Contrastive Test-Time Scaling (CTTS) - Extended Abstract

Motivation Test-time scaling is an incredibly powerful tool for increasing the performance of models Snell et al. (2024). Chain-of-Thought (CoT) reasoning is one of the most popular ways to achieve test test scaling Wei et al. (2023). Recently, a Stanford paper found that simply by appending a "wait" token whenever reasoning stops can prompt contrastive thinking, making the model check its answer and think for longer; this known as budget forcing Muennighoff et al. (2025). This simple strategy alongside efficient distillation of knowledge from a larger model allowed their relatively small open-source s1 model to outperform OpenAI o1-preview (a much larger model) by up to 27% on competition math benchmarks. However, spending more test-time compute on a problem comes with diminishing returns. The more tokens a model spends thinking, the more likely is it to accumulate errors and stray from the true answer. This phenomenon is know as "overthinking" and it has been shown that spending a lot of tokens during test time can lead to impaired performance Peng et al. (2025). The budget forcing methodology used in the s1 paper relies on heuristics. This means that the model is prompted to be a critic of its own answer only when it reaches one after long reasoning. LLMs tend to forget about context in the middle of the input especially when the chain of thought becomes longer Liu et al. (2023). This means if we only prompt the model to criticize its own reasoning only after it finishes reasoning, the model may fail to spot mistakes it made earlier in reasoning, leading to a waste of tokens and potentially overthinking. Our paper aims to trains a reinforcement learning model that can interrupt reasoning to prompt contrastive/critic thinking by appending wait tokens when it is most beneficial to do so. This can lead to more effective correcting of mistakes by the model and more efficient use of tokens and thus better test-time scaling.

Method We formulate strategic intervention as a Reinforcement Learning problem where a lightweight actor-critic agent controls a larger LLM's reasoning process. Using a limited budget of "wait" tokens, the agent maximizes the worker's problem-solving accuracy while minimizing number of tokens. The agent relies on CoT-step-invariant features that has LLM's final hidden state given the last 2048 tokens in the CoT, remaining token budget ratio that resets after 3 "waits" used, used "wait" budget ratio (resets every 3 waits), tokens generated since last wait, and confidence and entropy metrics (calculated w/ sliding windows). This position-agnostic approach enables the agent to learn generalizable intervention strategies instead of memorizing token positions to use wait tokens.

Implementation We implement a high-throughput training framework featuring a dual-model setup: a vLLM instance handles rapid batch text generation while a HuggingFace model performs streamlined hidden state extraction. The policy network learns when to use "wait" tokens from the GSM8K dataset through Proximal Policy Optimization with Generalized Advantage Estimation. We chose GSM8K since it's an easy dataset. Thus, it is one that models can easily overthink on given enough tokens and good to test our efforts to reduce overthinking and improve test time scaling.

Results CTTS consistently outperforms heuristic test-time scaling across a range of token budgets (with max accuracy up to 74% for CTTS vs. up to 64% for naive), especially at high tokens (3000+) where overthinking reduces model performance. We also found that tokens spent on contrastive reasoning more efficiently increases accuracy than tokens spend on non-contrastive reasoning. Despite limited training data, CTTS demonstrates that contrastive thinking is more impactful than extending reasoning, improving LLM accuracy in overthinking-prone settings.

Discussion Our results demonstrate that adaptive contrastive prompting with CTTS reduces overthinking in LLMs, leading to higher reasoning accuracy and more efficient use of compute. This is particularly impactful for smaller models that benefit most from test-time scaling. While our approach shows strong promise, it also has limitations, such as coarse decision intervals due to computational constraints and inefficiencies in accessing LLM hidden states. Despite this, the success of CTTS provides proof of concept for more resource-aware reasoning in LLMs.

Conclusion In this work, we introduced Contrastive Test-Time Scaling (CTTS), a reinforcement learning-based framework that improves LLM reasoning by reducing overthinking. CTTS consistently outperformed heuristic baselines at high token counts, making it more impactful than naively extending generation length. Future work should explore more frequent agent intervention and scaling to larger models and datasets to further validate and generalize CTTS.

