LightThinker: Thinking Step-by-Step Compression

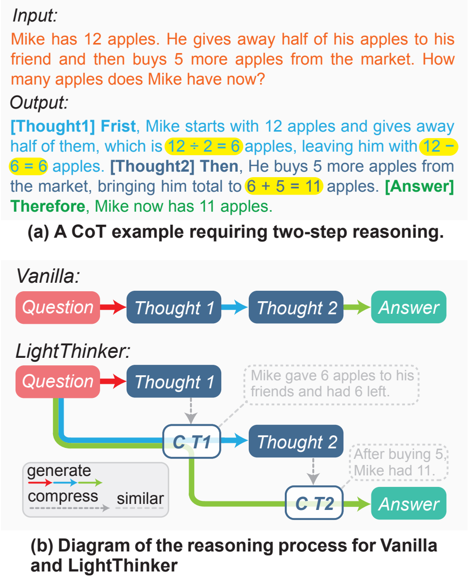
Jintian Zhang[♠♡*], **Yuqi Zhu**[♠♡*], **Mengshu Sun**[♣♡], **Yujie Luo**[♠♡],
Shuofei Qiao[♠♡], **Lun Du**[♣♡], **Da Zheng**[♣♡], **Huajun Chen**[♠♡], **Ningyu Zhang**[♠♡†]
[♠]Zhejiang University [♣]Ant Group
[♡]Zhejiang University - Ant Group Joint Laboratory of Knowledge Graph
{zhangjintian,zhangningyu}@zju.edu.cn
https://github.com/zjunlp/LightThinker

Abstract

Large language models (LLMs) have shown remarkable performance in complex reasoning tasks, but their efficiency is hindered by the substantial memory and computational costs associated with generating lengthy tokens. In this paper, we propose **LightThinker**, a novel method that enables LLMs to dynamically compress intermediate thoughts during reasoning. Inspired by human cognitive processes, LightThinker compresses verbose thought steps into compact representations and discards the original reasoning chains, thereby significantly reducing the number of tokens stored in the context window. This is achieved by training the model on when and how to perform compression through data construction, mapping hidden states to condensed gist tokens, and creating specialized attention masks. Additionally, we introduce the Dependency (Dep) metric to quantify the degree of compression by measuring the reliance on historical tokens during generation. Extensive experiments on four datasets and two models show that LightThinker reduces peak memory usage and inference time, while maintaining competitive accuracy. Our work provides a new direction for improving the efficiency of LLMs in complex reasoning tasks without sacrificing performance.

1 Introduction

Recent advancements in Large Language Models (LLMs) have demonstrated their remarkable capabilities in complex reasoning tasks (Zhao et al., 2023; Azaria et al., 2024). As research in this domain progresses, the reasoning patterns of these models have gradually evolved from "fast thinking" to "slow thinking". This transition is exemplified by methods such as Chain-of-Thought (CoT) (Wei et al., 2022) prompting, which enhances reasoning by breaking down complex problems into sequential sub-steps. Building on this, the *o1-like thinking mode* (Jaech et al., 2024; Qwen., 2024; DeepSeek-AI et al., 2025) introduces multiple reasoning abilities such as trial-and-error, backtracking, correction, and iteration, further improving the success rate of models in solving complex problems. However, this performance improvement comes at the cost of generating a large number of tokens (Wu et al., 2024b). Given that current LLMs are predominantly based on the Transformer architecture (Vaswani et al., 2017), the computational complexity of the attention mechanism grows quadratically with the context length, while the storage overhead of the KV Cache increases linearly with the context length. For example, in the case of Qwen32B (Yang et al., 2024), when the context length reaches 10^4, the KV Cache occupies


Input:
Mike has 12 apples. He gives away half of his apples to his friend and then buys 5 more apples from the market. How many apples does Mike have now?
Output:
[Thought1] Frist, Mike starts with 12 apples and gives away half of them, which is 12 ÷ 2 = 6 apples, leaving him with 12 − 6 = 6 apples. **[Thought2] Then**, He buys 5 more apples from the market, bringing him total to 6 + 5 = 11 apples. **[Answer] Therefore**, Mike now has 11 apples.

(a) A CoT example requiring two-step reasoning.

(b) Diagram of the reasoning process for Vanilla and LightThinker

Figure 1: (a) A CoT case. Tokens highlighted in yellow represent critical reasoning tokens, while the remaining tokens primarily ensure fluency. Humans typically only write the yellow parts when solving this problem. (b) Comparison of reasoning between Vanilla and LightThinker. "C Ti" denotes the i-th compressed thought representation, and we illustrate the semantic information potentially expressed after compression.

[*] Equal Contribution.
[†] Corresponding Author.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 13307–13328
November 4-9, 2025 ©2025 Association for Computational Linguistics

a space comparable to the model itself. Consequently, the increase in token generation leads to a sharp rise in memory overhead and computational costs, severely limiting the practical efficiency of LLMs in long-text generation and complex reasoning tasks.

To mitigate this issue, two main approaches have been proposed, primarily differentiated by their intervention requirements during inference. The first category requires no additional intervention during inference, achieving efficiency through prompt engineering (Han et al., 2024; Ding et al., 2024; Nayab et al., 2024) or specialized training (Liu et al., 2024a; Kang et al., 2024; Arora and Zanette, 2025; Luo et al., 2025; Cheng and Durme, 2024; Hao et al., 2024) to guide LLMs in generating fewer or even zero (Deng et al., 2023, 2024) intermediate tokens during reasoning. The second category operates through real-time token-by-token intervention during inference (Zhang et al., 2023; Chen et al., 2024), reducing memory usage by selectively retaining important parts of the KV Cache while discarding less critical ones. However, both approaches face distinct challenges: the first typically requires careful data construction and iterative refinement, while the second introduces substantial inference latency due to the computational overhead of token-wise importance assessment.

In this work, we propose a new approach by training LLMs to dynamically compress historical content during reasoning. Our motivation stems from two observations: 1) Tokens generated by the LLM serve dual purposes: ensuring linguistic fluency and facilitating actual reasoning (as highlighted in yellow in Fig. 1(a)), making compression feasible. 2) When humans solve problems similar to the one in Fig. 1(a), they typically write only key steps (highlighted in yellow), while storing the rest of the thought process mentally.

Based on these insights, we introduce LightThinker, a method that dynamically compresses intermediate thoughts during generation. As illustrated in Fig. 1(b), after generating a lengthy thought step (e.g., `Thought i`), it is compressed into a compact representation (e.g., `C Ti`), and the original thought chain is discarded, with reasoning continuing based solely on the compressed content. This approach significantly reduces the number of tokens stored in the context window, thereby lowering memory overhead and computational costs.

In practice, we train the LLM to learn when and how to compress. Specifically, we construct data to teach the model when to compress; the hidden states of the thoughts to be compressed are reduced to a set of hidden states corresponding to a small number of special tokens (i.e., gist tokens (Mu et al., 2023)). Through carefully designed attention masks, the LLM then learns how to compress and how to continue generating based on the compressed content. To quantify the amount of information used during reasoning, we further propose the Dependency (Dep) metric, defined as the total number of historical tokens each generated token depends on (see Fig. 3). This metric effectively measures the degree of compression, with a lower Dep value indicating reduced reliance on the original long context and more significant compression.

We conduct extensive experiments across four datasets using two different models. The results indicate that, with the Qwen model, LightThinker reduces the peak token usage by 70% and decreases inference time by 26% compared to the Vanilla model, while maintaining comparable accuracy (with only a 1% drop). Our contributions are as follows: 1) We propose LightThinker, a method that dynamically compresses thought chains during reasoning, significantly reducing memory overhead and inference time. 2) We introduce the Dependency (Dep) metric to measure the compression ratio and the amount of information used during reasoning. 3) We demonstrate that LightThinker achieves a good balance between reasoning efficiency and accuracy, offering new insights for future LLM inference acceleration.

2 Background

Slow Thinking. The reasoning ability of LLMs is crucial (Qiao et al., 2023), especially in solving complex problems, necessitating a shift from the fast-thinking System 1 to the slow-thinking System 2 (Sloman, 1996; Kahneman, 2011; Booch et al., 2021). For instance, Chain-of-Thought (CoT) (Wei et al., 2022) approaches decompose complex problems into sub-problems and solve them step-by-step. *o1-like thinking mode* (Jaech et al., 2024; Qwen., 2024; DeepSeek-AI et al., 2025) goes a step further by incorporating abilities such as trial, reflection, backtracking, and correction on top of the divide-and-conquer strategy. Empirical evidence (Jaech et al., 2024; DeepSeek-AI et al., 2025) shows that the *o1-like thinking mode* significantly enhances the model's ability to solve complex problems compared to CoT. This slow-thinking mode

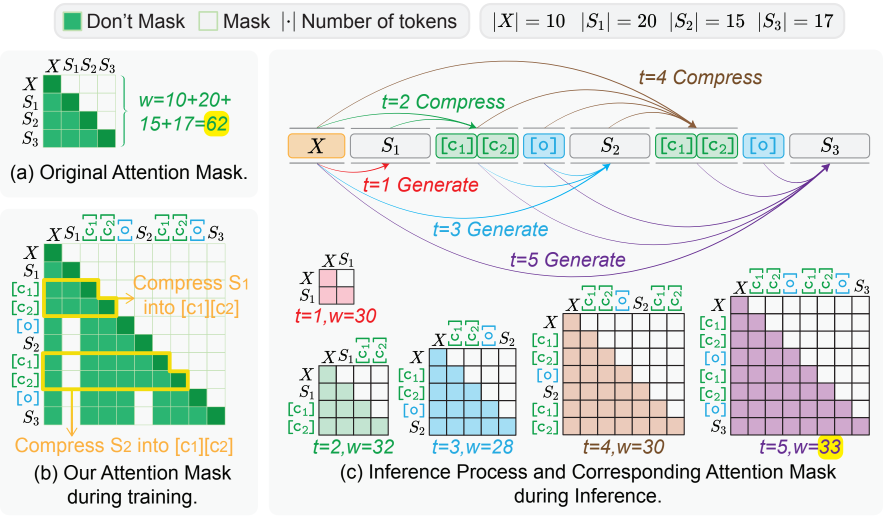

Don't Mask
Mask
|·| Number of tokens
$|X| = 10$ $|S_1| = 20$ $|S_2| = 15$ $|S_3| = 17$
w=10+20+15+17=62
(a) Original Attention Mask.
Compress S1 into [c1][c2]
Compress S2 into [c1][c2]
(b) Our Attention Mask during training.
t=2 Compress
t=4 Compress
t=1 Generate
t=3 Generate
t=5 Generate
t=1,w=30
t=2,w=32
t=3,w=28
t=4,w=30
t=5,w=33
(c) Inference Process and Corresponding Attention Mask during Inference.

