}}.$$

Transitions between U and W are omitted because they do not affect the number of correct answers. Across the 5,259 matched examples, the W/C component contributes a net gain of 68 correct answers, while the U/C component contributes 98, yielding a total net gain of 166. Therefore, the gain of CoTDoG is not limited to recovering unfinished outputs.

D.2 Completion-Length Distributions by Setting

Figure 3 shows the completion-length distributions for each model and benchmark setting.

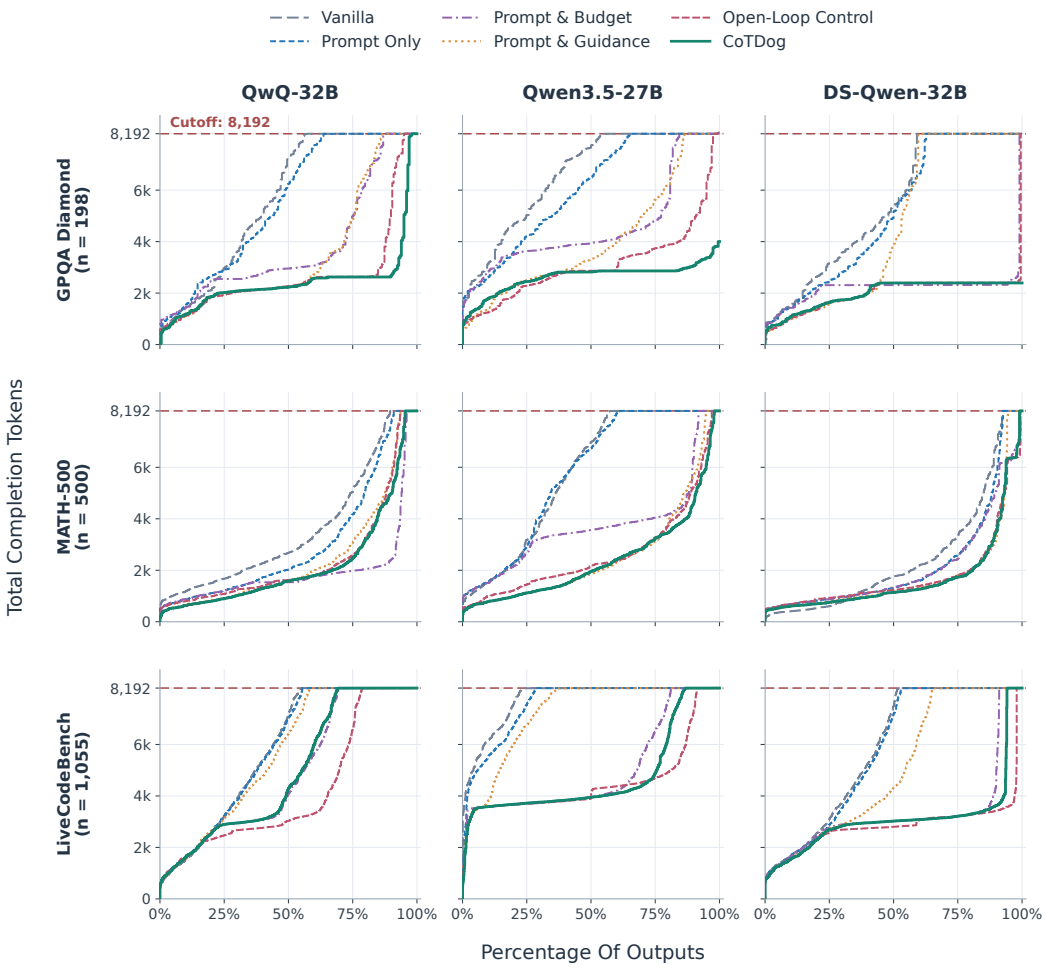


Figure 3: Completion length distributions at temperature zero. Rows show benchmarks and columns show models. Each panel includes all six variants and all output outcomes. The dashed line marks the 8,192-token limit.

