Prefix Sliding for efficient test-time scaling

Niklas Muennighoff[s] **Zhengyang Wang**[c] **Zeyi Chen**[w] **Weijia Shi**[w] **Binyuan Hui**
John Yang[s] **Dapeng Jiang**[w] **Mika Senghaas**[p] **Fares Obeid**[p] **Johannes Hagemann**[p]
Sami Jaghouar[p] **Ludwig Schmidt**[s] **Percy Liang**[s] **Jason Wei** **Andrew Y. Ng**[s]
Luke Zettlemoyer[w] **Yejin Choi**[s] **Mike Lewis**[w]
[s]Stanford University [c]University of California at Santa Barbara [p]Prime Intellect
[w]University of Washington
n.muennighoff@gmail.com

Abstract

Test-time scaling uses extra test-time compute to improve performance, such as letting language models reason longer when solving a problem. As models keep the entire reasoning trace in memory via full attention, hard tasks that need long thinking can be prohibitively expensive. However, we find most intermediate reasoning tokens lose importance as the model continues reasoning. This calls into question whether retaining them is worth the cost. Based on this insight, we propose Prefix Sliding, which discards tokens during reasoning that are not part of the prefix or the window of the last few thousand tokens. The prefix has key instructions and tools available to the model, while the most recent tokens are the current reasoning the model is working on. This caps the total memory requirement regardless of how long the model reasons, allowing for efficient long-horizon test-time scaling. Without training, Prefix Sliding can make existing models 3x faster while maintaining performance. Training with Prefix Sliding using reinforcement learning can achieve better performance by enabling scaling to reasoning traces beyond a hundred thousand tokens. Ablations show Prefix Sliding outperforms summarizing intermediate tokens or vanilla sliding window. Our code is at https://github.com/Muennighoff/prefix-sliding

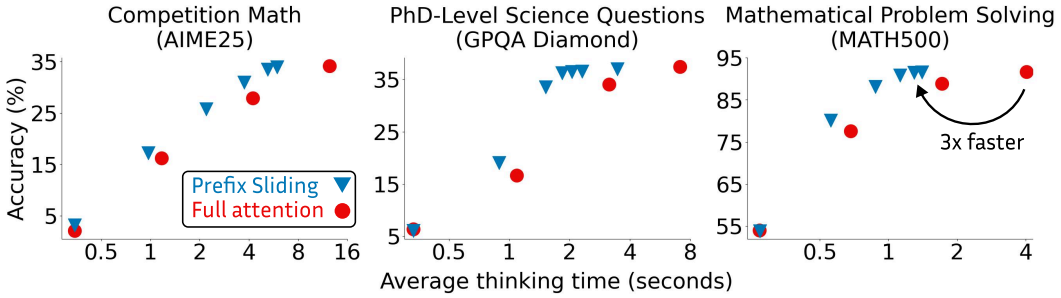


Figure 1: **Prefix Sliding without any training is more efficient than full attention.** Prefix Sliding performs better as it can generate more tokens in the same thinking time than full attention, not because each token it generates is better. Both use the same Qwen3 model (section 3). Here, Prefix Sliding uses a window size of 4096; see Appendix B for other sizes, and Appendix C for the results in tabular form.

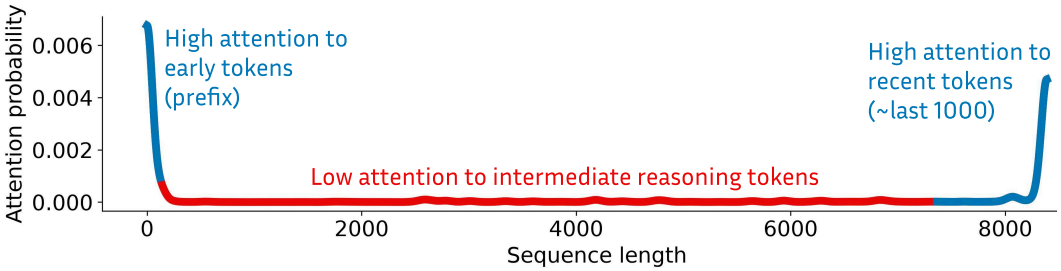


Figure 2: **The first and last few tokens receive most attention during reasoning with full attention.** We plot post-softmax attention probabilities averaged across layers and attention heads in Qwen3-1.7B for an AIME25 reasoning trace. Probabilities are Gaussian smoothed.

1 Introduction

Test-time scaling improves the performance of language models by using extra compute for hard problems (OpenAI, 2024). Commonly, this compute is used by letting the model reason longer (DeepSeek-AI et al., 2025a; Muennighoff et al., 2025). However, scaling this approach further is limited by the need to keep the entire reasoning trace in memory via full attention, as used in most language models (Sadhukhan et al., 2025). With full attention, the cost of each new token grows linearly with the number of already generated tokens, making long context windows prohibitively expensive. Long contexts have more issues, including distraction by old irrelevant tokens (Gema et al., 2025), context poisoning (Comanici et al., 2025), repetitive loops (Pipis et al., 2025), and lost knowledge (Liu et al., 2023b).

We explore a simple solution based on two observations. First, intermediate reasoning tokens quickly lose importance. For example, when solving an expression like "((42 + 84) × 4) - 5", once the addition 42 + 84 is completed, the reasoning behind that step is no longer needed; only the result matters for the next operation. Second, the prefix and the most recent reasoning tokens, however, are of high importance during generation. The prefix, which includes the system instruction and prompt, contains key information about tools the model can use and the task to complete. It also serves as an "attention sink" allowing the model to allocate excess probability weight (Xiao et al., 2024c). Meanwhile, the most recent reasoning tokens capture what the model is currently working on. This has motivated prior work on letting models generate using a sliding window (Ainslie et al., 2020; Gupta & Berant, 2020; Beltagy et al., 2020; Zaheer et al., 2021; Zhang et al., 2025b; Fu et al., 2025c).