Figure 2: Overview of LightThinker, illustrated with an example requiring three-step reasoning. Fig. (a) shows the attention mask of Vanilla during both training and inference. Fig. (b) depicts the attention mask of LightThinker during the training. Fig. (c) presents the complete inference process of LightThinker along with the attention mask corresponding to each step. Here, 'w' denotes the size of the matrix.

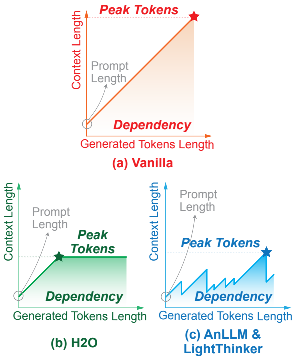


Figure 3: The relationship between context length and the number of generated tokens across different methods. The *Dependency* metric represents the area under the curve, while the *Peak Token* denotes the maximum value of the curve. See Appx. A for details.

can be instilled in models through carefully constructed data using Supervised Fine-Tuning (SFT). In terms of the number of output tokens, the token consumption of System 1, CoT, and *o1-like thinking mode* increases progressively.

Inference Challenges. Recent works on *o1-like thinking mode* (Wu et al., 2024b) highlight the necessity of generating a substantial number of tokens for complex problem-solving. As the core structure of Transformers (Vaswani et al., 2017), the attention mechanism faces two significant challenges during inference as token generation scales: 1) The *memory* overhead gradually increases. To speed up inference, each token's Key and Value are cached at every layer. For the Qwen-32B (Yang et al., 2024), when the context length reaches 10^4 tokens, the space occupied by the KV cache is comparable to that of the model itself. 2) The *computational cost* of generating a single token in an autoregressive manner also increases. Due to the attention mechanism in Transformers (Vaswani et al., 2017), the computational load grows *quadratically* with the number of tokens.

3 Methodology

We propose **LightThinker** to accelerate the reasoning process of LLMs, as illustrated in Figure 2. The core idea is to *train LLMs to dynamically compress the current thought during reasoning*, enabling subsequent generation to be based on the compressed content rather than the original long thought.

3.1 Overview

Notation. For clarity, we define the following notations. Lowercase letters, such as x_i, denote a single token. Uppercase letters, such as X, denote sequences of tokens. The notation '[·]' denotes a special token, such as '[c]', while '<·>' denotes an optional special token, such as '<w>'. The *o1-like thinking mode* dataset $\mathcal{D} = \{(X, Y)_i\}_{i=1}^{|\mathcal{D}|}$ consists of $|\mathcal{D}|$ samples, where $X = \{x_i\}_{i=1}^{|X|}$ represents a question, and $Y = \{y_i\}_{i=1}^{|Y|}$ represents the corresponding thought and final answer. Recent works (Team, 2025; DeepSeek-AI et al., 2025) show that SFT on $\mathcal{D}$ significantly enhances LLM reasoning capabilities.

Design. To achieve the core idea, we focus on addressing two key questions: *i) When to compress?* The timing of compression significantly impacts reasoning efficiency and compression quality. We explore two different strategies. The first is *token-level* (Zhang et al., 2024b), where compression is performed after a fixed number of tokens. This strategy is straightforward to implement but may ignore semantic boundaries. The second is *thought-level* (Pang et al., 2024), where compression is performed after a complete "thought", defined as a sentence or paragraph. This strategy better preserves semantic information but requires a more complex segmentation function. *ii) How to compress?*

The goal of compression is to encode the current lengthy thought into a more compact representation. We investigate two different approaches. The first is *text compression*, where the current thought is encoded into a shorter text (Jiang et al., 2023) or a chunk of continuous vectors (Chevalier et al., 2023; Ge et al., 2024). This method requires an additional encoding model and increases computational overhead. The second is *hidden state compression*, where the hidden state of the current thought is compressed into the hidden states of a few special tokens (i.e., *gist tokens* (Mu et al., 2023)). We choose this method as it does not require additional models. Specifically, in our work, we address the first question by *reconstructing data* and the second by *constructing thought-based attention mask*.

What content has been compressed? We do not aim to compress lengthy thought information into a compact representation without loss. Instead, our focus is on preserving only the information that is essential for subsequent reasoning. As highlighted by the gray dashed box in Figure 1(b), the lengthy thought is retained solely for the elements that contribute to further inference.

3.2 LightThinker

Data Reconstruction. To enable LLMs to dynamically compress during generation, we reconstruct the original dataset $\mathcal{D}$ as follows. First, we segment the output. Given the input X and output Y, we use a segmentation function `Seg()` to divide Y into k subsequences S, i.e., $Y = \{S_i\}_{i=1}^{k}$. The function can be based on *token-level* or *thought-level*. Then, we insert the special tokens. Specifically, we insert a set of special tokens $\{\texttt{<w>}, C, \texttt{[o]}\}$ between adjacent subsequences S_i, where `<w>` is an *optional* compression trigger, indicating the need to compress the preceding thought. It can be omitted if the `Seg()` is token-level or if $\texttt{<w>} \in S_i$. The token $C = \{\texttt{[c}_{\texttt{i}}\texttt{]}\}_{i=1}^{|C|}$ consists of $|C|$ special tokens, serving as gist tokens to store compressed content. Here we refer to C as *cache tokens* and denote $|C|$ as the *cache size*. The token `[o]` is a mandatory output token, enabling continual generation based on compressed content, inspired by Zhang et al.. Finally, the enhanced data is

$$\hat{Y} = \{S_1, \underline{\texttt{<w>}, C, \texttt{[o]}}, S_2, \underline{\texttt{<w>}, C, \texttt{[o]}}, \dots, S_k\},$$

and the enhanced dataset is defined as $\hat{\mathcal{D}} = \{(X, \hat{Y})_i\}_{i=1}^{|\hat{\mathcal{D}}|}$. For simplicity, we assume $\texttt{<w>} \in S_i$, so we omit it. Additionally, we use superscripts to distinguish different special tokens at different positions, such as $C^{(1)}$ and $\texttt{[o]}^{(1)}$ for tokens following S_1, though they are the same across different positions.

Thought-based Attention Mask Construction. To enable LLMs to learn how to compress and how to generate based on the compressed content (i.e., how to understand the compressed content), we manipulate *Thought-based Mask Construction* as shown in Figure 2(b). Specifically, let $S_{<i} = \{S_1, \dots, S_{i-1}\}$ denotes the sequence before the i-th thought S_i.

During compression, $C^{(i)}$ tokens can only attend to the question X, previous compressed content $\{C, \texttt{[o]}\}^{(<i)}$, and the current thought S_i, that is,

$$\begin{aligned} C^{(i)} \leftarrow \texttt{Cmp}(&X, \\ &\{C^{(1)}, \texttt{[o]}^{(1)}, \dots, C^{(i-1)}, \texttt{[o]}^{(i-1)}\}, S_i), \end{aligned}$$

where `Cmp()` is compression operation. This allows the LLM to compress the key content of S_i into $C^{(i)}$. A detailed mathematical description of `Cmp()` is in Appx. B.

During generation, token $\texttt{[o]}^{(i)}$ can only attend to the question X and the previous compressed content $\{C, \texttt{[o]}\}^{(\leq i)}$, that is,

$$S_{i+1} \leftarrow \texttt{Gen}(X, \{C^{(1)}, \texttt{[o]}^{(1)}, \dots, C^{(i)}, \texttt{[o]}^{(i)}\}),$$

where `Gen()` is generation operation. This enables the LLM to continue reasoning based on the question and previous compressed content.

Training and Inference. Training objective is to maximize the following probability distribution:

$$\begin{aligned} &P_\theta(S_1|X) \cdot P_\theta(S_2|X, C^{(1)}, \texttt{[o]}^{(1)}) \cdot \dots \\ &\cdot P_\theta(S_k|X, \{C^{(i)}, \texttt{[o]}^{(i)}\}_{i=1}^{k-1}), \end{aligned}$$

where θ represents the LLM parameters. Notably, during training, LLM is not allowed to predict the input X and the special tokens C and `[o]`. The training samples are drawn from the $\hat{\mathcal{D}}$, and we employ an attention mask to encourage the LLM to learn to compress and comprehend the compressed content. The entire training process remains based on next token prediction. The detailed inference procedure is illustrated in Fig. 1(b) and Fig. 2(c).

4 Experiments

4.1 Experimental Settings

Baselines. We conduct experiments on two LLMs: Qwen2.5-7B (Yang et al., 2024) and Llama3.1-8B (Dubey et al., 2024). To establish an upper