Contrastive Test-Time Scaling

Nattaput (Gorn) Namchittai
Department of Computer Science
Stanford University
gorn@stanford.edu

Julia (Juli) Huang
Department of Computer Science
Stanford University
julih@stanford.edu

Abstract

Recently, a Stanford paper found that simply by appending a "wait" token whenever reasoning stops can prompt contrastive thinking, making the model check its answer and think for longer; this is known as budget forcing Muennighoff et al. (2025). This simple strategy alongside efficient distillation of knowledge from a larger model allowed their relatively small open-source s1 model to outperform OpenAI o1-preview (a much larger model) by up to 27% on competition math benchmarks. Although an effective way to scale models' performance during test-time, budget forcing relies on heuristics that can lead to inefficient resource utilization and potentially critical reasoning mistakes earlier in the chain-of-thought. We present Contrastive Test-Time Scaling (CTTS), a novel approach to efficient test-time scaling, employing a compact reinforcement learning agent that learns optimal allocation of limited "wait" token budgets to enhance worker LLM performance.

1 Introduction

Test-time scaling is an incredibly powerful tool for increasing the performance of models. Test-time compute is also thought to be more beneficial to smaller models, many of which are open-source. Thus, the study and development of effective test-time scaling is not only significant to the improvement of LLMs generally but is also important to the democratization of AI Snell et al. (2024). Chain-of-Thought (CoT) prompting facilitates emergent problem-solving by instructing models to generate intermediate reasoning steps Wei et al. (2023). It is one of the most popular ways to increase model performance through test-time compute (by getting the model to think for longer) and is widely used in reasoning models.

However, early or repeated logical or factual errors can corrupt the entire reasoning chain, resulting in incorrect conclusions. The solution used by many in machine learning is upscaling. Whether that be in compute, data, or time. Similarly, in test time scaling, past solutions (discussed in the related works section) resort to sacrificing a lot of compute during test-time in order to achieve the highest performance achievable. A lot of these techniques are also based on heuristics, which further leads to even more wasted compute.

However, Test-time scaling is also bottlenecked by another problem: "overthinking". Spending more test-time compute on a problem comes with diminishing returns. The more tokens a model spends thinking, the more likely it is to accumulate errors and stray from the true answer. It has been shown that spending a lot of tokens during test time can lead to impaired performance Peng et al. (2025).

We propose Contrastive Test-Time Scaling (CTTS), a framework replacing static test-time scaling heuristics with a reinforcement learning (RL). We aim to develop a reinforcement learning agent that is able to alleviate the "ovethinking" problem. By prompting the reasoning model to criticize its own reasoning (i.e. prompt contrastive thinking) and learning when it is best to do so, we could achieve a better accuracy with less compute at test-time. We're interested in seeing if tokens spent on contrastive thinking can be more efficient in getting us to the correct answers, and if we were to learn

to adaptively allocate these points of contrastive thinking, would that lead to more efficient test-time scaling compared to a heuristic-based allocation. Finally, we aim make the model and its test-time scaling capabilities generalizable to different token budgets. We aim to train an test our model on GSM8K, which consists of easy grade-school math questions, which are prime candidates for testing overthinking, as it should not take that many tokens to solve the problems effectively, and spending a lot of tokens on this dataset is likely to lead to "overthinking". This will allow us to evaluate if CTTS helps with "overthinking" and test-time scaling or not.

We hypothesize that our learned agent will outperform heuristic methods, achieving higher accuracy and efficiency by optimally allocating intervention resources, advancing autonomous, resource-aware reasoning systems.

2 Related Work

Prior studies on test time scaling rely on high-compute methodologies that heavily rely on heuristics.

Tree of Thoughts is a test-time scaling technique that uses explicit tree search over partial CoT sequences, pruning low-value branches via value models. It achieves strong performance on hard reasoning tasks but incurs exponential compute with branching factors and requires careful search heuristics. The model assumes that by generating several best next steps, one would lead to the right answer, which is not necessarily true if error is already present in the reasoning so far. The model only samples for the next best steps and does not explicitly check if its reasoning so far is correct. Yao et al. (2023)

Multi-agent/ensemble reasoning models rely on the assumption that if we generate multiple reasoning chains, the majority or weighted majority will be correct. The heuristic here is that we assume that the agents on (weighted) are reliable for the problem in question. Critics studied in a multi-agent context tend to also be explicit critics that are prompted to criticize reasoning, which can be expensive compute-wise and adds complexity. We are interested in implicit critic prompting described later in this section that is a lot mroe simple and is effective. Xu et al. (2023).