D.3 Runtime and Sensitivity Analyses

The runtime study covers all three models and benchmarks. The adaptive-budget study covers all nine settings. The decoding study uses Qwen3.5-27B on MATH-500. All accuracy results count unfinished outputs as incorrect.

Runtime overhead. We time Vanilla and CoTDog on ten examples from each benchmark for every model, yielding 30 examples per model and 90 in total. For each single-threaded run i, we compute throughput as

$$\mathrm{TPS}_i = \frac{N_i^{\mathrm{total}}}{t_i^{\mathrm{end}} - t_i^{\mathrm{start}}},$$

where N_i^{total} is the total number of generated tokens, including inserted control tokens, and t_i^{start} and t_i^{end} are the inference start and end times. The timing interval covers the complete generation process, including controller monitoring and control actions.

Table 11: Runtime comparison using ten examples from each benchmark. TPS reports mean ± standard deviation across 30 examples per model. Total time sums the same examples.

Model	Vanilla TPS	CoTDoG TPS	Largest TPS Difference	Vanilla Time (s)	CoTDoG Time (s)	Reduction
Qwen3.5-27B	26.715 ± 0.085	26.753 ± 0.124	0.23%	7,185.1	3,133.6	56.4%
QwQ-32B	22.668 ± 0.130	22.711 ± 0.166	0.53%	5,643.3	3,809.7	32.5%
DS-Qwen-32B	22.757 ± 0.253	22.804 ± 0.168	0.50%	5,660.4	2,682.8	52.6%

Table 12: Comparison of preset and adaptive CoTDoG configurations at temperature 0 with an 8,192-token completion limit. Adaptive initialization obtains a per-query local budget from the target model. Bold marks the better result between the two configurations within each setting.

Dataset	Model	Preset CoTDoG			Adaptive CoTDoG			Δ Score	Δ AvgTok
		Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑		
GPQA Diamond	QwQ-32B	**55.1**	**2,407**	**1.259**	53.0	2,550	1.103	−2.0	+142
	Qwen3.5-27B	**79.8**	**2,630**	**2.421**	77.8	3,046	1.986	−2.0	+416
	DS-Qwen-32B	57.1	**1,982**	**1.643**	**58.1**	2,714	1.243	+1.0	+731
MATH-500	QwQ-32B	**92.6**	2,206	3.887	90.4	**1,544**	**5.294**	−2.2	−662
	Qwen3.5-27B	**96.8**	2,404	3.898	95.8	**2,022**	**4.540**	−1.0	−382
	DS-Qwen-32B	**92.8**	**1,659**	**5.191**	89.8	1,718	4.695	−3.0	+59
LiveCodeBench	QwQ-32B	**63.5**	4,919	**0.820**	59.7	**4,640**	0.769	−3.8	−279
	Qwen3.5-27B	**76.0**	4,726	1.222	72.6	**3,347**	**1.575**	−3.4	−1,379
	DS-Qwen-32B	**57.3**	**3,092**	**1.062**	54.6	4,436	0.672	−2.7	+1,344

Δ denotes Adaptive minus Preset. The two configurations are evaluated end to end, so the comparison characterizes the resulting accuracy-efficiency trade-off rather than isolating budget initialization as a single component.

Across the nine settings, the difference in per-token throughput is at most 0.53%. Because CoTDoG produces shorter outputs, total generation time falls from 18,488.8 to 9,626.1 seconds (i.e., 47.9%). The runtime measurements use temperature 0.6 and are separate from the main accuracy runs.

Adaptive budget initialization. The main experiments use the preset budgets in Appendix B. To test a different initialization, the adaptive variant asks the target model to estimate a local budget for each query before solving and uses that estimate to set later control points. The estimation and control tokens are included in the average token count.

Adaptive initialization reduces the average token count in four of the nine settings and increases $\mathcal{E}^3$ in three, but improves Score only on DS-Qwen-32B/GPQA Diamond. The change in average token count ranges from a reduction of 1,379 on Qwen3.5-27B/LiveCodeBench (i.e., 29.2%) to an increase of 1,344 on DS-Qwen-32B/LiveCodeBench (i.e., 43.5%). The effectiveness of model-estimated budgets therefore varies substantially across models and tasks.