Method	GSM8K				MMLU				GPQA				BBH				AVG.			
	Acc↑	Time↓	Peak↓	Dep↓	Acc↑	Time↓	Peak↓	Dep↓	Acc↑	Time↓	Peak↓	Dep↓	Acc↑	Time↓	Peak↓	Dep↓	Acc↑	Time↓	Peak↓	Dep↓
Qwen2.5-7B Series																				
CoT	86.12	1.66	513	0.1M	66.50	1.77	649	0.2M	26.76	0.60	968	0.5M	65.45	0.68	570	0.1M	61.21	1.18	675	0.2M
Distill-R1	81.88	5.60	844	1.1M	51.70	14.31	2483	7.5M	24.75	8.01	6718	31M	57.78	5.53	1967	6.0M	54.03	8.36	3003	11.3M
Vanilla	90.90	11.83	2086	3.9M	59.98	20.61	3417	10M	30.81	10.76	8055	39M	69.90	11.50	3786	13M	62.90	13.68	4336	16.6M
+ H2O	89.92	22.19	**640**	1.2M	59.69	29.02	1024	3.2M	24.75	15.61	1200	9.8M	70.10	15.61	**1024**	3.5M	61.12	20.61	**972**	4.4M
+ SepLLM	30.40	53.52	1024	6.9M	10.81	53.45	1024	9.0M	0.00	11.65	**1024**	10M	8.08	26.64	1024	9.4M	12.32	36.32	1024	8.9M
AnLLM	78.39	15.26	789	1.6M	54.63	14.13	875	2.0M	19.70	9.14	3401	11M	54.95	10.04	1303	3.8M	51.92	12.14	1592	4.6M
Ours (*tho.*)	**90.14**	**11.46**	676	**1.0M**	**60.47**	**13.09**	944	**1.9M**	**30.30**	8.41	2385	**9.3M**	**70.30**	**7.71**	1151	**2.7M**	**62.80**	**10.17**	1289	**3.7M**
Ours (*token*)	87.11	11.48	1038	1.5M	57.35	13.80	**489**	3.5M	28.28	**8.26**	3940	18M	62.83	8.95	1884	5.6M	58.89	10.62	1838	7.2M
Llama3.1-8B Series																				
CoT	85.14	2.15	550	0.2M	65.82	2.39	736	0.3M	24.75	0.96	1231	0.9M	66.46	0.93	642	0.2M	60.54	1.61	790	0.4M
Distill-R1	73.62	2.58	395	0.1M	53.46	2.97	582	0.8M	20.20	5.24	3972	16M	61.21	0.83	380	0.2M	52.12	2.91	1332	4.4M
Vanilla	91.43	12.06	1986	3.0M	69.62	14.82	2883	6.9M	40.91	7.98	6622	26M	83.03	6.80	2793	5.9M	71.25	10.42	3571	10.5M
+ H2O	**90.45**	20.23	640	1.0M	**65.92**	27.11	736	1.8M	31.81	12.55	1536	7.9M	78.99	11.43	1024	2.1M	66.79	17.83	984	3.2M
+ SepLLM	26.25	50.05	1024	5.8M	25.12	50.11	1024	7.5M	2.53	12.62	1024	10M	14.55	27.14	1024	8.5M	17.11	34.98	1024	8.0M
AnLLM	77.33	17.92	**589**	1.1M	58.62	16.53	**589**	**1.2M**	31.31	7.19	**838**	**3.7M**	68.89	9.79	**621**	**1.6M**	59.04	12.86	**659**	**1.9M**
Ours (*tho.*)	88.25	**12.65**	629	**0.9M**	63.39	**14.88**	882	1.8M	**36.36**	**6.38**	1796	6.4M	**79.39**	7.46	911	1.9M	**66.85**	**10.34**	1055	2.7M
Ours (*token*)	85.52	13.87	1104	1.7M	61.05	15.85	1538	3.3M	31.82	6.94	3150	12M	74.14	**7.43**	1512	2.9M	63.13	11.02	1826	4.8M

Table 1: Main results. The CoT is based on the instruction model, while Vanilla, AnLLM, and LightThinker are based on Distill-R1. The light blue background indicates acceleration methods, with bold representing the best and underline the second best among them. The Acc of Vanilla serves as the upper bound for Acc of acceleration methods. Dep is measured in million, Time in hours, and Peak in counts. The compression ratio can be roughly estimated by the ratio of Dep between acceleration methods and Vanilla. See Appendix A for more details. Note that the results here are based on the same batch size. The results under the same memory budget are shown in Table 2.

bound performance, we perform full parameter instruction tuning using the Bespoke-Stratos-17k dataset (abbr. BS17K, with a data sample shown in Fig. 15), and the fine-tuned model is denoted as *Vanilla*. Notably, we initialize the training with the R1-Distill (DeepSeek-AI et al., 2025) (e.g., `DeepSeek-R1-Distill-Qwen-7B`) model, as we found that finetuning on instruction models (e.g., `Qwen2.5-7B-instruct`) yields limited improvements. We introduce five baselines for comparison: two training-free acceleration methods applied to Vanilla (H2O (Zhang et al., 2023) and SepLLM (Chen et al., 2024), which retain important KV Cache through specific strategies), one training-based method (AnLLM (Pang et al., 2024)), and two CoT (Wei et al., 2022) baselines (prompt the instruction model and the R1-Distill model). More details about baselines can be found in Appx. C.2.

Evaluation Metrics and Datasets. We evaluate LightThinker on four datasets: GSM8K (Cobbe et al., 2021), MMLU (Hendrycks et al., 2021), GPQA (Rein et al., 2024), and BBH (Suzgun et al., 2023). For MMLU and BBH, we randomly sample a portion of the data for evaluation. The evaluation focuses on both *effectiveness* and *efficiency*. For effectiveness, we use accuracy as the evaluation metric (*Acc*); for efficiency, we employ three metrics: inference time (*Time*), the peak number of tokens in the context during inference (*Peak*), and the sum of *dependency* of each generated token on previous tokens during the generation (*Dep*). Fig. 3 visualizes the Peak and Dep metrics, where the value of Dep equals the area enclosed by the lines. The Dep metric characterizes the amount of information used during inference, with smaller values indicating more significant compression. We aim to compare the other three metrics under similar Dep values. It is important to note that Peak characterizes a momentary state, while Dep characterizes the entire inference process, so there is no direct correlation between the two. For more details about Dep, please refer to Appx. A.

Implementation. For LightThinker, we design two different segmentation functions `Seg()`. At the token level, we compress every 6 tokens into 2 tokens, i.e., $|C| = 2$, denoted as "ours (*token*)". At the thought level, we use "\n\n" as a delimiter to simply segment the B17K data into several thoughts, denoted as "ours (*tho.*)". For the Qwen, we compress a thought into 9 tokens, i.e., $|C| = 9$; for the Llama, we compress a thought into 7 tokens, i.e., $|C| = 7$. In all experiments, we use greedy decoding with a maximum output length of 10240 tokens. Please refer to Appx. C for more details.

4.2 Main Results

In Tab. 1, we report the results of four evaluation metrics for two models on four datasets. **Key observations include: *1)*** Distill-R1 performs worse than CoT across all datasets because its weaker instruction-following ability (Li et al., 2025) prevents effective answer extraction using rules, even with attempts at evaluation using an LLM evaluator. However, this issue is not the focus of this paper. ***2)*** H2O effectively reduces memory usage while maintaining the performance of the vanilla, indicating that the greedy eviction policy is effective in long-text generation tasks. However, H2O significantly increases inference time compared to Vanilla, with an average increase of 51% $((20.61 - 13.68)/13.68 \approx 0.51)$ on Qwen and 72% on Llama. This is attributed to the token-wise eviction policy of H2O, which introduces additional overhead for each generated token. ***3)*** SepLLM performs the worst in terms of performance, gradually losing language ability during generation, which results in the inability to output termination tokens and thus leads to excessive inference time. ***4)*** Compared to H2O, LightThinker (*tho.*) achieves similar performance with lower Dep values (i.e., similar compression rate), while reducing inference time by an average of 52% on Qwen and 41% on Llama. Additionally, LightThinker (*tho.*) retains higher accuracy and faster inference speed compared to AnLLM.

Based on these observations, **we draw the following conclusions:** ***1)*** B17K is an effective dataset. We find Vanilla outperforms CoT and Distill-R1 on most datasets, indicating that B17K is an effective dataset that mitigates the repetition issue in Distill-R1 through SFT. ***2)*** LightThinker is effective and achieves a good balance between effectiveness and efficiency in inference. Specifically, on the Qwen, LightThinker sacrifices 1% accuracy but saves 26% time, reduces the peak tokens by 70%, and decreases Dep. by 78% (i.e., achieves a 16.6/3.7=4.5x compression ratio). On the Llama, it sacrifices 6% accuracy but saves 1% inference time, reduces the peak tokens by 70%, and decreases Dep. by 74% (i.e., achieves a 10.5/2.7=3.9x compression ratio). ***3)*** The segmentation function is vital for LightThinker. The thought-level segmentation function outperforms the token-level, with accuracy improvements of 6.2% on Qwen and 5.6% on Llama. This suggests that token-level segmentation leads to the loss of semantic boundaries.

4.3 Efficiency

For clarity, "LightThinker" hereafter denotes LightThinker (*tho.*). In this section, we conduct an in-depth analysis of LightThinker's efficiency, focusing on the following four questions:

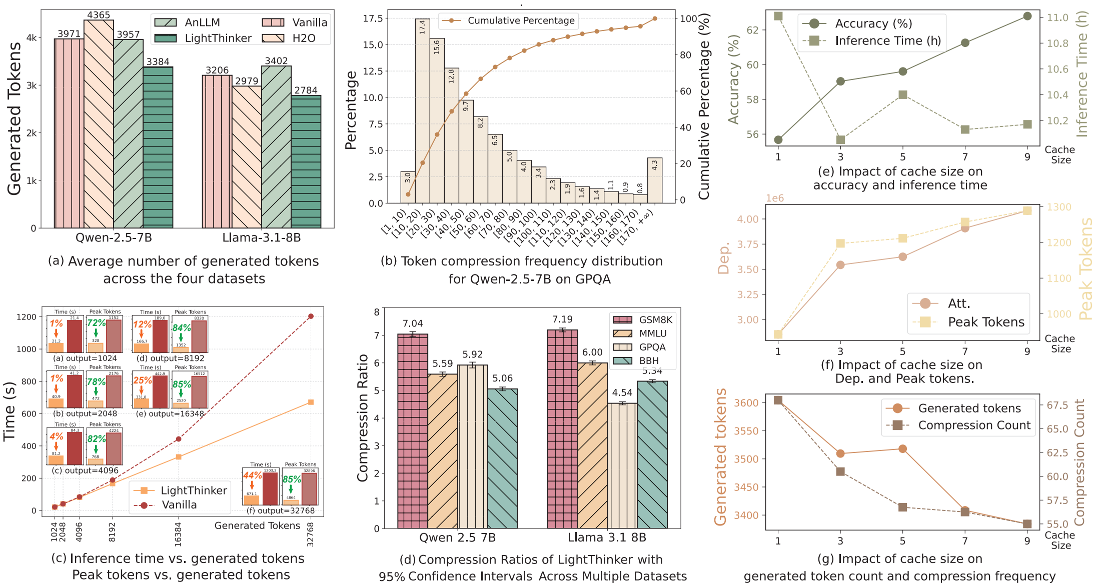
	GSM8K	MMLU	GPQA	BBH	AVG
Vanilla	11.83	20.61	10.76	11.50	13.68
LightThinker	**6.73**	**7.44**	**3.86**	**3.97**	**5.50**

Table 2: Inference time comparison (in hours) for Vanilla and LightThinker on the Qwen model across four datasets under the same memory budget.