Stanford researchers recently demonstrated that budget forcing (simply appending "wait" tokens when reasoning pauses) can implicitly trigger contrastive thinking and double checking of their answers, enabling their s1-32B model to outperform OpenAI's o1-preview by up to 27% on competition math benchmarks despite being significantly smaller Muennighoff et al. (2025).

The budget forcing methodology used in the s1 paper relies on heuristics. The model is prompted to be a critic of its own answer only when it reaches one after long reasoning. LLMs tend to forget about context in the middle of the input, especially when the chain of thought becomes longer Liu et al. (2023). This means if we only prompt the model to criticize its own reasoning only after it finishes reasoning, the model may fail to spot mistakes it made earlier in reasoning, leading to a waste of tokens and potentially overthinking.

Our paper aims to train a reinforcement learning model that can interrupt reasoning to prompt implicitly contrastive/critic thinking by appending wait tokens when it is most beneficial to do so. This can lead to more effective correcting of mistakes by the model and more efficient use of tokens and thus better test-time scaling.

To our knowledge this is the first study on implicit and adaptive contrastive/critic prompting in the context of test time scaling.

3 Method

Dataset: We train our reinforcement learning on the GSM8K dataset so that it can learn ho to best allocate wait tokens to achieve the highest accuracy without overthinking. We also hold off a validation split from the training set. Additionally we use GSM8K for testing; it's a good candidate for our study as it is an easy dataset, making overthinking possible even with not that many reasoning tokens. This allows us to see the effects of overthinking at lower token counts compared to harder datasets.

Model: As for the reasoning model, we picked s1.1-1.5B. The model is a strong reasoning model (a smaller updated version of the one in the s1 paper) that scales during test time depending on how

many wait tokens we append to it. The model's inherent reasoning ability as well as the level of control and simplicity it provides not only makes it a good baseline but also a good base model to use for our RL-controlled chains-of-thought. We chose the smaller model as fast and efficient generation of CoTs and our input features for our RL agent is compute and memory-intensive so using a smaller model helps make things more feasible. We will be referring to this model as s1.1 throughout this paper.

Baselines We test scaling of the s1.1 model without appending wait tokens as a baseline to examine the difference between using contrastive thinking and not. We can do this by increasing the number of tokens the model is allowed to generate and prompting an answer if it does not come to a conclusion within the limit. Consistent with previous studies in S1.1 test-time scaling when using wait tokens, we use a baseline that uses wait tokens after a set maximum number of tokens. We let the model generate a maximum of around 2000 tokens before appending a wait token. Allowing to generate more could lead to us exceeding the max token limit of the LLM. After n wait tokens, we allow the model to generate for at model for at most another roughly 2000 tokens before we prompt for an answer.

Hou et al. (2025). The number of wait tokens we append scales with the number of tokens the model uses in reasoning.

We formalize the intervention problem as a finite-horizon Markov Decision Process (MDP) under the assumption that the state at time t encapsulates sufficient information for optimal decision-making. The MDP components are defined as follows.

Steps: Each step we prompt the LLM to answer a question from our dataset; we allow it to generate up to 512 tokens, after which the RL agent is prompted to choose between continuing the reasoning or appending a wait token which would prompt contrastive reasoning, leading it to check its reasoning so far. We pick 512 tokens because it is long enough to allow the model to make considerable progress in reasoning but not too long that the RL model has limited control over the reasoning chain. It is not too short that we have to compute features too often which can be expensive given our choice of features explained below. If the model reaches 2048 tokens (which we refer to as a safety threshold) without using a wait token, it it forced to use one and the count is reset. This is to keep the model within the max token limit of the LLM and keep things consistent with the baseline since we want to see if an RL agent can learn to interrupt natural reasoning effectively in a given number of tokens as supposed to the naive strategy of appending a wait at the end of a long chain of reasoning like in the baseline.) Once we use up all our tokens, the model is allowed to generate for at most another 2000 tokens approximately (so that we are consistent with the baseline) before it is prompted to give an answer. In testing, we can "revive" the model by replenishing its tokens to the set starting number and restart the loop for steps. After specific number of "revives" we can then allow it to generate for at most another 2000 tokens approximately (so that we are consistent with the baseline) before it is prompted to give an answer. This methodology allows us to train a modle that can scale with different test time budgets.