We combine these two observations to propose Prefix Sliding. During reasoning, Prefix Sliding keeps only the prefix and a sliding window in memory. The prefix contains the model instructions. As the model generates, the sliding window advances, removing older intermediate tokens. For example, a 40-token system instruction and a 60-token task prompt could constitute a 100-token prefix. With a 4096-token sliding window, at most 4196 tokens are then kept in memory. The cost of generating an additional token is the same regardless of whether the model has already generated millions or billions of tokens. Such constant cost is necessary to enable very long-horizon test-time scaling.

Empirically, Prefix Sliding can match full-attention performance while running 3× faster without training, and enables reinforcement learning rollouts beyond 100,000 tokens.

2 Prefix Sliding

Motivation In Figure 2, we depict the two observations from section 1: (1) Intermediate tokens lack importance and (2) the prefix and recent tokens are key. For the prefix, much probability mass falls on the first four tokens as they function as attention sinks (Xiao et al., 2024c). The other tokens in the prompt also receive more attention than later intermediate reasoning tokens. The common `<think>` delimiter (DeepSeek-AI et al., 2025a) marking the start of the reasoning trace also receives high attention, likely because of its ongoing important role in signaling to the model that it is in thinking mode. The attention probabilities then drop throughout the reasoning trace, but increase sharply toward the end, especially for the token preceding the one being generated.

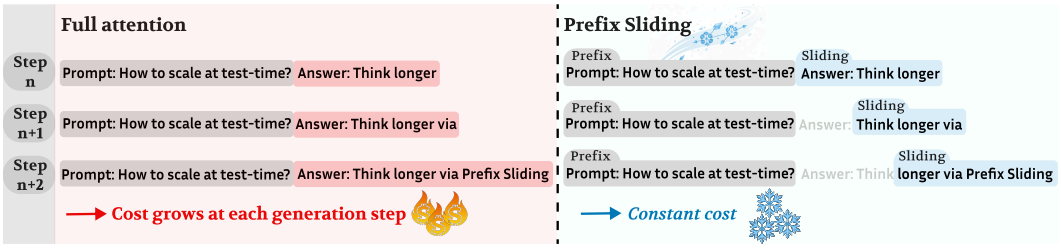


Figure 3: **Prefix Sliding enables efficient long-horizon test-time scaling while full attention inevitably becomes prohibitively expensive.** For Prefix Sliding, the cost at each generation step is constant as attention is only paid to the prefix and sliding window tokens. For full attention, the cost grows at each step due to more tokens in the attention window.

Prefix Sliding without training Figure 3 shows how Prefix Sliding works by simply retaining the prefix of tokens and a sliding window. This makes it applicable to generative language models out of the box without any further training. If the model has been trained with position embeddings (PE), such as RoPE (Su et al., 2023), then Figure 4 shows two options for handling them. Compared to Reset PE, Continue PE is more efficient as it does not require reapplying new position embeddings to the same token, but allows reusing cached representations with position embeddings already applied to them. While Continue PE may perform worse (Xiao et al., 2024c), we have found performance differences insignificant in Appendix D; thus, we use Continue PE. For an even simpler alternative, future work may combine Prefix Sliding with DroPE to simply remove the positional embeddings of pretrained models (Gelberg et al., 2025) or train models without position embeddings from scratch (Kazemnejad et al., 2023).

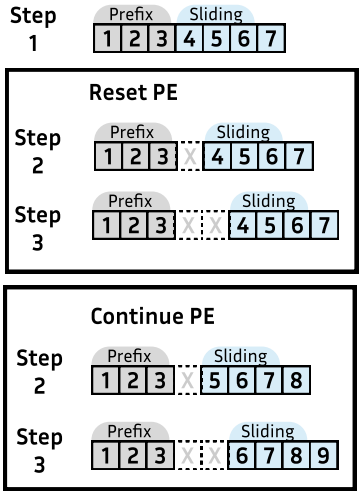


Figure 4: **Position embeddings (PE) with Prefix Sliding.**

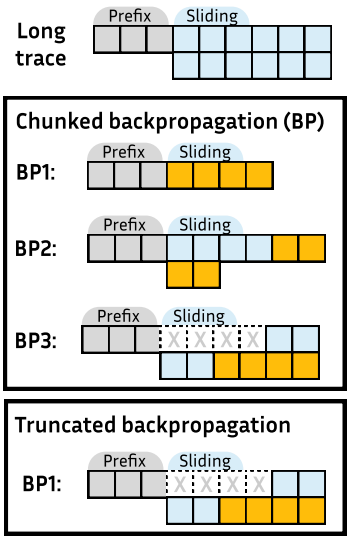


Figure 5: **Backpropagating long reasoning traces with Prefix Sliding.** Yellow marks backpropagated tokens.