How does LightThinker accelerate under same memory budget? Inference efficiency is measured through memory consumption and inference speed. Tab. 1 highlights LightThinker's ability to significantly reduce memory consumption at the same batch size. In practice, this reduction allows for larger batch sizes under the same memory budget, improving throughput. Experiments on four datasets with the Qwen model, conducted under the same memory constraints, show that LightThinker reduces inference time by an average of 2.5× compared to Vanilla, as demonstrated in Tab. 2. This indicates that LightThinker not only reduces both memory and time overhead at the same batch size (as shown in Tab. 1) but also significantly lowers time overhead under the same memory budget, thereby improving throughput.

Does LightThinker generate more tokens compared to Vanilla? Fig. 4(a) shows the average number of generated tokens for H2O, AnLLM, LightThinker, and Vanilla across four datasets (others in the Appx. C.5). We observe that: 1) LightThinker is the only method that reduces the number of generated tokens compared to Vanilla, with an average reduction of 15% on Qwen and 13% on Llama. This is one of the reasons for its faster inference speed. 2) H2O increases token generation by 10% on Qwen but reduces it by 7% on Llama. Despite the reduction in tokens for Llama, the inference time still increases as shown in Tab. 1, indicating that its eviction policy accumulates additional overhead as token generation grows.

What is the compression ratio of LightThinker? Fig. 4(d) illustrates the compression ratio across four datasets, Tab. 3 reports the average compression counts, and Fig. 4(b) shows the distribution of compressed token counts for GPQA using Qwen (other datasets are in the Appx. C.5). We find that: 1) Compression counts and ratios are more closely related to downstream tasks than to the model itself. Simple tasks like GSM8K exhibit lower compression counts and higher compression ratios, while complex tasks like GPQA require more frequent compressions and smaller compression ratios. 2)



(a) Average number of generated tokens across the four datasets

(b) Token compression frequency distribution for Qwen-2.5-7B on GPQA

(c) Inference time vs. generated tokens Peak tokens vs. generated tokens

(d) Compression Ratios of LightThinker with 95% Confidence Intervals Across Multiple Datasets

(e) Impact of cache size on accuracy and inference time

(f) Impact of cache size on Dep. and Peak tokens.

(g) Impact of cache size on generated token count and compression frequency

Figure 4: Efficiency Analysis and Ablation Results. Fig.(a) represents the average tokens generated by the respective model on the specified dataset. Fig.(b) shows the percentage of tokens falling within specified ranges, while the cumulative percentage curve illustrates the total proportion of tokens up to each range. Fig.(c) illustrates the relationship between the number of generated tokens and inference time. Each subplot displays the inference time and peak token for various numbers of output tokens. Fig.(d) represents the average compression ratios with 95% confidence intervals indicated by error bars. Fig.(e-f) examines the impact of cache size (i.e., $|C|$) on accuracy, Dep, inference time, peak tokens, generated tokens, and compression frequency.

	GSM8K	MMLU	GPQA	BBH
Qwen	20	37	115	48
Llama	26	47	139	55

Table 3: Statistics of the average number of compressions per dataset for LightThinker.

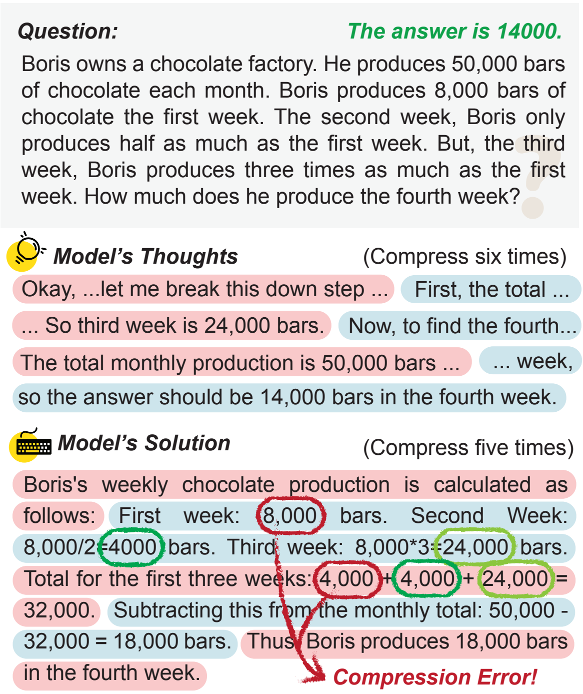
The distribution of compressed token counts follows a long-tail pattern.

How efficient is LightThinker in memory usage and inference for long-text generation? Fig. 4(c) shows the inference time and peak tokens of LightThinker and Vanilla as a function of output token length. We set the prompt length to 125 and compressed 56 tokens to 8 tokens (i.e., $|C| = 7$). We observe that: 1) Our method significantly reduces inference time. For example, when generating 32K tokens, the inference time is reduced by 44%. For shorter texts (from 1K to 4K tokens), the reduction is more modest, ranging from 1% to 4%. 2) Even for shorter texts, LightThinker substantially reduces peak tokens. For instance, when generating 1K tokens, peak tokens are reduced by 72%, and for 32K tokens, it is reduced by 85%.

4.4 Ablation

Decoupled Token and Attention Mask Mode. LightThinker differs from AnLLM in two key aspects: the decoupled token design and the attention mask as shown in Figure 9. To validate the effectiveness of these mechanisms, we conduct ablation experiments. As shown in Table 4, under the same cache size setting and using AnLLM's attention mask mechanism ("AnLLM" vs. "Ours ($|C| = 1$, T)"), the decoupled design improves accuracy by 2%. Further adopting LightThinker's attention mask mode yields an additional 7% improvement. These results demonstrate the effectiveness of both the decoupled token and the attention mask mode in LightThinker.

Cache Size. We varied $|C|$ in $\{1, 3, 5, 7, 9\}$ to observe its impact on accuracy, inference time, dependency (i.e., Dep), peak tokens, generated token count, and compression frequency. Fig. 4(e-g) illustrate these trends on the Qwen model. We observe that: 1) As shown in Figure 4(e), increasing the cache size significantly improves accuracy while reducing inference time. This indicates that a larger cache size mitigates information loss caused by compression. 2) As shown in Figure 4(g), increasing the cache size reduces both the compression frequency and the number of generated tokens. 3) Combining Fig. 4(e) and Fig. 4(g), we find that a smaller cache size leads to more frequent generation and compression to retain more information,



Question: *The answer is 14000.*

Boris owns a chocolate factory. He produces 50,000 bars of chocolate each month. Boris produces 8,000 bars of chocolate the first week. The second week, Boris only produces half as much as the first week. But, the third week, Boris produces three times as much as the first week. How much does he produce the fourth week?

Model's Thoughts (Compress six times)

Okay, ...let me break this down step ... First, the total ... ... So third week is 24,000 bars. Now, to find the fourth... The total monthly production is 50,000 bars ... ... week, so the answer should be 14,000 bars in the fourth week.

Model's Solution (Compress five times)

Boris's weekly chocolate production is calculated as follows: First week: 8,000 bars. Second Week: 8,000/2=4000 bars. Third week: 8,000*3=24,000 bars. Total for the first three weeks: 4,000 + 4,000 + 24,000 = 32,000. Subtracting this from the monthly total: 50,000 - 32,000 = 18,000 bars. Thus, Boris produces 18,000 bars in the fourth week.

Figure 5: Case Study. The figure illustrates partial inference results of a case from GSM8K. See App. C.5 for the complete content. Pink and light blue backgrounds are used to distinguish adjacent compression processes, where each color represents one compression.

while a larger cache size reduces this frequency.

4.5 Case Study

Fig. 5 illustrates a failure case from the GSM8K dataset. We observe that although the LLM arrives at the correct answer during the thinking process (see `Model's Thoughts` field in the Fig. 5), it makes an error in the final output (see `Model's Solution` field in the Figure). Specifically, in the third sentence of the `Model's Solution` field, the first occurrence of "4000" is incorrect. This indicates that information loss occurred during the second compression step (theoretically, "8000", "4000", and "24000" should have been compressed, but the LLM only compressed "4000" and "24000"), leading to subsequent reasoning errors. Such errors occur frequently in the GSM8K dataset, suggesting that the current compression method is not sufficiently sensitive to numerical values.

5 Related Work

Current research on accelerating the inference process of LLMs primarily focuses on three categories of methods: *Quantizing Model*, *Generating Fewer Tokens*, and *Reducing KV Cache*. Quantizing Model includes both parameter quantization (Lin et al., 2024) and KV Cache quantization (Liu et al., 2024b). Notably, generating long texts and understanding long-text represent distinct scenarios; therefore, acceleration methods specifically targeting the long-text generation phase (e.g., pre-filling stage acceleration techniques (Chevalier et al., 2023; Ge et al., 2024; Jiang et al., 2023; Zhang et al., 2024b; Li et al., 2024; Cai et al., 2024) are not discussed here. Due to page limits, we focus on the last one. See Appx. D for other details.

	GSM8K	MMLU	GPQA	BBH	AVG
AnLLM	78.39	54.63	19.70	54.95	51.92
Ours (|C|=1, T)	78.32	58.23	20.71	55.35	53.15
Ours (|C|=1, F)	**80.21**	**58.23**	**22.22**	**62.02**	**55.67**

Table 4: Ablation results on the Qwen, reporting accuracy on four datasets. "T" denotes the use of AnLLM's attention mask mechanism, while "F" indicates the use of LightThinker's attention mask mechanism.

Reducing KV Cache. This category can be divided into two types of strategies: pruning-based KV Cache selection in discrete space and merging-based KV Cache compression in continuous space. 1) *Pruning-Based Strategies*. Specific eviction policies (Zhang et al., 2023; Xiao et al., 2024; Chen et al., 2024) are designed to retain important tokens during inference. 2) *Merging-Based Strategies*. Anchor tokens are introduced, and LLMs are trained to compress historically important information into these tokens, thereby achieving KV Cache merging (Pang et al., 2024). Both strategies require intervention during inference. The key difference is that the first strategy is training-free but applies the eviction policy for every generated token, while the second is a training-based method and allows the LLM to decide when to apply the eviction policy.

6 Conclusion

In this paper, we present LightThinker, a new approach to enhance the efficiency of LLMs in complex reasoning tasks by dynamically compressing intermediate thoughts during generation. By training the LLM to learn when and how to compress verbose thought steps into compact representations, LightThinker significantly reduces memory overhead and computational costs while maintaining competitive accuracy. We introduce the *Dependency* (abbr., Dep) metric to quantify the degree of compression across different accelerating methods. Extensive experiments demonstrate that LightThinker is an effective approach to balancing efficiency and performance.