State Space ($\mathcal{S}$): The state $s_t \in \mathcal{S}$ at timestep t constitutes a feature vector engineered to exhibit reasoning-position-invariance, thereby ensuring policy generalizability across diverse reasoning stages. In our experiments, we define a reasoning cycle to be the start of reasoning given 3 wait tokens to append and the end of reasoning when the 3 tokens are used up. After which we can "revive" the model by giving it 3 more wait tokens and continuing or stopping reasoning and prompting for an answer. This can be repeated as many times given a test-time budget. Using 3 tokens per cycle simplifies training while the "revive" methodology allows us to test on different test-time budgets without having to train separate models for different budgets. Formally, the state is represented as:

$$s_t = (h_t, r_{b,t}, r_{w,t}, r_{l,t}, e_t, c_t) \tag{1}$$

where:

- h_t: The terminal hidden state vector from the LLM. This obtained by passing in the last 2048 tokens from the reasoning so far into the LLM. The idea is that this encapsulates the current understanding of the problem by the LLM.
- $r_{b,t}$: Remaining token budget ratio within the current reasoning cycle
- $r_{w,t}$: Wait budget utilization ratio within the current cycle
- $r_{l,t}$: Token generation ratio since the last wait action relative to safety threshold

- e_t: Average token entropy computed over a sliding temporal window
- c_t: Exponentially weighted moving average of token probabilities over a sliding window

Note that all these states are designed to be essentially invariant to the absolute states of the CoT. I.e. they are all relative features that only depend on the most part on a sliding window of previous states. This is so that we can generalize the model to larger token counts and longer reasoning without having to train a separate model for that. The states give the RL model a rich set of information on what the state of the LLM is with respect to the question in the recent steps of reasoning and also information about the environment, such as ratio of tokens and waits and how close it is to using the next wait. These information-rich states should be enough for the RL agent to make informed decisions on what to do.

Action Space ($\mathcal{A}$): The agent selects actions $a_t \in \mathcal{A}$ from a discrete binary set:

$$\mathcal{A} = \{\text{continue}, \text{wait}\} \tag{2}$$

The wait action is masked when the agent has exhausted its allocated wait token budget for the current cycle.

Reward Function ($\mathcal{R}$): The reward mechanism incentivizes both correctness and computational efficiency through a dense signal. The terminal reward R_{final} is defined as:

$$R_{\text{final}} = \begin{cases} +1.0 & \text{if answer is correct} \\ -1.1 & \text{if answer is incorrect} \end{cases} \tag{3}$$

Additionally, step-wise a token penalty p_t promotes efficiency:

$$p_t = -0.1 \cdot \frac{N_{\text{tokens}}}{1000} \tag{4}$$

where N_{tokens} represents tokens generated in the preceding step. The total reward is just $R_{final} + \sum p_t$.

The slightly larger magnitude of the penalty for incorrect answers makes the agent risk-averse, encouraging it to use its waits to secure a correct answer rather than rushing and guessing. Since we also want the model to be efficient with token usage, the penalty for token generated helps the agent learn to use waits earlier if it helps it get the right answer. Using waits earlier tend to lead to fewer tokens being used overall Muennighoff et al. (2025)

Policy: The agent's policy $\pi(a_t|s_t)$ and value function $V(s_t)$ are jointly parameterized by a unified actor-critic architecture, facilitating shared feature representation and enhanced sample efficiency.

Input Processing: Ratio-based features $(r_{b,t}, r_{w,t}, r_{l,t})$ undergo discretization into N_{bins} bins, followed by processing through dedicated embedding layers (E_b, E_w, E_l). This is so that these the signal from these feautures are not overwhelmed by the features from the high-dimensional hidden state. Similarly, the high-dimensional hidden state h_t is transformed via linear projection W_p with ReLU activation for dimensionality reduction so we maintain only the most important signals and don't overfit to noise or overwhelm the other features too much.

Feature Integration: Processed inputs are concatenated into a unified feature vector:

$$\mathbf{x}_t = \text{concat}(\text{ReLU}(W_p h_t), E_b(b(r_{b,t})), E_w(b(r_{w,t})), E_l(b(r_{l,t})), e_t, c_t) \tag{5}$$

where $b(\cdot)$ denotes the binning transformation.

Actor-Critic Architecture: The combined feature vector $\mathbf{x}_t$ feeds into two specialized multi-layer perceptron heads:

- **Actor**: Outputs action logits over $\mathcal{A}$
- **Critic**: Produces scalar value estimate $\hat{V}(s_t)$

Both of these have the same number of hidden dimensions and have one hidden layer. We kept these networks small as the features we feed into them are already information-rich rich and we may risk overfitting if we make the policy net too big.