Prefix Sliding with training Training with Prefix Sliding enables very long RL rollouts, avoiding the common practice of truncating and discarding overlong generations (Yu et al., 2025). Training on completed generations can substantially improve the model. This is best accomplished in an asynchronous RL setup to avoid idle GPUs when other short generations in the same batch are already done (Fu et al., 2025b; Noukhovitch et al., 2025), but also works with synchronous RL. Naive backpropagation of generations that span hundreds of thousands of tokens can lead to out-of-memory errors in the trainer. Figure 5 depicts two solutions for this issue. Both rely on the limited receptive field of sliding windows. Sliding windows across multiple layers have a theoretical receptive field of $W \times L$, where W is the window size and L the number of layers. However, due to information bottlenecks, it is closer to $1.5 \times W$ in practice (Xiao, 2025). Thus, if we want to backpropagate a set of W tokens, we may only need to pass around $1.5 \times W$ of preceding tokens and the prefix to the trainer. Chunked backpropagation backpropagates on a reasoning chain in chunks and accumulates the gradients to ensure near-equivalence with standard full backpropagation. Truncated backpropagation involves only backpropagating on the last chunk. We use Prefix Sliding with truncated

backpropagation for our training experiments, as we found its performance can match full attention with full backpropagation in Appendix E. For example, a model generated a reasoning trace of 100,000 tokens with a sliding window size of 2048. Under truncated backpropagation, we may only send the last 8192 tokens from the sampler to the trainer for gradient computation. We then use the first 6144 tokens only as context and compute the token-level RL loss only on the final 2048 tokens. In our implementation, this is simply a loss mask: the loss for the preceding 6144 tokens is set to zero, and autograd backpropagates normally from the masked loss and only updates with respect to the last 2048 tokens. As the gradients of those 2048 tokens were computed using $4\times$ the sliding window size, they are very accurate relative to the full 100,000-token generation. Like for Prefix Sliding without training, we also use Continue PE for Prefix Sliding with training. Resetting PE in the trainer is very complex due to teacher-forcing, which is key for training efficiency (Lamb et al., 2016; Brown et al., 2020). This is because when resetting PE, each token has seen a different combination of positions before it.

Prefix Sliding kernel implementation We implement the Prefix Sliding attention kernel with two-level filtering:

- **Intra-tile masking:** For tiles that partially overlap the allowed attention region (prefix $\cup$ sliding window), we apply an elementwise mask so that only valid (q,k) pairs contribute to the softmax and output. This ensures mathematical correctness without changing the FlashAttention tiling strategy.
- **Inter-tile skipping:** We skip tiles that fall entirely outside the allowed region. Concretely, we restructure the producer–consumer pipeline to iterate over two disjoint block ranges (prefix blocks and window blocks). This avoids redundant loads and computations, substantially matching the efficiency of standard sliding window attention.

3 Setup

Modeling We use the Qwen3-1.7B model unless otherwise specified (Yang et al., 2025). We use vLLM (Kwon et al., 2023) with FlashAttention (Dao et al., 2022; Dao, 2023) for all generations. We write custom kernels for the Nvidia Hopper architecture to enable running Prefix Sliding with FlashAttention. We experiment with sliding window sizes 512, 1024, 2048, 4096, 8192, and 16384. For summary ablations, we treat the summary as a tool call and let the model itself write the summary instead of an external model.

Training For reinforcement learning experiments, we use GRPO (Shao et al., 2024) via its synchronous implementation in trl (von Werra et al., 2020), as well as its asynchronous implementation in prime-rl (Intellect, 2025). We either backpropagate the entire generation or only the last sliding window of tokens, as explained in section 2. In the latter case, we always pass four times as many of the last tokens to the trainer to ensure we compute accurate gradients for the sliding window. We do not tune other hyperparameters not directly related to Prefix Sliding (e.g., learning rate) and fix them across comparisons. We create our own dataset of math problems and filter it using guessability, verifiability, and difficulty as our three criteria. Details are in Appendix F.

Evaluation We evaluate on standard reasoning benchmarks: GPQA (Rein et al., 2023), MATH500 (Hendrycks et al., 2021; Lightman et al., 2023), AIME25 (Mathematical Association of America, 2025). We average results across 64 runs to increase confidence in our results, which we refer to as either accuracy or avg@64. We use a temperature of 0.6 and top p of 0.95 (DeepSeek-AI et al., 2025a). We verify answers using a small verification library called simpleverify. We use budget forcing to keep generations to specific thinking budgets (without the use of "Wait" tokens) (Muennighoff et al., 2025). We benchmark models by their average thinking time per sample, measured in seconds, as speed is what users ultimately experience, making it the most important efficiency metric. FLOPs or total generated tokens can be a good proxy, but they miss memory differences among methods.

4 Results

Prefix Sliding without training Figure 1 shows Prefix Sliding is more efficient even when applied to an existing model that has been trained with full attention. Figure 6 shows our FlashAttention kernel for Prefix Sliding reaches about the same speeds as a regular sliding window kernel. A slightly slower speed is expected due to the additional memory requirements of the prefix. The tokens per second for Prefix Sliding and regular sliding window drop initially before stabilizing around 5,000. The initial drop is due to the generation spending less time in the cheap warm-up phase, where the generated tokens are still less than the sliding window. Once the sliding window size is reached, generating each new token costs the same amount. Meanwhile, full attention keeps getting slower indefinitely as the cost per token progressively increases because all prior tokens need to be in memory.

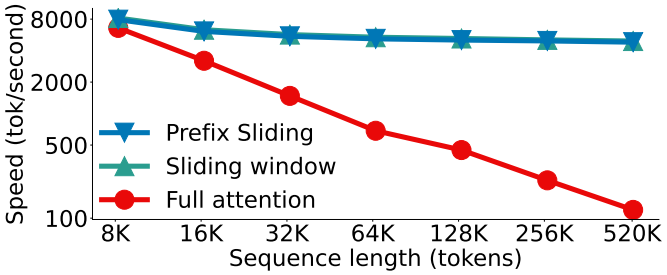


Figure 6: **Prefix Sliding faster than full attention.** We generate 1024 sequences of the respective sequence length with vLLM using its auto batch size, FlashAttention, one 80GB Nvidia H100 GPU, and a window size of 4096 tokens for sliding methods.