Limitations

Although LightThinker has shown remarkable advancements in memory optimization and inference speed enhancement, certain limitations warrant careful consideration:

1. The number of cache tokens is fixed during training and must remain consistent during inference. The generalization capability of these token representations is uncertain. For instance, whether representations trained with 3 tokens can extrapolate to scenarios requiring more tokens during inference.

2. The design of the segmentation function is relatively simplistic, relying on rule-based methods. Future work could investigate more advanced segmentation strategies.

3. The performance of LightThinker on tasks such as novel generation, code generation, and multi-turn dialogue remains unassessed.

Acknowledgement

We would like to express our sincere gratitude to the anonymous reviewers for their thoughtful and constructive feedback. This work was supported by the National Natural Science Foundation of China (No. 62576307, No. NSFCU23B2055, No. NSFCU19B2027), the Fundamental Research Funds for the Central Universities (226-2023- 00138), Ningbo Natural Science Foundation (2024J020), Yongjiang Talent Introduction Programme (2021A-156-G), and Information Technology Center and State Key Lab of CAD&CG, Zhejiang University. This work was supported by Ant Group and Zhejiang University - Ant Group Joint Laboratory of Knowledge Graph.

References

Daman Arora and Andrea Zanette. 2025. Training language models to reason efficiently. *arXiv preprint arXiv:2502.04463*.

Amos Azaria, Rina Azoulay, and Shulamit Reches. 2024. Chatgpt is a remarkable tool—for experts. *Data Intelligence*, 6(1):240–296.

Grady Booch, Francesco Fabiano, Lior Horesh, Kiran Kate, Jonathan Lenchner, Nick Linck, Andreas Loreggia, Keerthiram Murgesan, Nicholas Mattei, Francesca Rossi, et al. 2021. Thinking fast and slow in ai. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 15042–15046.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. 2024. Pyramidkv: Dynamic KV cache compression based on pyramidal information funneling. *CoRR*, abs/2406.02069.

Guoxuan Chen, Han Shi, Jiawei Li, Yihang Gao, Xiaozhe Ren, Yimeng Chen, Xin Jiang, Zhenguo Li, Weiyang Liu, and Chao Huang. 2024. Sepllm: Accelerate large language models by compressing one segment into one separator. *CoRR*, abs/2412.12094.

Jeffrey Cheng and Benjamin Van Durme. 2024. Compressed chain of thought: Efficient reasoning through dense representations. *CoRR*, abs/2412.13171.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 3829–3846. Association for Computational Linguistics.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Yuntian Deng, Yejin Choi, and Stuart M. Shieber. 2024. From explicit cot to implicit cot: Learning to internalize cot step by step. *CoRR*, abs/2405.14838.

Yuntian Deng, Kiran Prasad, Roland Fernandez, Paul Smolensky, Vishrav Chaudhary, and Stuart M. Shieber. 2023. Implicit chain of thought reasoning via knowledge distillation. *CoRR*, abs/2311.01460.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Mengru Ding, Hanmeng Liu, Zhizhang Fu, Jian Song, Wenbo Xie, and Yue Zhang. 2024. Break the chain: Large language models can be shortcut reasoners. *CoRR*, abs/2406.06580.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. 2024. In-context autoencoder for context compression in a large language model. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. 2024. Token-budget-aware LLM reasoning. *CoRR*, abs/2412.18547.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2024. Training large language models to reason in a continuous latent space. *CoRR*, abs/2412.06769.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, and Ilge Akkaya. 2024. Openai o1 system card. *CoRR*, abs/2412.16720.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 13358–13376. Association for Computational Linguistics.

Daniel Kahneman. 2011. Thinking, fast and slow. *Farrar, Straus and Giroux*.

Yu Kang, Xianghui Sun, Liangyu Chen, and Wei Zou. 2024. C3ot: Generating shorter chain-of-thought without compromising effectiveness. *CoRR*, abs/2412.11664.

Bespoke Labs. 2025. Bespoke-stratos: The unreasonable effectiveness of reasoning distillation. https://www.bespokelabs.ai/blog/bespoke-stratos-the-unreasonable-effectiveness-of-reasoning-distillation. Accessed: 2025-01-22.

Xiaomin Li, Zhou Yu, Zhiwei Zhang, Xupeng Chen, Ziji Zhang, Yingying Zhuang, Narayanan Sadagopan, and Anurag Beniwal. 2025. When thinking fails: The pitfalls of reasoning for instruction-following in llms. *Preprint*, arXiv:2505.11423.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org.

Tengxiao Liu, Qipeng Guo, Xiangkun Hu, Cheng Jiayang, Yue Zhang, Xipeng Qiu, and Zheng Zhang. 2024a. Can language models learn to skip steps? In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Haotian Luo, Li Shen, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Naiqiang Tan, Xiaochun Cao, and Dacheng Tao. 2025. O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning. *Preprint*, arXiv:2501.12570.

Jesse Mu, Xiang Li, and Noah D. Goodman. 2023. Learning to compress prompts with gist tokens. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Sania Nayab, Giulio Rossolini, Giorgio C. Buttazzo, Nicolamaria Manes, and Fabrizio Giacomelli. 2024. Concise thoughts: Impact of output length on LLM reasoning and cost. *CoRR*, abs/2407.19825.

Jianhui Pang, Fanghua Ye, Derek F. Wong, Xin He, Wanshun Chen, and Longyue Wang. 2024. Anchor-based large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 4958–4976. Association for Computational Linguistics.

Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2023. Reasoning with language model prompting: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5368–5393, Toronto, Canada. Association for Computational Linguistics.

Team Qwen. 2024. Qwq: Reflect deeply on the boundaries of the unknown.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2024. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Steven A Sloman. 1996. The empirical case for two systems of reasoning. *Psychological bulletin*, 119(1):3.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2023. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada,* *July 9-14, 2023*, pages 13003–13051. Association for Computational Linguistics.

OpenThoughts Team. 2025. Open Thoughts. https://open-thoughts.ai.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Xiaoqiang Wang, Suyuchen Wang, Yun Zhu, and Bang Liu. 2025. System-1.5 reasoning: Traversal in language and latent spaces with dynamic shortcuts. *CoRR*, abs/2505.18962.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Jialong Wu, Zhenglin Wang, Linhai Zhang, Yilong Lai, Yulan He, and Deyu Zhou. 2024a. SCOPE: optimizing key-value cache compression in long-context generation. *CoRR*, abs/2412.13649.

Siwei Wu, Zhongyuan Peng, Xinrun Du, Tuney Zheng, Minghao Liu, Jialong Wu, Jiachen Ma, Yizhi Li, Jian Yang, Wangchunshu Zhou, Qunshu Lin, Junbo Zhao, Zhaoxiang Zhang, Wenhao Huang, Ge Zhang, Chenghua Lin, and Jiaheng Liu. 2024b. A comparative study on reasoning patterns of openai's o1 model. *CoRR*, abs/2410.13639.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024. Qwen2.5 technical report. *CoRR*, abs/2412.15115.

Jintian Zhang, Cheng Peng, Mengshu Sun, Xiang Chen, Lei Liang, Zhiqiang Zhang, Jun Zhou, Huajun Chen, and Ningyu Zhang. 2024a. OneGen: Efficient one-pass unified generation and retrieval for LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4088–4119, Miami, Florida, USA. Association for Computational Linguistics.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. 2024b. Long context compression with activation beacon. *Preprint*, arXiv:2401.03462.

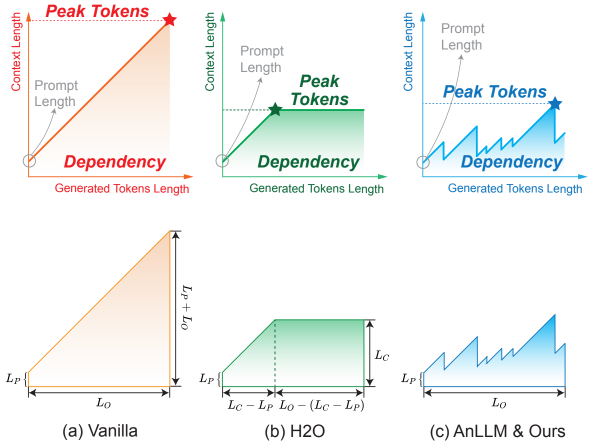
Zhen Zhang, Xuehai He, Weixiang Yan, Ao Shen, Chenyang Zhao, Shuohang Wang, Yelong Shen, and Xin Eric Wang. 2025. Soft thinking: Unlocking the reasoning potential of llms in continuous concept space. *CoRR*, abs/2505.15778.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

Appendix

A Metric: Dependency



(a) Vanilla (b) H2O (c) AnLLM & Ours

Figure 6: Illustration of the metric Dependency.

A.1 Motivation

LightThinker and AnLLM (Pang et al., 2024) are dynamic compression methods, meaning the number of compressions and the compression ratio are determined by the LLM itself rather than being predefined hyperparameters. In contrast, H2O (Zhang et al., 2023) and SepLLM (Chen et al., 2024) allow users to set hyperparameters to control the maximum number of tokens retained during inference. This fundamental difference makes it challenging to directly and fairly compare dynamic compression methods like LightThinker and AnLLM with KV cache compression approaches like H2O and SepLLM.

Traditionally, KV cache compression methods are compared by setting the same maximum peak token count, but this metric becomes inadequate in our context. As illustrated in Figure 6, which shows the relationship between generated tokens and context length for Vanilla, H2O, and LightThinker, LightThinker occasionally exceeds H2O in peak token count. However, this metric is misleading because LightThinker's peak memory usage occurs only momentarily, while H2O maintains a consistently high token count over time.

Moreover, previous KV cache compression methods often compress prompt parts only and assume a fixed prompt length, allowing compression ratios to be predefined. In our setting, however, the output is also needed to be compressed. The output token count is unknown, making it impossible to preset a global compression ratio. Consequently, relying solely on maximum peak token count as a comparison metric is insufficient.

To address these challenges, we propose a new metric called *Dependency*, which quantifies the total amount of information dependencies during the generation process. This metric enables fair comparisons between dynamic compression methods and traditional KV cache compression approaches by ensuring evaluations are conducted under similar effective compression ratios.