The dimensions of all the actual projections/embeddings are all summarized in the experimental setup section.

Training: We employ Proximal Policy Optimization (PPO) for policy network training, leveraging its superior stability and sample efficiency characteristics through constrained policy updates.

Advantage Estimation: We utilize Generalized Advantage Estimation (GAE) for low-variance advantage computation:

$$\hat{A}_t = \sum_{k=0}^{T-t-1} (\gamma\lambda)^k \delta_{t+k} \tag{6}$$

where $\delta_t = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$, with γ as the discount factor and λ as the GAE parameter.

Objective Function: PPO optimizes a clipped surrogate objective. Let $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ represent the probability ratio. The policy loss is:

$$L^{\text{CLIP}}(\theta) = -\hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{7}$$

The complete loss function combines policy loss, value function loss L^{VF}, and entropy regularization L^{S}:

$$L(\theta) = L^{\text{CLIP}}(\theta) + c_1 L^{\text{VF}}(\theta) - c_2 L^{\text{S}}(\theta) \tag{8}$$

PPO is selected over alternative policy gradient methods due to its superior stability and sample efficiency which is crucial for stochastic environments like this where we can get very unstable signals. The trust region optimization inherent in the clipping mechanism proves essential for complex policy networks operating in sophisticated environments. Sample efficiency is important here too as we have a limited dataset.

Implementation Details: To address the computational requirements of high-throughput generation and internal state access, we implement a dual-model architecture. A vLLM instance provides optimized batched generation for trajectory collection, while a separate HuggingFace model enables efficient hidden state extraction.

4 Experimental Setup

We trained our model using the following hyperparameters.

Parameter	Value	Justification
Learning Rate	1.0×10^{-5}	Ensures stable convergence with Adam optimizer
Discount Factor (γ)	0.99	Standard value balancing immediate/future rewards
GAE Parameter (λ)	0.95	Optimal bias-variance trade-off
Clip Epsilon (ϵ)	0.1	Conservative clipping for stable updates
Entropy Coefficient	0.02	Promotes exploration, prevents premature convergence
PPO Epochs	4	For sample efficiency
Feature Bins	51	Sufficient granularity for ratio-based features
Hidden Dimension	1536	Matches base LLM architecture
Projection Dimension	256	Efficient hidden state dimensionality reduction
Embedding Dimension	16	Prevents binned features from being too insignificant

Table 1: Hyperparameter specifications and rationale

During training, we set the wait token budget to be 3. Since the input features are only dependent on recent states, once we run out of the token budget during testing, we can "revive" the model by giving it 3 more tokens and resetting the metrics involved in defining its state, allowing the RL agent to continue allocating the wait budget. This approach allows our model to generalize over multiple wait budgets (which tend to correspond to tokens used) and thus allow us to see how our model scales with test time compute.

We use 90% of the GSM8K train split during training and we train for roughly 450 epochs, selecting the model that achieved the best validation accuracy for our final tests. 10% of the train split is reserved for validation and each validation we use 128 problems from this split.

The relevant hyperparameters above are also used in testing on the GSM8K test split.

We evaluate how our model performs over different number of wait tokens assigned at test time, which correspond to the number of tokens used. We record the number of tokens used in each experiment and report it with respect to accuracy. This allows us to compare it with our baselines to see if our method increases token efficiency during test time and if tokens spent on contrastive thinking can be more efficient than non-contrastive tokens.

5 Results

5.1 Quantitative Evaluation

Avg Tokens (Range)	CTTS Accuracy (%)	s1.1 Baseline Accuracy (%)
3052.81 – 3076.50	71.88	63.31
4313.81 – 4471.20	72.66	64.90
6136.09 – 5851.52	72.66	64.37
7749.67 – 7532.98	67.19	63.38
9310.11 – 9331.86	74.22	64.75
10863.46 – 11284.61	71.09	65.73

Table 2: Accuracy comparison between CTTS and s1.1 naive wait baseline, with Avg Token ranges