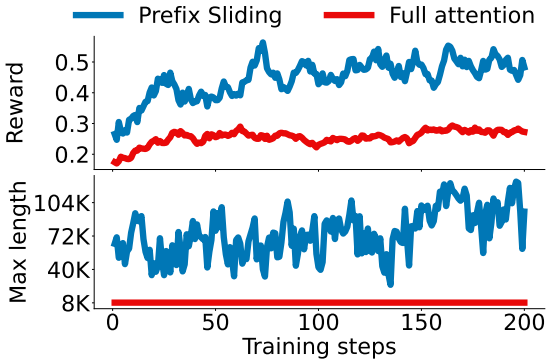


Figure 7: **Prefix Sliding can improve performance.** We restrict both to near-equal memory budgets: 8,192 max tokens for full attention and an 8,192 sliding window size for Prefix Sliding.

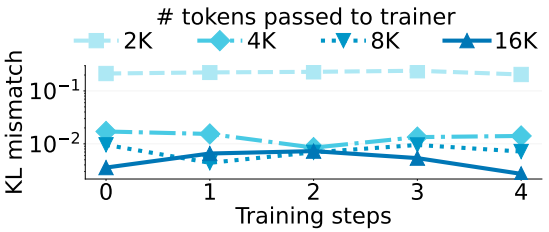


Figure 8: **Truncated backpropagation numerics.** We compute the Kullback–Leibler (KL) divergence between generator and trainer per-token log probabilities of the last 2048 generated tokens; lower is better. All generate with a max sequence length of 16384 tokens, with a subset passed to the trainer (2K, 4K, 8K, or all 16K). All use Prefix Sliding with a window size of 2048.

Prefix Sliding with training Figure 7 shows that reinforcement learning with Prefix Sliding allows for much longer reasoning traces at near-equal memory budgets, thereby leading to higher rewards. Figure 8 ablates the choice of tokens passed to the trainer as explained in section 2. The runs with Prefix Sliding backpropagate one final sliding window (2K tokens). As sliding windows have a limited receptive field (section 2), only a limited number of tokens prior to this final sliding window are necessary to ensure accurate log probabilities. Only passing the sliding window itself leads to a high KL of above 0.1 as expected. Passing $2\times$ as much (4K) significantly lowers the mismatch. We go with $4\times$ (8K) for most runs as it appears to have a slightly lower KL mismatch and is around on par with $8\times$ (16K). Some remaining KL mismatch is expected due to tiny numerical differences between our custom Flash Attention kernel in the generator and our FlexAttention (Dong et al., 2024) implementation in the trainer. We find the multiplier of $4\times$ the window size also works well for larger windows. In Appendix E, we train a 7B model with RL using a window of 8,192 and a multiplier of 4, and find performance comparable to full attention when controlling for sequence length.

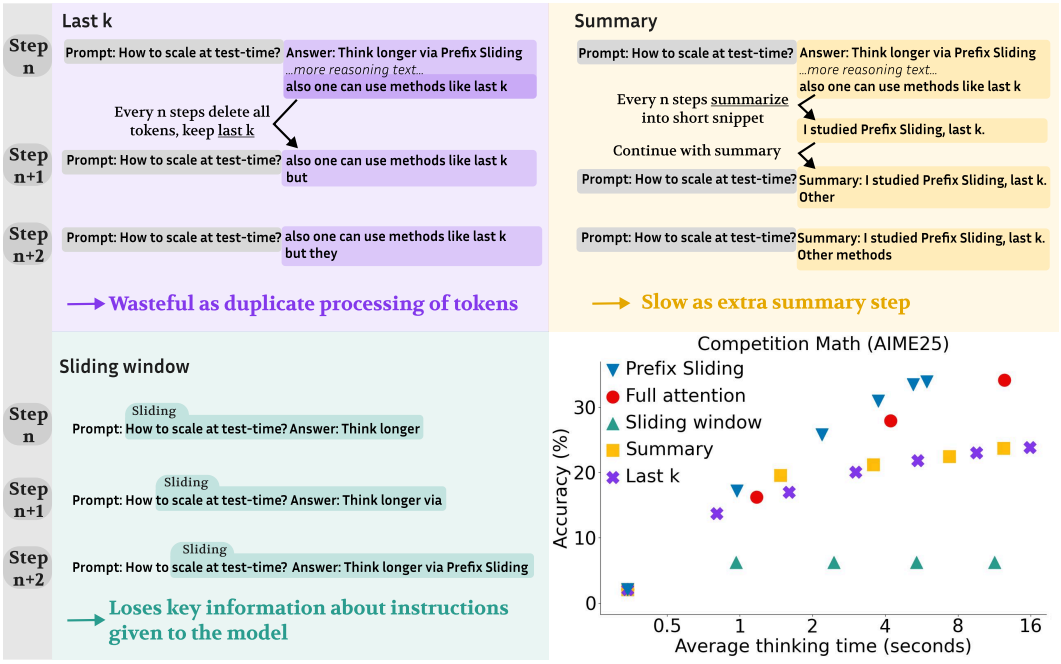


Figure 9: **Prefix Sliding outperforms test-time scaling alternatives.** See Figure 3 for visual explanations of Prefix Sliding and full attention. For benchmarking on AIME25, we use a maximum generation length of 262144 and a local window of 4096 for all methods except full attention. Thus, after generating 4096 tokens, the window either slides or is restarted with the last k tokens or summary. We set k and the maximum summary length to 256 tokens. See section 3 and Appendix G for additional hyperparameters.

5 Ablations

We compare Prefix Sliding with three key alternatives, as shown in Figure 9 and described below. In Appendix G, we provide details on their hyperparameter selection and contrast Prefix Sliding with an additional cache-eviction method.