A.2 Definition

We introduce the **Dependency** (abbr., Dep) metric, defined as the sum of dependencies of each generated token on previous tokens during the generation of an output. Geometrically, it represents the area under the curve in Figure 6. Dependency can be calculated either from its definition or through its geometric interpretation. Here, we focus on the geometric approach. Let the initial prompt length be L_P, the model's output length be L_O, and the maximum context length set by KV cache compression methods be L_C.

Dependency for Vanilla. The area under Vanilla's curve forms a right trapezoid, calculated as:

$$\begin{aligned}\text{Dependency} &= \frac{(L_P + L_P + L_O) \times L_O}{2} \\ &= \frac{{L_O}^2}{2} + L_P \times L_O\end{aligned}$$

Dependency for H2O. The area under H2O's curve consists of a trapezoid (left part in Figure 6(b)) and a rectangle (right part in Figure 6(b)):

$$\begin{aligned}S_{\texttt{Trapezoid}} &= \frac{(L_P + L_C) \times (L_C - L_P)}{2} \\ S_{\texttt{rectangle}} &= L_C \times (L_O - L_C + L_P) \\ \text{Dependency} &= S_{\texttt{Trapezoid}} + S_{\texttt{rectangle}} \\ &= \frac{2L_PL_C + 2L_OL_C - {L_P}^2 - {L_C}^2}{2}\end{aligned}$$

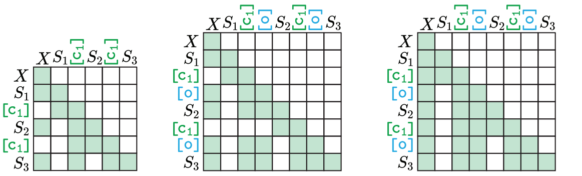
Dependency for LightThinker and AnLLM. For LightThinker and AnLLM, Dependency does not have a closed-form solution and must be computed iteratively based on its definition.

A.3 Application

Value of Dependency. A higher Dependency value indicates that more tokens need to be considered during generation, reflecting greater information usage. Conversely, a lower Dependency value suggests a higher effective compression ratio.

Dependency Ratio. By dividing the Dependency of an accelerated method by that of Vanilla, we obtain the compression ratio relative to Vanilla. For example, in Table 1's "Avg." column, Vanilla's Dependency is 16.6M, H2O's is 4.4M, and LightThinker's is 3.7M. Thus, H2O achieves a compression ratio of $\frac{16.6}{4.4} \approx 3.8$, while LightThinker achieves $\frac{16.6}{3.7} \approx 4.5$.

This metric provides a unified framework for evaluating both dynamic and static compression methods, ensuring fair and meaningful comparisons.



(a) AnLLM's Attention Mask.

(b) LightThinker's Attention Mask. (|C|=1, T)

(c) LightThinker's Attention Mask. (|C|=1, F)

Figure 7: Illustration of Attention Mask in Table 4.

B Mathematical Description of Compression

In this section, we provide a detailed formulation of the compression operation introduced in Section 3.2.

Notation. During compression, the context can be divided into three segments: 1. The sequence that remains in the context without being compressed, denoted as $Pre := \{X, \{C^{(1)}, \texttt{[o]}^{(1)} \ldots, C^{(i-1)}, \texttt{[o]}^{i-1}\}\}$, with the number of tokens represented by N; 2. The thought sequence to be compressed, defined as $Tho := S_i$, with the number of tokens denoted by T; 3. The sequence storing the compressed content, $C := C^{(i)}$, with its length represented by $|C|$.

Compression Operation. Here, we describe the compression operation at a specific layer, focusing on the information passed to the sequence C. According to the definition of self-attention (Vaswani et al., 2017), the attention matrix for the sequence C with respect to other content is calculated as:

$$A = \text{Softmax}(\text{mask}(\frac{Q^C[K^{Pre} : K^{Tho} : K^C]^\top}{\sqrt{d}}))$$

where $[:]$ denotes the concatenation operation, $\text{mask}(\cdot)$ represents the attention mask corresponding to the "Thought-based Attention Mask Construction" in Section 3.2, $K^{Pre}, V^{Pre} \in \mathbb{R}^{N \times d}$, $K^{Tho}, V^{Tho} \in \mathbb{R}^{T \times d}$, $K^C, V^C \in \mathbb{R}^{|C| \times d}$, $Q^C \in \mathbb{R}^{|C| \times d}$, and d is the hidden dimension. The matrix $A \in \mathbb{R}^{|C| \times (N+T+|C|)}$ describes the attention of sequence C to other content. The *values* of the other sequences are then weighted and summed according to the attention matrix:

$$H = A \times [V^{Pre} : V^{Tho} : V^C]$$

where $[V^{Pre} : V^{Tho} : V^C] \in \mathbb{R}^{(N+T+|C|) \times d}$, and thus $H \in \mathbb{R}^{|C| \times d}$. At this point, the information from the current Tho is preserved in H. Through training, the model learns to selectively retain useful information from Tho in H. H is then stored in the KV Cache after passing through an MLP and the next layer's projection.

C Experiment

C.1 Training Data

Examples of training samples are shown in Figure 15.

C.2 Baseline Details

H2O (Zhang et al., 2023) is a training-free acceleration method that greedily retains tokens with the highest cumulative attention values from historical tokens. It includes two hyper-parameters: the maximum number of tokens and the current window size (i.e., `local_size`). The maximum number of tokens for each task is listed in the "Peak" column of Table 1, and the `local_size` is set to half of the maximum number of tokens. The experimental code is implemented based on `https://github.com/meta-llama/llama-cookbook`.

SepLLM (Chen et al., 2024) is another training-free acceleration method that considers tokens at punctuation positions as more important. It includes four parameters: the maximum number of tokens is set to 1024, `local_size` is set to 256, `sep_cache_size` is set to 64, and `init_cache_size` is set to 384. We also tried another set of parameters (`init_cache_size=4`, `sep_cache_size=64`, `local_size=720`, maximum number of tokens=1024), but found that the first set of parameters performed slightly better.

AnLLM (Pang et al., 2024) is a training-based method that shares a similar overall approach with LightThinker but accelerates by saving historical content in anchor tokens. The specific differences between the two are detailed in Section E.

C.3 Training Details

Both **Vanilla** and **AnLLM** are trained on the B17K (Labs, 2025) dataset using the R1-Distill (DeepSeek-AI et al., 2025) model for 5 epochs, while LightThinker is trained for 6 epochs. The maximum length is set to 4096, and a cosine warmup strategy is adopted with a `warmup_ratio` of 0.05. Experiments are conducted on 4 A800 GPUs with DeepSpeed ZeRo3 offload enabled. The batch size per GPU is set to 5, and the gradient accumulation step is set to 4, resulting in a global batch size of 80. The learning rate for Vanilla is set to 1e-5, while for AnLLM and LightThinker, it is set to 2e-5.

C.4 Evaluation Details

For the CoT in Table 1, the prompts used are shown in Figure 11 and Figure 14. For the R1-Distill model, no system prompt is used, and the task-specific prompts are shown in Figure 13. Vanilla, H2O, SepLLM, AnLLM, and LightThinker share the same set of prompts, with the system prompt shown in Figure 12 and downstream task prompts shown in Figure 13. The options for MMLU (Hendrycks et al., 2021) and GPQA (Rein et al., 2024) multiple-choice questions are randomized.

C.5 Additional Results

Figure 8 compares the number of tokens generated by two models across different datasets. Figure 10 shows the distribution of compressed lengths for LightThinker on two models and four datasets. Figure 7 illustrates the attention masks for the baselines in Table 4. Figure 16 shows a complete case in Figure 5.

D Related Work

Current research on accelerating the inference process of large language models (LLMs) primarily focuses on three categories of methods: *Quantizing Model*, *Generating Fewer Tokens*, and *Reducing KV Cache*. Quantizing Model includes both parameter quantization (Lin et al., 2024; Dettmers et al., 2022) and KV Cache quantization (Liu et al., 2024b; Hooper et al., 2024), while this section will concentrate on the latter two categories. It is important to note that generating long texts and understanding long texts represent distinct application scenarios; therefore, acceleration methods specifically targeting the long-text generation phase (e.g., pre-filling stage acceleration techniques such as AutoCompressor (Chevalier et al., 2023), ICAE (Ge et al., 2024), LLMLingua (Jiang et al., 2023), Activation Beacon (Zhang et al., 2024b), SnapKV (Li et al., 2024), and PyramidKV (Cai et al., 2024)) are not discussed here. Below is a detailed overview of the last two categories.

Generating Fewer Tokens. This category can be further divided into three strategies based on the number and type of tokens used during inference. 1) *Discrete Token Reduction*. Techniques such as prompt engineering (Han et al., 2024; Ding et al., 2024; Nayab et al., 2024), instruction fine-tuning (Liu et al., 2024a; Kang et al., 2024), or reinforcement learning (Arora and Zanette, 2025; Luo et al., 2025) are used to guide LLMs to use fewer discrete tokens during inference. For example, TALE (Han et al., 2024) prompts LLMs to complete tasks under a predefined token budget. Arora and Zanette construct specific datasets and employ reinforcement learning reward mechanisms to encourage models to generate concise and accurate outputs, thereby reducing token usage. 2) *Continuous Token Replacement*. These methods (Hao et al., 2024; Cheng and Durme, 2024) explore using continuous-space tokens instead of traditional discrete vocabulary tokens. A representative example is CoConut (Hao et al., 2024), which leverages Curriculum Learning to train LLMs to perform inference with continuous tokens. 3)*No Token Usage*. By internalizing the inference process between model layers, the final answer is generated directly during inference without intermediate tokens (Deng et al., 2024, 2023). These three strategies are implemented after model training and do not require additional intervention during inference. Technically, the acceleration effect of these methods increases sequentially, but at the cost of a gradual decline in the generalization performance of LLMs. Additionally, the first strategy does not significantly reduce GPU memory usage.