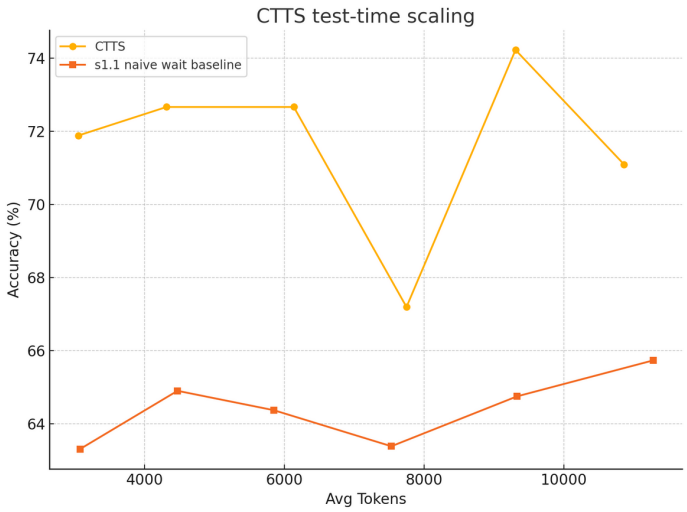

Figure 1: Accuracy vs average number of tokens on naive wait baseline and CTTS

30000 to 11200 tokens are likely to be overkill for grade school math problems in GSM8K. Thus, we are more than certain the base LLM will be overthinking at these token ranges. The graph and table above compare CTTS to the baseline that naively appends wait to the end of each reasoning step (at most 2048 tokens).

The fact that CTTS consistently maintains a higher accuracy at these token ranges is strong evidence that our RL agent is able to more efficiently use wait tokens to reduce overthinking and thus allowing the model to achieve a higher accuracy at high token counts than the naive heuristic approach used in previous studies. This also suggests that CTTS effectively learns when to wait and critize its own reasoning effectively.

However, we do see that CTTS fluctuates in accuracy a lot more than the baseline. This could be due to the fact that our RL agent is not completely perfect as it was only trained on a relatively small dataset and may not generalize as well yet. This fact is especially relevant in our setting due to the stochasticity of LLM generation, and high generalizability in the context of LLMs tend to require a lot more data. Also the fact that the RL agent only gets to control the CoT after at most 512 tokens means the model doesn't have fine grained control over the reasoning. This could by by chance lead

to slightly impaired performances in some cases, meaning our agent can sometimes be less effective if the reasoning during each 512 max tokens passes an important intervention point as that point can be further behind in the CoT and implicit critic prompting via appending waits may not allow the model to catch the mistake further back. The fact that the RL agent has somewhat crude control over the CoT would explain why we get more fluctuation in the CTTS graph.

Nevertheless, the fact that CTTS is able to achieve a consistently higher accuracy than the baseline proves the validity of our approach.

Number of Revives	Avg Tokens	Number of Wait Tokens
0	3052.81	3
1	4313.81	6
2	6136.09	9
3	7749.67	12
4	9310.11	15
5	10863.46	18

Table 3: CTTS: Number of Revives vs Avg Tokens and Wait Tokens

Wait Tokens	Avg Tokens
1	3076.50
2	4471.20
3	5851.52
4	7532.98
5	9331.86
6	11284.61

Table 4: s1.1 Baseline: Wait Tokens vs Avg Tokens

Here we also see that CTTS is able to use more wait tokens (and thus prompt contrastive thinking) more often without extending reasoning for as long as the baseline.

3 wait tokens in CTTS is effectively used in roughly the number of tokens that would be used per 1 token in the baseline. Coupled with the fact that CTTS achieved higher accuracy at high token ranges, this is strong evidence that prompting the model to criticize its reasoning more often and at the right time is essential to improving test-time scaling and reducing overthinking. As a result, the model with CTTS achieves higher accuracy even when generating many tokens.

5.2 Qualitative Analysis

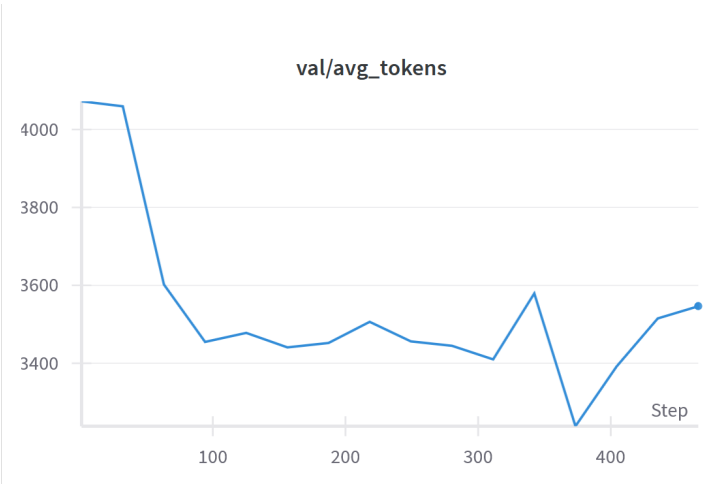

Figure 2: Average number of tokens outputted by model for validation set over epochs