Last k

- **Explanation** Text is generated until a threshold n is reached, when all text except for the last k tokens is deleted. This way, the context never exceeds the length of the prompt + n. As long as n is reasonably small, last k can use full attention without generation inevitably becoming too expensive to continue.
- **Pros and Cons** Last k can be very fast in terms of tokens per second. However, many of those tokens may be wasted. If k is large, the model has to reprocess a lot of tokens. This is because the last k tokens are processed twice: first upon generation and second when their context window changes due to the removal of prior tokens. If k is small, some of the more recent useful tokens may be dropped, and the model may need to regenerate parts of them (e.g., Figure 21). Last k also incurs volatile memory usage; memory usage drops drastically whenever the context is cleared, making it harder to use compute resources optimally.
- **Examples** Variants of last k are often used in agents: For example, after n turns, earlier turns can be deleted, leaving only the most recent few turns (e.g., Wang et al., 2025b). This approach has also been proposed as "Markovian Thinking / Delethink" in Aghajohari et al. (2025).

Summary

- **Explanation** Text is generated until a threshold n is reached, when all text is summarized, either by the model itself or an external summarizer. Together with the prompt, this summary is then used to start a new context window to continue reasoning. The procedure repeats when n is reached again.
- **Pros and Cons** A benefit of this approach is the model can draw information from anywhere in the current context window for its summary, which in theory could allow it to retain important insights over many summary steps. In practice, however, models struggle to retain important information over many turns (Wang et al., 2026a). Summary adds complexity by introducing new hyperparameters, such as n, the summary length, the summarizer prompt, the summarizer model, and the placement of the summary in the new context. Further, the extra summary-generation step adds overhead, especially if the summary model is large. Like last k, the summary must be processed twice, first upon generation and second when used in the new context window. Its memory usage is also volatile like last k.
- **Examples** This approach has been explored via prompting (Vajipey et al., 2025), supervised finetuning (Yan et al., 2025; Kontonis et al., 2026), or reinforcement learning (Wu et al., 2025b; Yan et al., 2026; Li et al., 2026b). Wu et al. (2021) use a hierarchical, rather than sequential, version of this approach to handle long inputs. It is also referred to as compaction and used by models like Opus 4.6 (Anthropic, 2026), GPT 5.4 (OpenAI, 2025b), and Composer (Cursor Research et al., 2026).

Sliding window

- **Explanation** This is a baseline equivalent to Prefix Sliding without prefix.
- **Pros and Cons** The method is very simple. However, as the prefix contains key information about the task, this method performs poorly on longer reasoning tasks. The model forgets which problem it is solving or which tools it can use.
- **Examples** Sliding window attention (Beltagy et al., 2020) is common in many large language models, such as gpt-neo (Black et al., 2021) and gpt-oss (OpenAI, 2025a). To compensate for the lack of information about the prefix, they interleave it with full attention layers that process the entire context.

Results Figure 9 shows Prefix Sliding provides the best performance efficiency trade-off. Prefix Sliding also adds only one hyperparameter: the size of the sliding window. Pure sliding window attention quickly flattens out due to a lack of information about the task at long thinking times. As soon as the model reaches the sliding window size, it starts losing tokens at the beginning that contain critical information. Last k and summary approaches can reach good performance but are fundamentally constrained by their required token reprocessing and extra summary step. They also add other complexity overhead by requiring more hyperparameters and generation restarts.

6 Related Work

Test-time scaling Current methods to scale compute at test-time are either **sequential** or **parallel** (Snell et al., 2024; Muennighoff et al., 2025). Parallel methods allow for infinite test-time scaling by design, e.g., majority voting (Wang et al., 2023) simply requires launching more parallel processes to try to solve the same question. However, they face stark diminishing returns (Brown et al., 2024; Ehrlich et al., 2025; Schaeffer et al., 2025). Sequential scaling can scale better than parallel (Muennighoff et al., 2025). While there has been much work on improving sequential scaling and reasoning models in general (Zhang et al., 2025a; Aggarwal & Welleck, 2025; Yue et al., 2025; Yong et al., 2025; Lu et al., 2025b; Guha et al., 2025), long-horizon scaling remains a fundamental limitation due to the quadratic complexity of the transformer (Vaswani et al., 2017). We build a simple method that significantly improves reasoning efficiency while enabling long-horizon scaling due to its constant cost, as elaborated in the next paragraph.

Context extension We distinguish between methods that are **bounded** and **unbounded** in their cost per new token. Bounded methods are asymptotically constant; in the limit, they cost at most a certain amount per new token (Peng et al., 2022; Munkhdalai et al., 2024; Yang et al., 2024). Unbounded methods cost more for each new token in the limit, even if they may exhibit subquadratic complexity (Child et al., 2019; Kitaev et al., 2020; Wang et al., 2020; Jaegle et al., 2021; Xiong et al., 2021; Choromanski et al., 2022; Liu et al., 2023a; Ho et al., 2024; DeepSeek-AI et al., 2025b; Shyam et al., 2025). Full attention is unbounded: Every generated token gets more expensive. One can trade off space and time complexity of full attention to make one bounded, but the other stays unbounded (Rabe & Staats, 2022). Crucially, to enable infinite test-time scaling, i.e., models that reason for weeks, cost must be bounded per new token. Figure 10 shows Prefix Sliding is bounded once the sequence length reaches the combined size of the prefix and the sliding window. Other bounded methods include RNNs (e.g. RWKV (Peng et al., 2023; 2024)), SSMs (e.g. Mamba (Gu & Dao, 2024; Dao & Gu, 2024; Gu et al., 2022; Wang et al., 2024)), and transformer-based approaches (Dai et al., 2019; Rae et al., 2019; Chevalier et al., 2023; Tandon et al., 2025). However, they do not work out of the box with existing models but require training models to adapt them. Prefix Sliding works with existing pretrained models without further training and can optionally also be used for training. Last k and summary approaches are also bounded and work out of the box, but exhibit irregular cost as shown in Figure 10 due to deleting and refilling of the context window. This makes full GPU utilization difficult. They also incur fundamental latency overhead due to duplicate token processing, which Prefix Sliding bypasses, as detailed in section 5. One way to view Prefix Sliding is as sliding window attention with global tokens (Beltagy et al., 2020; Han et al., 2024), but with many consecutive global tokens forming a prefix that preserves full task instructions and other necessary context. Another related approach is StreamingLLM (Xiao et al., 2024c), which retains only a few fixed initial tokens, e.g., 4.