Reducing KV Cache. This category can be divided into two types of strategies: pruning-based KV Cache selection in discrete space and merging-based KV Cache compression in continuous space. 1) *Pruning-Based Strategies*. Specific eviction policies (Zhang et al., 2023; Xiao et al., 2024; Chen et al., 2024; Wu et al., 2024a) are designed to retain important tokens during inference. For example, StreamingLLM (Xiao et al., 2024) considers the initial sink tokens and the most recent tokens as important. H2O (Zhang et al., 2023)

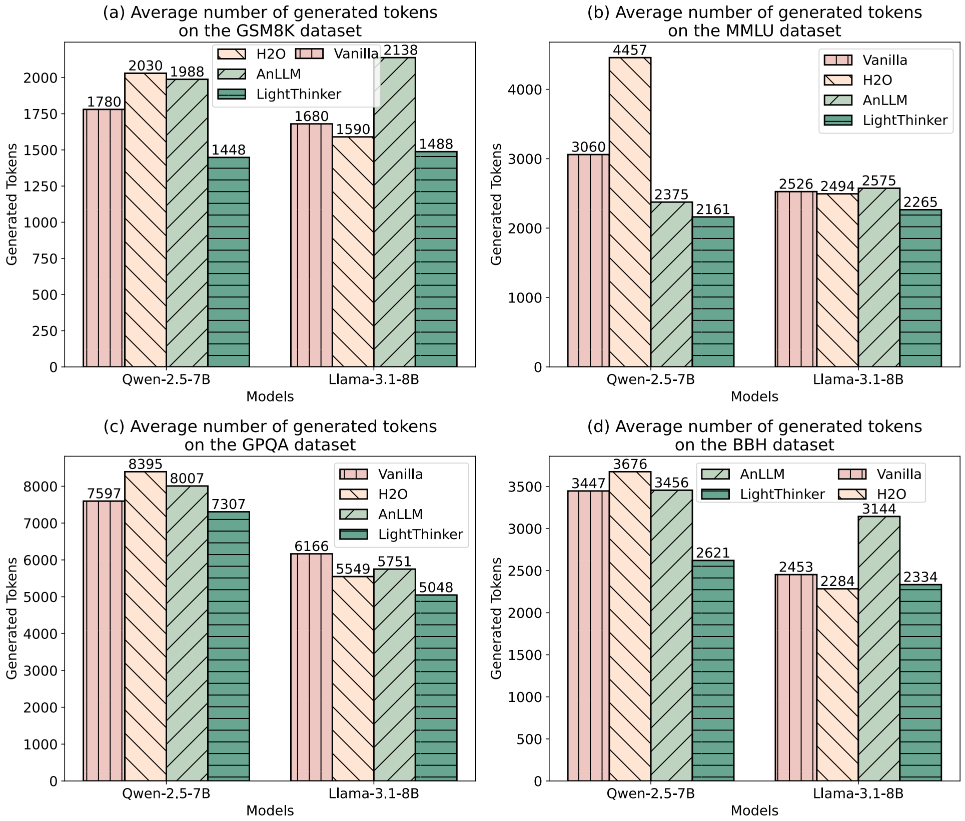


Figure 8: Average number of generated tokens.

focuses on tokens with high historical attention scores. SepLLM (Chen et al., 2024) emphasizes tokens corresponding to punctuation marks. 2) *Merging-Based Strategies*. Anchor tokens are introduced, and LLMs are trained to compress historically important information into these tokens, thereby achieving KV Cache merging (Pang et al., 2024). Both strategies require intervention during inference. The key difference is that the first strategy is training-free but applies the eviction policy for every generated token, while the second strategy is a training-based method and allows the LLM to decide when to apply the eviction policy.

E Discussions

E.1 Difference between LightThinker and AnLLM

AnLLM (Pang et al., 2024) is a work from 2023, at which time the concept of long-cot (Jaech et al., 2024; DeepSeek-AI et al., 2025) did not exist. AnLLM itself focuses more on prompt compression rather than output compression. Additionally, our method decouples compression and generation, allowing for scaling the number of cache tokens—something AnLLM cannot do. Therefore, our work is only related to AnLLM in that both use sparse attention (Zhang et al., 2023; Li et al., 2024) to speed up processes, but they are not similar works.

AnLLM is a method related to ours. In Figure 9, we compare the differences in Attention Mask between LightThinker and AnLLM: 1) *Decoupling Generation and Compression.* In AnLLM, the $[\mathtt{c_i}]$ token is tasked with both compressing historical information and generating subsequent content, as shown by the blue and pink arrows in Fig. 9. This design tightly couples generation and compression. In contrast, LightThinker decouples these tasks: the $[\mathtt{c_i}]$ token solely compresses historical information, while the $[\mathtt{o}]$ token performs reasoning based on the compressed content. 2) *Context Visibility during Compression.* AnLLM can only access the current thought during compression. LightThinker, however, allows access to X, historical compressed content, and the current thought during compression, thereby enhancing contextual understanding. Ablation experiments in Section 4.4 demonstrate that these designs significantly improve performance.

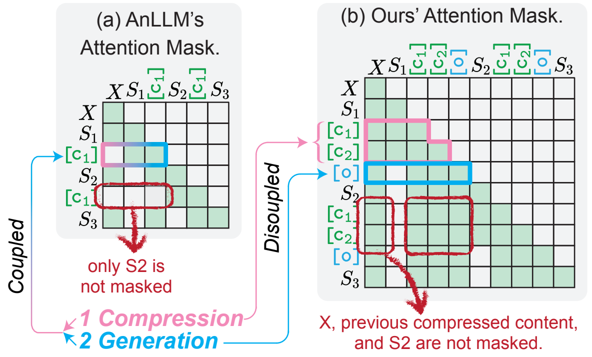

(a) AnLLM's Attention Mask.
X S1 [c1] S2 [c1] S3
X
S1
[c1]
S2
[c1]
S3
only S2 is not masked
Coupled
(b) Ours' Attention Mask.
X S1 [c1] [c2] [o] S2 [c1] [c2] [o] S3
X
S1
[c1]
[c2]
[o]
S2
[c1]
[c2]
[o]
S3
Disoupled
X, previous compressed content, and S2 are not masked.
1 Compression
2 Generation

Figure 9: Contrast of AnLLM and ours. Two differences are marked: one with a red box, and the other with blue and pink arrows.

E.2 Viewing LightThinker from Other Perspectives

In previous sections, we design LightThinker from a compression perspective. Here, we further discuss it from the perspectives of *Memory* and *KV Cache Compression*, where KV Cache can be viewed as a form of LLM work memory.

In Memory perspective, LightThinker's workflow can be summarized as follows: it first performs autoregressive reasoning, then stores key information from the reasoning process as memory (memory), and continues reasoning based on the memorized content. Thus, the information in the cache tokens acts as a compact memory, though it is only effective for the current LLM and lacks transferability.

In KV Cache Compression perspective, unlike methods such as H2O (Zhang et al., 2023), which rely on manually designed eviction policy to select important tokens, LightThinker merges previous tokens in a continuous space, *ceating* new representations. The content and manner of merging are autonomously determined by the LLM, rather than being a discrete selection process.

E.3 Why LightThinker generates more tokens with smaller cache size?

As shown in Figure 4(e-f), we find that LightThinker generates more tokens with smaller cache size. We examined outputs under different cache sizes and found that when the cache size is small, the model tends to repeat previous content more often. We believe this is because smaller cache sizes lead to greater information loss during compression, prompting the model to regenerate earlier content more frequently to retain as much information as possible.

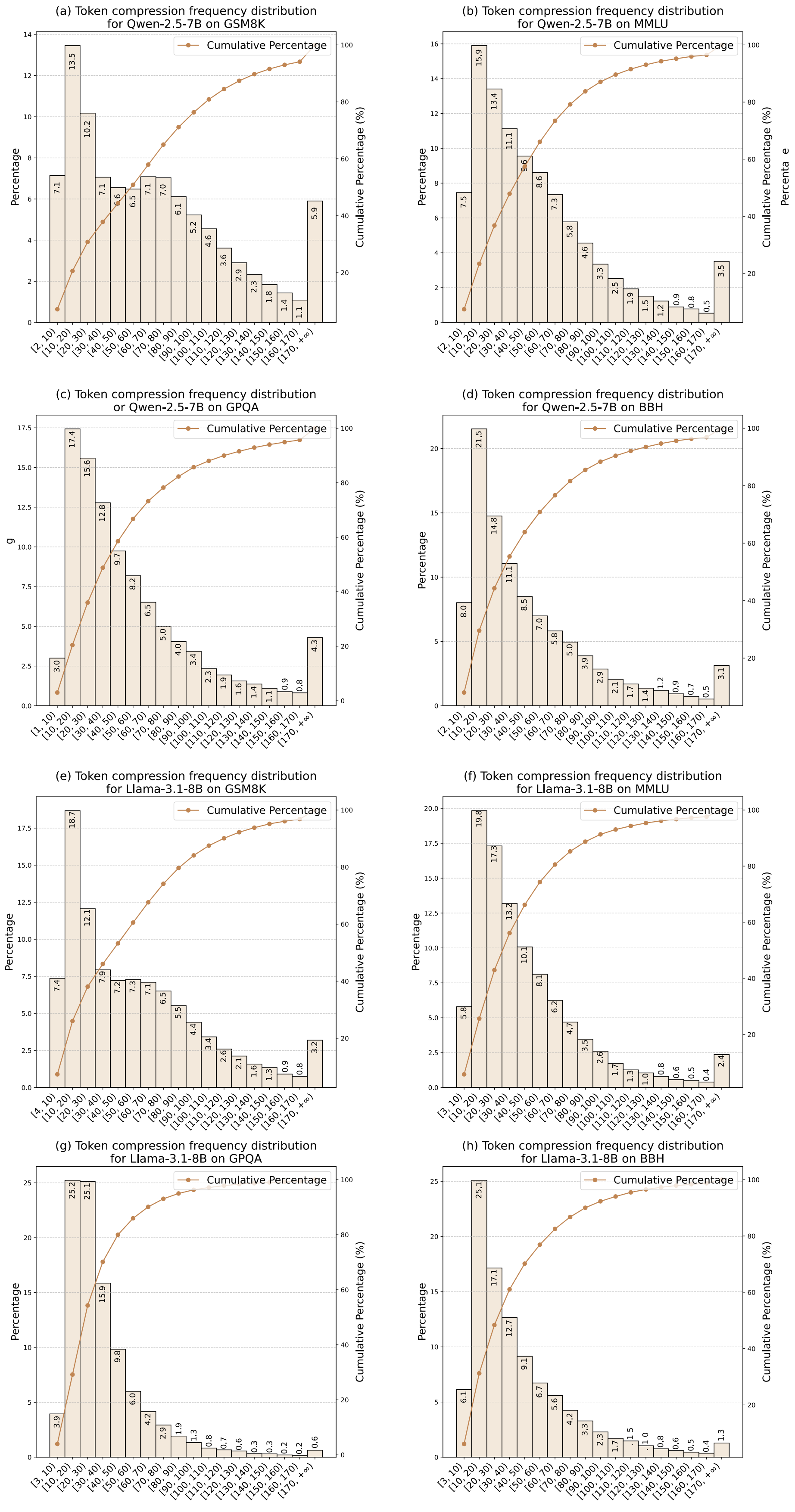
E.4 Comparison with Implicit CoT Works

Both our work and implicit CoT methods operate in continuous spaces. While implicit CoT performs reasoning entirely in continuous space, LightThinker employs a hybrid approach combining continuous and discrete space reasoning. The overview differences are shown in Table 5. Below we clarify the key differences:

- **Reasoning Acceleration Mechanism**. *Implicit CoT* methods (e.g., System-1.5 (Wang et al., 2025), SoftThinking (Zhang et al., 2025)) accelerate reasoning by reducing generation steps through continuous token representations. The entire reasoning process depends on full context. *LightThinker* accelerates inference by reducing the number of historical tokens needed for generation, without requiring full context dependence.