Despite this variation, Adaptive CoTDoG still exceeds the strongest baseline Score in seven of the nine settings (Tables 1 and 12). The gains range from 0.4 percentage points on DS-Qwen-32B/LiveCodeBench to 15.4 points on Qwen3.5-27B/LiveCodeBench, where Score reaches 72.6% compared with the strongest baseline's 57.2%. Adaptive CoTDoG also achieves higher $\mathcal{E}^3$ than every baseline in seven settings. Thus, the controller can also use model-estimated budgets, but budget initialization still affects accuracy and token use.

Temperature and output limit. We compare Vanilla, Structured Prompt, and CoTDoG on all 500 MATH-500 problems at temperatures 0 and 0.6 with completion limits of 8,192, 16,384, and 32,768 tokens.

CoTDoG achieves the highest Score, lowest average token count, and highest $\mathcal{E}^3$ in all six settings. Increasing the completion limit from 16,384 to 32,768 lowers $\mathcal{E}^3$ in five of the six comparisons. The only exception is Vanilla at temperature 0.6, where $\mathcal{E}^3$ increases slightly from 0.704 to 0.709. For CoTDoG, $\mathcal{E}^3$ decreases from 3.526 to 3.415 at temperature 0 and from 3.588 to 3.341 at temperature 0.6. At the 32,768-token limit, CoTDoG still reaches Scores of 97.6% and 98.0% with average token counts of 2,790 and 2,876, whereas the prompt-only baselines average between 10,841 and 15,789 tokens. Extending the limit beyond 16,384 therefore provides little additional efficiency benefit, while CoTDoG remains robust to the larger generation budget.

Table 13: Robustness results for Qwen3.5-27B on MATH-500 under different decoding temperatures and generation limits. Each method reports strict-finish Score, average completion-token count (AvgTok), and efficiency $\mathcal{E}^3$. Bold marks the best value within each setting.

Temperature	Max. Tokens	Vanilla			Structured Prompt			CoTDog		
		Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑	Score ↑	AvgTok ↓	$\mathcal{E}^3$ ↑
0	8,192	57.4	5,760	0.572	60.8	5,762	0.642	**96.8**	**2,404**	**3.898**
	16,384	79.6	7,994	0.793	75.8	8,146	0.705	**96.6**	**2,647**	**3.526**
	32,768	80.6	11,490	0.565	77.4	11,674	0.513	**97.6**	**2,790**	**3.415**
0.6	8,192	56.5 ± 8.6	5,870 ± 299	0.558 ± 0.201	39.8 ± 2.2	6,255 ± 132	0.254 ± 0.034	**95.9 ± 0.4**	**2,604 ± 66**	**3.537 ± 0.111**
	16,384	76.8 ± 4.3	8,510 ± 813	0.704 ± 0.155	59.4 ± 2.2	10,295 ± 226	0.344 ± 0.033	**98.1 ± 0.3**	**2,687 ± 121**	**3.588 ± 0.146**
	32,768	87.1 ± 2.3	10,841 ± 1,179	0.709 ± 0.122	68.1 ± 1.5	15,789 ± 495	0.295 ± 0.022	**98.0 ± 0.2**	**2,876 ± 72**	**3.341 ± 0.078**

Score is accuracy in percent under strict-finish evaluation: every non-stop or length-limited generation is incorrect. AvgTok is the average completion-token count, including control tokens. We compute $\mathcal{E}^3 = A^2/T$, where A is the unrounded percentage Score and T is the unrounded AvgTok. Temperature-0 entries use seed 42. Temperature-0.6 entries report mean ± sample standard deviation over seeds 42, 43, and 44; $\mathcal{E}^3$ is computed per run before aggregation. The historical temperature-0/8,192-token predictions are re-scored with the same strict finish rule. All runs use batch size 1,024.

E Prompts and Pseudocode

E.1 Solver and control prompts

Vanilla uses the instruction `Please reason step by step.` Structured Prompt and CoTDog use the following reasoning instruction before the task and output requirements.