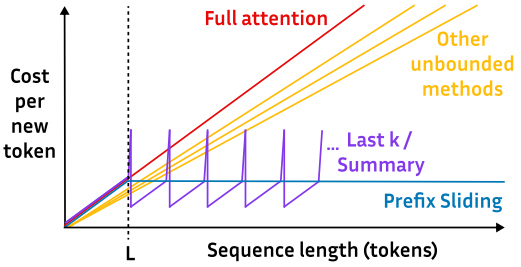


Figure 10: **Prefix Sliding has constant cost per new token in the limit.** "L" is where the sequence length reaches the size of the sliding window + the prefix. Last k and summary follow a sawtooth pattern with a cost spike at each chunk end, as the last k tokens need to be passed over or the entire chunk summarized.

7 Conclusion

We propose Prefix Sliding to enable language models to reason for extremely long horizons. Even at short reasoning horizons of only thousands of tokens, Prefix Sliding is more efficient than the status quo of using full attention. Prefix Sliding is applicable to language models without further training. It can also be used during training with reinforcement learning. It outperforms alternatives that could also support infinite test-time scaling. We hope that enabling language models to think longer via Prefix Sliding inspires future work on solving ever harder problems with language models.

Limitations

Limited comparisons We restrict ourselves to empirical comparisons with alternatives that fulfill two properties: (1) they work on existing pretrained transformers out of the box and (2) they lead to a bounded cost per new token (see section 6). This excludes many approaches, such as alternative architectures, subquadratic methods, or mixed sliding window models (Tay et al., 2020; Bulatov et al., 2022; Hwang et al., 2024; He et al., 2025b; Li et al., 2025c). Future work may consider relaxing the first criterion by comparing with alternative architectures that still exhibit a bounded cost per new token, such as RNNs. We consider it beyond the scope of this work, as it likely requires pretraining models from scratch to control for computational resources and other hyperparameters.

Information loss While intermediate tokens can lack importance for later reasoning, as we show in section 2, sometimes this is not the case. Figure 11 shows this limitation on the example of LiveCodeBench, where a larger window size is necessary to match full attention. Inspecting samples reveals that the issue is likely that for LiveCodeBench, the model starts a function implementation during reasoning and then thinks using comments for potentially thousands of tokens (see Figure 22 for an example). By the time it continues coding, the beginning of the code may have moved outside its sliding window. This evaluation is without any training. Training with Prefix Sliding during reinforcement learning would likely teach the model to simply adapt its commenting behavior, thus enabling a shorter window size. Alternatively, a mechanism for the model to append sliding tokens to the prefix, or another knowledge store, may avoid the need for larger window sizes.

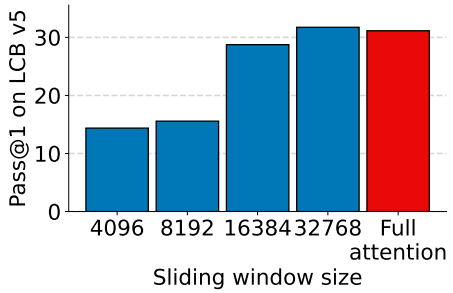


Figure 11: **LiveCodeBench (LCB) requires Prefix Sliding with a window of at least 16384 to match full attention.** Max tokens are 262144.

Limited benefit for short generations As is clear from Figure 6, the benefits of Prefix Sliding are larger the longer the generation of the model. For short generations, a larger proportion of the generation still uses full attention while the sliding window size has not yet been reached. We call this the sliding window warm-up phase. Only after this phase is complete does the window slide and evict old tokens, thereby offering major speed-ups over full attention. In Figure 12, we benchmark Prefix Sliding on a task that requires only 2086 tokens on average: HealthBench (Arora et al., 2025). As we use a sliding window of 2048 for Prefix Sliding, the model slides very rarely. It is equivalent to full attention for the many samples that require fewer than 2048 tokens. Thus, there is little room for any speed-up.

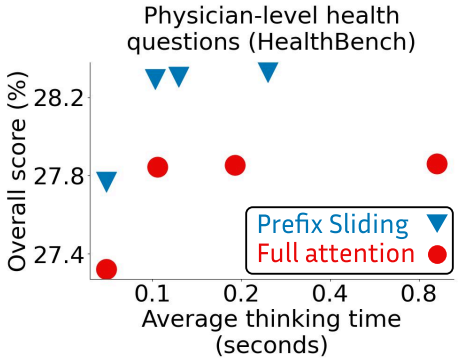


Figure 12: **Fast tasks benefit less from Prefix Sliding.** The sliding window size is 2048 for Prefix Sliding.

System outputs and multi-turn In agentic tasks, a model may read the contents of a website or read files, which could flood the entire context window. This could be problematic because if the sliding window is smaller than the content the model is trying to read, then it strictly cannot read the entire content. Even worse, it may lose important content, as its sliding window is flooded with this new output. A second related issue is what to do with future user instructions in a multi-turn setup. Append them to the prefix? Let the sliding window remove them eventually? These two problems also exist with summarization or last k techniques, assuming the same window size. Extensive reinforcement learning would likely teach the model to be extra careful to avoid this behavior. Another approach could be to let the model learn to read content step by step rather than in one go (e.g., using "head" rather than "cat" commands in Unix). One can also add automatic guardrails that prevent excessive outputs in the model's context by quickly checking such outputs before and not providing them to the model beyond a prespecified threshold.