- **Training Approach.** Implicit CoT typically requires complex, multi-stage training with significant overhead (e.g., System-1.5's two-phase training (Wang et al., 2025) or Coconut's curriculum learning (Hao et al., 2024)). Some methods (Zhang et al., 2025; Cheng and Durme, 2024) even require architecture modifications. LightThinker uses standard SFT with modified attention masks, requiring neither specialized training data nor architectural changes.

- **Interpretability and Generalization**. Current implicit CoT methods face interpretability challenges (due to continuous reasoning) and limited out-of-domain generalization (except SoftThinking (Zhang et al., 2025)). LightThinker maintains discrete tokens for better interpretability and shows promising out-of-domain generalization in our experiments, as shown in Table 5.



(a) Token compression frequency distribution for Qwen-2.5-7B on GSM8K

(b) Token compression frequency distribution for Qwen-2.5-7B on MMLU

(c) Token compression frequency distribution or Qwen-2.5-7B on GPQA

(d) Token compression frequency distribution for Qwen-2.5-7B on BBH

(e) Token compression frequency distribution for Llama-3.1-8B on GSM8K

(f) Token compression frequency distribution for Llama-3.1-8B on MMLU

(g) Token compression frequency distribution for Llama-3.1-8B on GPQA

(h) Token compression frequency distribution for Llama-3.1-8B on BBH

Figure 10: Token compression frequency distribution for LightThinker.

System Prompt:
Below is a question. Please think through it step by step, and then provide the final answer. If options are provided, please select the correct one.
Output format:
Use “<THOUGHT>...</THOUGHT>” to outline your reasoning process, and enclose the final answer in ‘\boxed{}‘.

Example 1:
Question:
What is 2 + 3?
Output:
<THOUGHT>First, I recognize that this is a simple addition problem. Adding 2 and 3 together gives 5.</THOUGHT>
Therefore, the final answer is \boxed{5}.

Example 2:
Question:
What is 2 + 3?
A. 4
B. 5
C. 10

Output:
<THOUGHT>First, I recognize that this is a simple addition problem. Adding 2 and 3 together gives 5.</THOUGHT>
Therefore, the final answer is \boxed{B}.

Figure 11: System prompt for `Qwen2.5-7B-Instruct` and `Llama3.1-8B-Instruct`.

System Prompt:
Your role as an assistant involves thoroughly exploring questions through a systematic long thinking process before providing the final precise and accurate solutions. This requires engaging in a comprehensive cycle of analysis, summarizing, exploration, reassessment, reflection, backtracing, and iteration to develop well-considered thinking process. Please structure your response into two main sections: Thought and Solution. In the Thought section, detail your reasoning process using the specified format: `<|begin_of_thought|>` {thought with steps separated with ‘\n\n’} `<|end_of_thought|>` Each step should include detailed considerations such as analisying questions, summarizing relevant findings, brainstorming new ideas, verifying the accuracy of the current steps, refining any errors, and revisiting previous steps. In the Solution section, based on various attempts, explorations, and reflections from the Thought section, systematically present the final solution that you deem correct. The solution should remain a logical, accurate, concise expression style and detail necessary step needed to reach the conclusion, formatted as follows: `<|begin_of_solution|>` {final formatted, precise, and clear solution} `<|end_of_thought|>` Now, try to solve the following question through the above guidelines:

Figure 12: System prompt for `Vanilla`, `H2O`, `SepLLM`, `AnLLM`, and `LightThinker` for both Qwen-based model and Llama-based model.

GSM8K/MMLU/GPQA/BBH:
Return your final response within \boxed{}. {Question}

Figure 13: Task prompt for Distill-R1, Vanilla, H2O, SepLLM, AnLLM, and LightThinker for both Qwen-based model and Llama-based model.

GSM8K:
Return your final response within \boxed{}. {Question}

MMLU:
Please select the option that best answers the question. Return your final response within \boxed{}. Here are the Question:
{Question}

GPQA:
Given a question, please select the option that best answers it. Return your final response within \boxed{}. {Question}

BBH:
Return your final response within \boxed{}. If options are provided, please select the correct one.
{Question}

Figure 14: Task prompt for Qwen2.5-7B-Instruct and Llama3.1-8B-Instruct.

Method	**Memory Optimization**	**Training Cost**	**Interpretability**	**Generalization**
System-1.5 (Wang et al., 2025)	Significant	High	Weak	Weak
SoftThinking (Zhang et al., 2025)	Limited	None	Weak	Good
LightThinker	Significant	Low	Good	Good

Table 5: Comparison of different methods.

System Prompt:
See Figure 12

Input:
Return your final response within \boxed{}. The average of the numbers 23 and x is 27. What is the positive difference between 23 and x? (Recall that the average of two numbers is their sum divided by 2).

Output:
`<|begin_of_thought|>` Okay, let's see. The problem says that the average of the numbers 23 and x is 27. We need to find the positive difference between 23 and x. Hmm, right. Since the average of two numbers is their sum divided by 2, I can start by setting up the equation for the average.<w>So, the average of 23 and x is (23 + x)/2. According to the problem, this average equals 27. So I can write the equation:<w>(23 + x)/2 = 27<w>To find x, I should solve this equation. Let me multiply both sides by 2 to get rid of the denominator. That gives:<w>23 + x = 54<w>Then, subtract 23 from both sides to isolate x:<w>x = 54 - 23<w>Calculating that, 54 minus 23 is 31. So x is 31. Now, the question asks for the positive difference between 23 and x. The positive difference would be the absolute value of 23 - x. Since x is 31, that's |23 - 31|, which is | -8 |, so 8. Alternatively, it could also be |31 - 23|, which is the same result, 8. Either way, the positive difference is 8.<w>Wait, let me check my steps again to make sure I didn't make a mistake. Starting with the average: (23 + x)/2 = 27. Multiply both sides by 2: 23 + x = 54. Subtract 23: x = 31. Difference between 23 and 31 is 8. That seems right. Yeah, I think that's correct. No issues here.<w> `<|end_of_thought|>` `<|begin_of_solution|>` <w>The average of 23 and x is given by $\frac{23+x}{2} = 27$. Multiplying both sides by 2:

$$23 + x = 54$$

Subtracting 23 from both sides:

$$x = 54 - 23 = 31$$

The positive difference between 23 and x is:

$$|31 - 23| = 8$$

<w>$\boxed{8}$ `<|end_of_solution|>`

Figure 15: A training case. <w> is a compression trigger.

Question:
Boris owns a chocolate factory. He produces 50,000 bars of chocolate each month. Boris produces 8,000 bars of chocolate the first week. The second week, Boris only produces half as much as the first week. But, the third week, Boris produces three times as much as the first week. How much does he produce the fourth week?

Model Output:
<|begin_of_thought|>
Okay, let's see. Boris has a chocolate factory that produces 50,000 bars each month. The question is asking how much he produces in the fourth week. Hmm, let me break this down step by step. <|splitter|>

First, the total monthly production is 50,000 bars. The month is divided into four weeks, right? So the production for each week adds up to 50,000. The problem gives the production for the first three weeks and asks for the fourth week. Let me note down the given information. <|splitter|>

First week: 8,000 bars. That's straightforward. Then the second week, he produces half as much as the first week. Half of 8,000 is 4,000. So second week is 4,000 bars. Then the third week, he produces three times as much as the first week. Three times 8,000 is 24,000. So third week is 24,000 bars. <|splitter|>

Now, to find the fourth week's production, I need to subtract the sum of the first three weeks from the total monthly production. Let me add up the first three weeks: 8,000 + 4,000 + 24,000. Let's do the math step by step. 8,000 plus 4,000 is 12,000. Then 12,000 plus 24,000 is 36,000. So the total for the first three weeks is 36,000 bars. <|splitter|>

The total monthly production is 50,000 bars. Subtracting the first three weeks from the total gives the fourth week's production. So 50,000 minus 36,000 equals 14,000 bars. Therefore, Boris produces 14,000 bars in the fourth week. <|splitter|>

Wait a second, let me double-check my calculations to make sure I didn't make a mistake. First week: 8,000. Second week: half of 8,000 is indeed 4,000. Third week: three times 8,000 is 24,000. Adding those: 8,000 + 4,000 = 12,000; 12,000 + 24,000 = 36,000. Total monthly is 50,000. So 50,000 - 36,000 = 14,000. Yep, that seems right. I don't think I missed anything here. The problem states the production for each week as fractions or multiples of the first week, so the calculations are straightforward. No complicated formulas needed. Just basic arithmetic. So the answer should be 14,000 bars in the fourth week. <|splitter|>
<|end_of_thought|>

<|begin_of_solution|>
Boris's weekly chocolate production is calculated as follows: <|splitter|>

- **First week**: 8,000 bars
- **Second week**: $\frac{8,000}{2} = 4,000$ bars
- **Third week**: $8,000 \times 3 = 24,000$ bars <|splitter|>

Total for the first three weeks:
$4,000 + 4,000 + 24,000 = 32,000$ bars <|splitter|>

Subtracting this from the monthly total:
$50,000 - 32,000 = 18,000$ bars <|splitter|>

Thus, Boris produces $\boxed{18000}$ bars in the fourth week. <|splitter|>
<|begin_of_solution|>

Figure 16: Bad Case. <|splitter|> is equal to <w> in Figure 15.