Shared Reasoning Prompt

```
Use the following concise reasoning structure before the final answer:
## Understand --- Identify the givens, target, constraints, and
required answer format.
## Plan --- Choose one solution route.
## Solve --- Execute the route without restarting or looping.
## Verify --- Check the answer against the problem and repair only
local errors.
```

Table 14: Required output format for each benchmark.

Task	Required Final Output
MATH-500	Put the final answer in the form `\boxed{...}`.
GPQA Diamond	Output `**Final Answer:** Answer: X`, where X is exactly one of A, B, C, or D.
LiveCodeBench	Output exactly one complete Python solution in a Markdown code fence, with no prose outside the fence.

CoTDog, Prompt & Guidance, and Open-Loop Control also receive the following monitor instruction.

Monitor Instruction

```
An external CoTDog monitor may insert short control text inside your
chain-of-thought while you are solving, including monitor messages or
structured headings.  Treat inserted CoTDog text as runtime control
instructions, continue from it naturally, and do not copy or discuss
the monitor in the final answer unless the user explicitly asks about
CoTDog.
```

Input prompts are excluded when computing the average token count. Control-action tokens are included.

Control actions. Line breaks are omitted for presentation. Reasoning-end markers and task-specific answer prefixes are inserted at the transitions defined in Appendix B.

Action	Action Wording
MATH: Understand	`## Understand. List givens, target quantity, constraints, and answer format only. Do not start broad search here.`
MATH: Plan	`## Plan. Select one solution route now. Name the key formula, theorem, substitution, or cases to check. Keep the plan short.`
MATH: Solve	`## Solve. Execute the chosen route. Do not restart the problem unless a contradiction is found; repair locally and keep derivation compact.`
MATH: Focus	`Checkpoint: continue the current solution path. Avoid re-deriving the setup; finish the computation, track domain/case assumptions, and keep one candidate answer.`
MATH: Verify	`## Verify. Check the candidate against constraints, signs, cases, and simplification. Fix only local errors, then prepare the final boxed answer.`
MATH: Finish	`Finish verification now. If the candidate passes, close the thinking section yourself and output only the final answer as \boxed{...}. If one small local repair remains, do it here without restarting.`
GPQA: compact start	`Use compact structured reasoning: Route -> Work -> Choice map -> Check. Keep each field short. Do not restart; if uncertain, resolve one discriminating fact and finish.`
GPQA: loop	`Repetition detected. Stop re-deriving. Use the current facts to choose the best A/B/C/D after one choice-map check.`
GPQA: checkpoint	`Compress now: state the current candidate letter, one evidence line, and one check against the closest alternative. Do not restart or add background.`
LCB: Verify	`Check the chosen plan against constraints, input/output format, complexity, and one edge case. Repair only local issues.`
LCB: planned handoff	`Use the checked plan now. Do not add more analysis. Output only the complete Python solution.`
DS-MATH: loop/hard handoff	`{REASON} Use the best current candidate. Do not restart; only apply a local arithmetic/case check.`

P&B supplies planned subproblems with soft word allocations. TALE-EP replaces Vanilla's instruction with a soft token budget. The supplementary artifact provides the complete baseline templates.

E.2 CONTROLLER PSEUDOCODE

The pseudocode below summarizes the shared execution process. All output tokens are produced through the model decoding procedure. When a control action is active, the controller modifies the next-token logits according to the action before decoding.

One COTDOG completion

1. Initialize the output prefix, ordered events, event state, counters $g = c = 0$, and an empty action queue.
2. While generation is active and $g + c < B$, obtain the model's next-token logits z_t.
3. If the action queue is nonempty, adjust z_t according to the next control token specified by the active action, decode the next token from the adjusted logits using the same decoding rule, increment c, and update the action progress.
4. Otherwise, update observations from the visible prefix, counters, and event state.
5. Select the first active matching event. Apply its state updates and enqueue its action, if any.
6. If an action was enqueued, return to step 3. Otherwise, decode the next token from the unchanged logits and increment g.
7. Append the decoded token to the output prefix and update the observations and event state.
8. Stop on EOS, a policy completion action, or the common completion limit. Otherwise, return to step 2.