Scale In this work, we scale up Prefix Sliding to hundreds of thousands of thinking tokens and 7 billion parameter models across training-free and reinforcement learning training setups. Future work is necessary to scale up Prefix Sliding further and study its trends.

Reproducibility Statement

As Prefix Sliding is very simple and section 2 describes it in detail, it is likely easy to reproduce our key results using only the paper. We also make our code public at https://github.com/Muennighoff/prefix-sliding.

Author Contributions

Niklas Muennighoff ran training, evaluation, wrote the paper, led the project. Zhengyang Wang, Niklas Muennighoff worked on kernels. Zeyi Chen, Niklas Muennighoff, Dapeng Jiang implemented ablations. Niklas Muennighoff, John Yang, Weijia Shi implemented evaluation. Niklas Muennighoff, Binyuan Hui, John Yang made datasets. Mike Lewis, Yejin Choi, Luke Zettlemoyer, Weijia Shi, Andrew Y. Ng, Jason Wei, Percy Liang, Ludwig Schmidt, Sami Jaghouar, Johannes Hagemann, Fares Obeid, Mika Senghaas advised the project.

Acknowledgments

We are extremely thankful to Laude Institute for supporting this work. Research supported by the NVIDIA Academic Grant Program. NM is supported by a graduate fellowship award from Knight-Hennessy Scholars at Stanford University. This work was supported by IITP funded by the Korean Government (MSIT) (No. RS-2024-00457882, National AI Research Lab Project). This research was supported in part by a gift from DSO National Laboratories. We thank Hunter Lang for insightful discussions.

References

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.04697.

Milad Aghajohari, Kamran Chitsaz, Amirhossein Kazemnejad, Sarath Chandar, Alessandro Sordoni, Aaron Courville, and Siva Reddy. The markovian thinker, 2025. URL https://arxiv.org/abs/2510.06557.

Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. Etc: Encoding long and structured inputs in transformers, 2020. URL https://arxiv.org/abs/2004.08483.

Yash Akhauri, Ahmed F AbouElhamayed, Yifei Gao, Chi-Chih Chang, Nilesh Jain, and Mohamed S. Abdelfattah. Tokenbutler: Token importance is predictable, 2025. URL https://arxiv.org/abs/2503.07518.

Anthropic. Introducing claude opus 4.6, 2026. URL https://www.anthropic.com/news/claude-opus-4-6.

Daman Arora, Himanshu Gaurav Singh, and Mausam. Have llms advanced enough? a challenging problem solving benchmark for large language models, 2023. URL https://arxiv.org/abs/2305.15074.

Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. Healthbench: Evaluating large language models towards improved human health, 2025. URL https://arxiv.org/abs/2505.08775.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

Adithya Bhaskar, Alexander Wettig, Tianyu Gao, Yihe Dong, and Danqi Chen. Cache me if you can: How many kvs do you need for effective long-context lms?, 2025. URL https://arxiv.org/abs/2506.17121.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow, 2021. URL https://doi.org/10.5281/zenodo.5297715.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. Recurrent memory transformer, 2022. URL https://arxiv.org/abs/2207.06881.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling, 2025. URL https://arxiv.org/abs/2406.02069.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Anima Anandkumar, Abedelkadir Asi, and Junjie Hu. R-kv: Redundancy-aware kv cache compression for reasoning models, 2026. URL https://arxiv.org/abs/2505.24133.

Ting-Yun Chang, Harvey Yiyun Fu, Deqing Fu, Chenghao Yang, Jesse Thomason, and Robin Jia. Value-aware stochastic kv cache eviction for reasoning models, 2026. URL https://arxiv.org/abs/2606.03928.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. Theoremqa: A theorem-driven question answering dataset, 2023. URL https://arxiv.org/abs/2305.12524.

Yanda Chen, Joe Benton, Ansh Radhakrishnan, Jonathan Uesato, Carson Denison, John Schulman, Arushi Somani, Peter Hase, Misha Wagner, Fabien Roger, Vlad Mikulik, Samuel R. Bowman, Jan Leike, Jared Kaplan, and Ethan Perez. Reasoning models don't always say what they think, 2025. URL https://arxiv.org/abs/2505.05410.

Yilong Chen, Guoxia Wang, Junyuan Shang, Shiyao Cui, Zhenyu Zhang, Tingwen Liu, Shuohuan Wang, Yu Sun, Dianhai Yu, and Hua Wu. Nacl: A general and effective kv cache eviction framework for llms at inference time, 2024. URL https://arxiv.org/abs/2408.03675.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts, 2023. URL https://arxiv.org/abs/2305.14788.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019. URL https://arxiv.org/abs/1904.10509.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022. URL https://arxiv.org/abs/2009.14794.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Cursor Research, Aaron Chan, Ahmed Shalaby, Alexander Wettig, Aman Sanger, Andrew Zhai, Anurag Ajay, Ashvin Nair, Charlie Snell, Chen Lu, Chen Shen, Emily Jia, Federico Cassano, Hanpeng Liu, Haoyu Chen, Henry Wildermuth, Jacob Jackson, Janet Li, Jediah Katz, Jiajun Yao, Joey Hejna, Josh Warner, Julius Vering, Kevin Frans, Lee

Danilek, Less Wright, Lujing Cen, Luke Melas-Kyriazi, Michael Truell, Michiel de Jong, Naman Jain, Nate Schmidt, Nathan Wang, Niklas Muennighoff, Oleg Rybkin, Paul Loh, Phillip Kravtsov, Rishabh Yadav, Sahil Shah, Sam Kottler, Alexander M Rush, Shengtong Zhang, Shomil Jain, Sriram Sankar, Stefan Heule, Stuart H. Sul, Sualeh Asif, Victor Rong, Wanqi Zhu, William Lin, Yuchen Wu, Yuri Volkov, Yury Zemlyanskiy, Zack Holbrook, and Zhiyuan Zhang. Composer 2 technical report, 2026. URL https://arxiv.org/abs/2603.24477.