Here we see that the number of average tokens used during training, coupled with the fact that our method increased test-time scaling efficiency, this suggests that our reward model worked in guiding the model to use its tokens more efficiently and also proves that the heuristic of appending wait tokens at the end of long reasoning is not an optimal strategy.

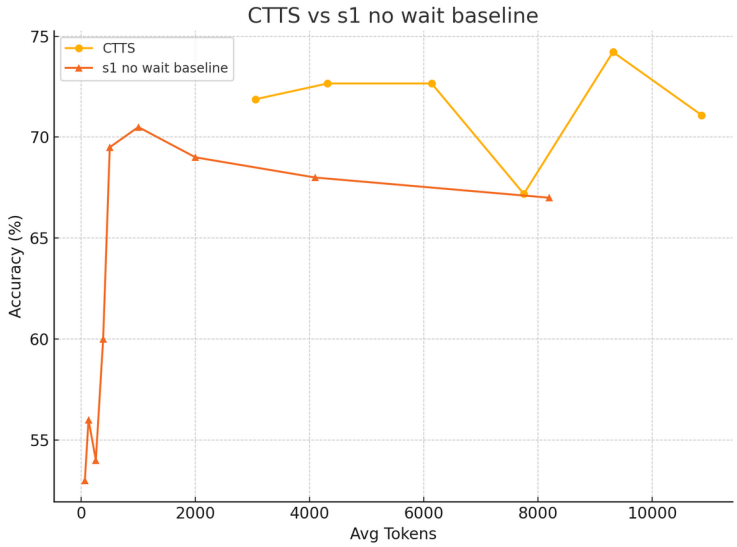

Figure 3: Accuracy vs average number of tokens on naive wait baseline and CTTS

Here we compare CTTS to the s1 baseline that is allowed to generate for longer without using any wait tokens (and is thus not prompted to think contrastively). We call this the no wait baseline. We see that performance degrades at high token counts for the no wait baseline, which is a classic sign of overthinking (which also proves our initial intuition earlier that overthinking will be happening at token counts over 3000 and that GSM8K is a good dataset to test or overthinking).

The fact that CTTS can achieve consistently higher accuracy at high token counts compared to a baseline that is not prompted to criticize its reasoning implicitly through wait tokens prove that tokens spent on contrastive thinking can help increase accuracy more sore than just tokens spent on non-contrastive thinking alone.

6 Discussion

Our results above show that adaptive and implicit contrastive/critic prompting can address overthinking, allowing models to achieve a higher accuracy ceiling when scaling with test-time compute. Our results also suggest that tokens spent on contrastive thinking can be more efficient in getting us the right answers compared to just tokens spent on purely non-contrastive thinking. We also show that using our specific input features designed to be invariant to very old states, our RL agent is able to learn to interrupt reasoning at the right times and prompt contrastive thinking to increase accuracy over the heuristic method used in previous studies. Our study serves as effective proof that adaptive and implicit contrastive/critic prompting can help models overcome the accuracy bottleneck that results from overthinking and thus contributes to the tools we have in increasing the performance of LLMs. AS mentioned earlier this is especially important for small LLM (many of which are open source) that benefit msot from test-time compute. Thus our study also contributes to the democratization of AI and the development of better open-source models.

There are, however, limitations to our study. Firstly, we adopted the crude approach of only allowing the RL agent to make a decision after the base LLM is allowed to generate at most 512 tokens. This decision was mostly done for the sake of reducing the cost of training as stopping every token to get the hidden state of the LLM will be very expensive. It would also make the RL problem much harder and less feasible given our timeframe and dataset. Nevertheless the fact that we got good results is a good proof of concept that our methodology worked.