Control-action tokens are therefore generated through the same decoding interface as ordinary model tokens rather than being directly appended to the output. They remain in the generated prefix and consume the same output allowance. Each request has independent state, and an event cannot fire after it has been removed.

F QUALITATIVE EXAMPLES AND LIMITATIONS

The three DS-Qwen-32B MATH-500 pairs below correspond to the examples in Section 4.4. We selected them from matched temperature-zero outputs and independently checked the mathematical claims. They illustrate observed behavior rather than its frequency or the effect of any single controller action. Quoted text comes from the stored outputs. Headings, omissions, and unquoted equations are editorial. The supplementary artifact provides the complete outputs and checks.

Table 15: Selected qualitative examples. Tokens are total completion lengths. U and C denote unfinished and completed outputs.

Case (ID)	Method	Tokens	Finish	Correct	Observed Pattern
Overthinking (32)	Vanilla / CoTDoG	4,123 / 872	C / C	0 / 1	Correct result reversed
Underthinking (462)	Vanilla / CoTDoG	469 / 1,266	C / C	0 / 1	Required arithmetic omitted
Repetition (153)	Vanilla / CoTDoG	8,192 / 3,420	U / C	0 / 1	Repeated reasoning interrupted
Residual repetition (82)	Vanilla / CoTDoG	8,192 / 4,539	U / C	0 / 1	Rechecking remains

For ID 32, treating the three specified people as one block gives $5!\,3! = 720$. Vanilla first obtains 720 and later replaces it with 240, whereas CoTDoG retains the checked result. For ID 462, exact addition gives $12{,}659$ cents and a remainder of 59 cents. Vanilla closes its reasoning before performing the addition. CoTDoG performs and checks it. For ID 153, eight of the $\binom{9}{3} = 84$ triples are collinear. Vanilla repeats the correct probability without completing an answer. CoTDoG also begins to repeat, after which the controller starts the answer and the model returns the correct $2/21$.

ID 82 shows that repeated checking can remain under CoTDoG. Both outputs identify the correct extremal points for a slope of $3/2$. Vanilla repeats until the token limit, while CoTDoG revisits the same candidate several times before completing the correct answer.

Overthinking: a correct solution is explicitly rejected

MATH-500, ID 32 DS-Qwen-32B, greedy decoding

Problem. In how many ways can eight people sit around a round table if Pierre, Rosa, and Thomas must sit together? Seatings that differ only by rotation are identical.

Correct answer. $5! \times 3! = 720$ by arranging six units on the circle and three people within their block.

Vanilla

Thinking 3,907 tokens. Total 4,123.
Finish `stop`. Final answer incorrect.

Correct 720 ⟶ **wrong** 240
Correct solution explicitly rejected.

1. The correct result is already available

"Let me calculate that. 5! is 120, and 3! is 6. So, 120 multiplied by 6 is 720. Therefore, there are 720 ways."
≈3,500 more thinking tokens follow.

2. Reconsideration introduces a restriction

[Intervening checks omitted.]
"But wait, if we fix Pierre, then Rosa and Thomas can be on either side of him. So, Rosa can be to his left and Thomas to his right, or vice versa. That gives 2 arrangements for Rosa and Thomas relative to Pierre."

*"So, total arrangements would be 2 * 5! = 2 * 120 = 240. Wait, that's different from 720. Hmm, now I'm confused."*

3. A mistaken counterexample reinforces it

[Later, using a four-person example.]
"Wait, but according to the first approach, it's 6, which is clearly wrong because in reality, it's only 2 distinct arrangements."

4. The correct approach is rejected

[Further reconsideration omitted.]
"Therefore, the first approach is incorrect, and the second approach is correct."
"So, in the original problem, the correct answer is 240."

5. Normal completion with the wrong answer

[Remaining reasoning and final explanation omitted.]

`</think>` 240 **Wrong**

CoTDoG

Thinking 706 tokens. Total 872.
Finish `stop`. Final answer correct.

720 ⟶ **local check** ⟶ 720
Correct solution retained.

1. Plan: commit to the block construction

"Select one solution route now. Name the key formula, theorem, substitution, or cases to check. Keep the plan short."
Three people become one block, leaving six units.

2. Solve: carry out the chosen route

"Execute the chosen route. Do not restart the problem unless a contradiction is found; repair locally and keep derivation compact."
The calculation is typeset below.
$5! = 120, \quad 3! = 6, \quad 120 \times 6 = 720.$

3. A doubt is resolved locally

"Wait, but hold on. I think I might have made a mistake here. Let me double-check."
[The model rechecks both factors.]
"Hmm, no, because the block is treated as a single unit, and the internal arrangement is already accounted for by the 3!."
"So, I think 720 is correct."

4. Verify: retain the checked candidate

"Check the answer against the problem and repair only local errors."
The check uses six units, 5! circular orders, and 3! internal orders.
"Yes, that seems correct. I don't see any errors in the reasoning. So, the final answer should be 720."

5. Normal completion with the correct answer

[Final explanation omitted.]

`</think>` 720

Reading the comparison. Vanilla fixes Pierre in the middle of the block, omitting valid end positions. Its four-person check also undercounts. After reaching 720, it spends approximately 3,475 further thinking tokens before closing the thinking section and answering 240. CoTDoG's local verification preserves 720. Both responses end normally.

Figure 4: Overthinking example. Quotations reproduce the stored outputs. Blue text marks controller instructions, red marks the error, and the remaining annotations are editorial.

Underthinking: a plan is finalized before the calculation

MATH-500, ID 462 DS-Qwen-32B, greedy decoding

Problem. Linda, Sherry, June, and Connie earn \$27.47, \$35.23, \$37.37, and \$26.52. They pool their money and exchange as many coins as possible for whole-dollar bills. How many cents remain?

Correct answer. The total is $2,747 + 3,523 + 3,737 + 2,652 = 12,659$ cents, leaving a remainder of 59 cents.

Vanilla

Thinking 122 tokens. Total 469.
Finish `stop`. Final answer incorrect.

Plan only ⟶ thinking closes
The calculation has not been done.

1. The entire thinking section: only a plan

"First, I need to calculate the total amount of money that Linda, Sherry, June, and Connie earned by adding up their individual amounts."

"Next, I'll convert the total amount from dollars to cents to make it easier to work with the coins."

"Then, I'll determine how many whole dollars they can convert by dividing the total cents by 100 and taking the integer part of the result."

"After that, I'll subtract the value of the whole dollars from the total cents to find out how much money is left in cents."

"Finally, this remaining amount will be the answer to the problem."

2. Thinking closes before any addition

`</think>` **Stops here.**

3. The final response starts from a wrong sum

The calculation is typeset below.

$$27.47 + 35.23 + 37.37 + 26.52 = 126.60,$$
$$126.60 \times 100 = 12,660,$$
$$12,660 - 12,600 = 60.$$

Wrong total 126.60. Correct total 126.59.
The one-cent error propagates to the final answer.

4. Final answer

$\boxed{60}$ **Incorrect.**

CoTDoG

Thinking 1,192 tokens. Total 1,266.
Finish `stop`. Final answer correct.

Calculate ⟶ verify ⟶ answer
The total and remainder are checked.

1. Plan: identify the required computation

"First, I'll add up all the amounts each girl earned to get the total amount."

2. Solve: actually add the amounts

"Execute the chosen route. Do not restart the problem unless a contradiction is found; repair locally and keep derivation compact."
The model's calculation is typeset below.

$$27.47 + 35.23 = 62.70,$$
$$62.70 + 37.37 = 100.07,$$
$$100.07 + 26.52 = 126.59.$$

3. Continue to the remainder

"Checkpoint: continue the current solution path."
[Rest of the checkpoint text omitted.]
"This is essentially finding the remainder when 12,659 cents is divided by 100."