The training required a lot of compute and getting streamlining the training was a large obstacle. CoT generation is expensive so we used an optimized approach where we used vLLM to generate multiple CoTs concurrently. Our decision to only call the RL agent every few states is a result of the fact that vLLM doesn't easily let us see the hidden state of the LLM and so we have to pass the truncated CoT from it into a separate model using the HuggingFace library instead which is slower and less optimized and much more memory-inefficient (making it the bottleneck in our training).

7 Conclusion

CTTS helps improve reasoning accuracy and efficiency in LLMs by address overthinking. We demonstrated that CTTS consistently outperforms heuristic test time scaling methods across different token budgets at high token counts (3000+), where overthinking usually impairs performance in oru chosen dataset. To achieve this, CTTS dynamically allocates wait tokens to encourage more effective contrastive reasoning at the correct moments instead of naively extending reasoning length. Despite usuing limited training data, CTTS was able to achieve higehr average accuracy while using more tokens. Since tokens spent on contrastive thinking are more impactful than those spent on non-contrastive reasoning, adaptive contrastive prompting leads to higher test time performance especially in environments where overthinking is prevalent (i.e. simpler tasks/data). Overall, CTTS provides a more resource efficient LLM reasoning system that can help future models achieve a higher accuracy ceiling previously unachieveable because fo overthinking.

Future studies should try to allow the RL agent to be called more often (i.e. generate for fewer max tokens before caling the RL agent). This could allow for more fine grained control of the reasoning chain and thus less variance in results. This however, could require more data to train as the states we encounter would be more diverse, meaning we would need more examples for an effective agent.

In addition, we could train and benchmark with bigger models such as s1.1-7B or s1.1-32B and see if our contrastive test-time scaling Sresults generalize to larger models and larger datasets. Our method could potentially help these models achieve a higher accuracy ceiling on very hard datasets.

8 Team Contributions

- **Group Member 1:** Nattaput (Gorn) Namchittai: Project idea, algorithm/reward design/optimization, training of models, model benchmarking, paper writing
- **Group Member 2:** Juli Huang: Data processing and data pipeline development, dataset research, model benchmarking, paper writing, data presentation

Changes from Proposal Our original hypothesis was that increasing chain-of-thought consistency by optimizing reasoning trace alignment between multiple agents could potentially lead to less overthinking and a higher reasoning accuracy.

Our new hypothesis is that learning an optimal allocation of "wait" tokens through reinforcement learning can improve LLM performance at test time by encouraging more deliberate and contrastive reasoning, without relying on fixed heuristics, thus reducing overthinking and increasing answer accuracy more efficiently.

References

Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. 2025. ThinkPrune: Pruning Long Chain-of-Thought of LLMs via Reinforcement Learning. `https://arxiv.org/abs/2504.01296` _eprint: 2504.01296.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. `https://arxiv.org/abs/2307.03172` _eprint: 2307.03172.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. `https://arxiv.org/abs/2501.19393` _eprint: 2501.19393.

Keqin Peng, Liang Ding, Yuanxin Ouyang, Meng Fang, and Dacheng Tao. 2025. Revisiting Overthinking in Long Chain-of-Thought from the Perspective of Self-Doubt. https://arxiv.org/abs/2505.23480 _eprint: 2505.23480.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. https://arxiv.org/abs/2408.03314 _eprint: 2408.03314.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://arxiv.org/abs/2201.11903 _eprint: 2201.11903.

Zhenran Xu, Senbao Shi, Baotian Hu, Jindi Yu, Dongfang Li, Min Zhang, and Yuxiang Wu. 2023. Towards Reasoning in Large Language Models via Multi-Agent Peer Review Collaboration. https://arxiv.org/abs/2311.08152 _eprint: 2311.08152.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. https://arxiv.org/abs/2305.10601 _eprint: 2305.10601.

A Additional Experiments

In the past, we attempted to apply CTTS on the s1.1-3B model and evaluated its performance on the AIME math 2024 dataset in a zero-shot setting.

The reinforcement learning component was trained separately using the DAPO-Math-17k-Processed dataset.

However, back then instead of prompting implicit critic thinking during reasoning through wait tokens, we used a more explicit approach where the model is prompted to reason on the last n steps of reasoning and criticize accordingly. However doing so is much more token-inefficient compared to our new method. The high complexity of the AIME dataset also meant that the model isn't able to perform well from the start. The DAPO dataset used to train is also difficult for our base model which meant it might not learn very much since it will not get correct answers often.