4. Verify: repeat the arithmetic before closing

"Check the candidate against constraints, signs, cases, and simplification. Fix only local errors, then prepare the final boxed answer."
"Let me double-check the calculations to ensure there are no errors."
All three additions are repeated. The total and 59-cent remainder are verified.

5. Finish with a checked answer

[Finishing cue and checks omitted.]
"Therefore, the amount left in cents after converting as many coins into bills as possible is 59 cents."

`</think>` $\boxed{59}$

Reading the comparison. All five paragraphs of Vanilla's reasoning are shown. They describe intended operations without adding the amounts. The final answer does contain arithmetic, but begins with an unchecked wrong total. CoTDoG uses more tokens to execute and verify that computation before finalization. This case illustrates the completion of missing work. Reasoning length alone does not establish the cause of the error.

Figure 5: Underthinking example. Quotations reproduce the stored outputs. Blue text marks controller instructions, red marks the error, and the remaining annotations are editorial.

Repetition: the controller ends an existing loop

MATH-500, ID 153 DS-Qwen-32B, greedy decoding

Problem. Choose three distinct points uniformly from a 3×3 grid of nine points. What is the probability that all three lie on one straight line?

Correct answer. There are $\binom{9}{3} = 84$ triples and eight collinear triples from three rows, three columns, and two diagonals. The probability is $8/84 = 2/21$.

Vanilla

Thinking 8,192 tokens. Total 8,192.
Finish `length`. No completed answer.

≈6,400 tokens of repeated text (i.e., 78% of the output), with no final answer.

1. The essential counting is correct

The model computes 84 possible triples.
"Adding these up: 3 horizontal + 3 vertical + 2 diagonal = 8 lines in total."
[Checks of other possible slopes omitted.]

2. Same answer ⇄ same doubt

One repeated block is shown below.
"Therefore, the probability is 8/84, which simplifies to 2/21."
"Wait, but hold on. I just thought, is 8 the correct number of lines? Because sometimes people might consider other lines, but in a 3x3 grid, I don't think so. So, I think 8 is correct."

⋮ ↻ ⋮
88 full cycles + one cut-off cycle
⋮ ⋮

The final copy is incomplete.
"Therefore, the probability is 8/84, which simplifies to 2/21."
"Wait, but hold on. I just thought, is 8 the correct number of lines?"
. . . **[cut off at 8,192 tokens]**

3. The loop consumes the remaining budget

Across the full output, the probability sentence occurs 91 times.
No `</think>` and no completed answer.

CoTDoG

Thinking 3,390 tokens. Total 3,420.
Finish `stop`. Final answer correct.

Loop ⟶ answer transition ⟶ answer
Correct $2/21$ in 3,420 tokens.

1. Structured reasoning obtains the counts

"Select one solution route now."
The model counts 84 possible triples and eight collinear triples.

2. A soft finish cue is followed by repetition

"Finish verification now."
[Soft cue shortened. Further reasoning omitted.]
"Therefore, the probability is 8/84, which simplifies to 2/21."
[The model repeatedly rechecks slopes, combinations, and the number of lines.]
"Wait, but I think I might have made a mistake in counting the number of lines. Let me recount:"
The final recount before the transition follows.
"- Horizontal lines: 3"

3. The loop is interrupted here

The same probability sentence occurs 10 times.
"Repeated reasoning detected. Use the best current candidate. Do not restart; only apply a local arithmetic/case check."
`</think>`
`Final answer:`
These transition tokens are part of the injected action.

4. The model completes the answer

"8/84 simplifies to 2/21."

$\boxed{\frac{2}{21}}$ Correct.

No further recounting follows the guard.

Reading the comparison. Vanilla repeats the two-paragraph block for 88 complete copies and one partial copy. This occupies approximately 6,407 tokens (i.e., 78.2% of its output). CoTDoG also enters a loop. The later reminder and reasoning closure interrupt it, after which the model returns the correct answer.

Figure 6: Repetition example. Quotations reproduce the stored outputs. Blue text marks controller instructions, and the remaining annotations are editorial.