Muzhi Dai, Chenxu Yang, and Qingyi Si. S-grpo: Early exit via reinforcement learning in reasoning models, 2025. URL https://arxiv.org/abs/2505.07686.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL https://arxiv.org/abs/1901.02860.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning how hard to think: Input-adaptive allocation of lm computation, 2024. URL https://arxiv.org/abs/2410.04707.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025a. URL https://arxiv.org/abs/2501.12948.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models, 2025b. URL https://arxiv.org/abs/2512.02556.

Harry Dong, Bilge Acun, Beidi Chen, and Yuejie Chi. Scalable llm reasoning acceleration with low-rank distillation, 2026. URL https://arxiv.org/abs/2505.07861.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels, 2024. URL https://arxiv.org/abs/2412.05496.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering, 2025. URL https://arxiv.org/abs/2501.14723.

Sabri Eyuboglu, Ryan Ehrlich, Simran Arora, Neel Guha, Dylan Zinsley, Emily Liu, Will Tennien, Atri Rudra, James Zou, Azalia Mirhoseini, and Christopher Re. Cartridges: Lightweight and general-purpose long context representations via self-study, 2025. URL https://arxiv.org/abs/2506.06266.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023. URL https://arxiv.org/abs/2210.17323.

Tianyu Fu, Haofeng Huang, Xuefei Ning, Genghan Zhang, Boju Chen, Tianqi Wu, Hongyi Wang, Zixiao Huang, Shiyao Li, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Mixture of attention spans: Optimizing llm inference efficiency with heterogeneous sliding-window lengths, 2025a. URL https://arxiv.org/abs/2406.14909.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. Areal: A large-scale asynchronous reinforcement learning system for language reasoning, 2025b. URL https://arxiv.org/abs/2505.24298.

Zichuan Fu, Wentao Song, Yejing Wang, Xian Wu, Yefeng Zheng, Yingying Zhang, Derong Xu, Xuetao Wei, Tong Xu, and Xiangyu Zhao. Sliding window attention training for efficient large language models, 2025c. URL https://arxiv.org/abs/2502.18845.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. Omni-math: A universal olympiad level mathematic benchmark for large language models, 2024. URL https://arxiv.org/abs/2410.07985.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2024. URL https://arxiv.org/abs/2310.01801.

Yoav Gelberg, Koshi Eguchi, Takuya Akiba, and Edoardo Cetin. Extending the context of pretrained llms by dropping their positional embeddings, 2025. URL https://arxiv.org/abs/2512.12167.

Aryo Pradipta Gema, Alexander Hägele, Runjin Chen, Andy Arditi, Jacob Goldman-Wetzler, Kit Fraser-Taliente, Henry Sleight, Linda Petrini, Julian Michael, Beatrice Alex, Pasquale Minervini, Yanda Chen, Joe Benton, and Ethan Perez. Inverse scaling in test-time compute, 2025. URL https://arxiv.org/abs/2507.14417.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022. URL https://arxiv.org/abs/2111.00396.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

Ankit Gupta and Jonathan Berant. Gmat: Global memory augmentation for transformers, 2020. URL https://arxiv.org/abs/2006.03274.

Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. Lm-infinite: Zero-shot extreme length generalization for large language models, 2024. URL https://arxiv.org/abs/2308.16137.

Shen Han, Yuyang Wu, Junpu Yu, and Olexandr Isayev. Kara: Efficient reasoning llm serving via sliding-window kv cache compression, 2026. URL https://arxiv.org/abs/2607.01237.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems, 2024. URL https://arxiv.org/abs/2402.14008.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork open reasoner 1 technical report, 2025a. URL https://arxiv.org/abs/2505.22312.

Zifan He, Yingqi Cao, Zongyue Qin, Neha Prakriya, Yizhou Sun, and Jason Cong. Hmt: Hierarchical memory transformer for efficient long context language processing, 2025b. URL https://arxiv.org/abs/2405.06067.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. URL https://arxiv.org/abs/2103.03874.

Namgyu Ho, Sangmin Bae, Taehyeon Kim, Hyunjik Jo, Yireun Kim, Tal Schuster, Adam Fisch, James Thorne, and Se-Young Yun. Block transformer: Global-to-local language modeling for fast inference, 2024. URL https://arxiv.org/abs/2406.02657.

Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. Thinkprune: Pruning long chain-of-thought of llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2504.01296.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. Raas: Reasoning-aware attention sparsity for efficient llm reasoning, 2025. URL https://arxiv.org/abs/2502.11147.

Zhen Huang, Zengzhi Wang, Shijie Xia, Xuefeng Li, Haoyang Zou, Ruijie Xu, Run-Ze Fan, Lyumanshan Ye, Ethan Chern, Yixin Ye, Yikai Zhang, Yuqing Yang, Ting Wu, Binjie Wang, Shichao Sun, Yang Xiao, Yiyuan Li, Fan Zhou, Steffi Chern, Yiwei Qin, Yan Ma, Jiadi Su, Yixiu Liu, Yuxiang Zheng, Shaoting Zhang, Dahua Lin, Yu Qiao, and Pengfei Liu. Olympicarena: Benchmarking multi-discipline cognitive reasoning for superintelligent ai, 2024. URL https://arxiv.org/abs/2406.12753.

Dongseong Hwang, Weiran Wang, Zhuoyuan Huo, Khe Chai Sim, and Pedro Moreno Mengibar. Transformerfam: Feedback attention is working memory, 2024. URL https://arxiv.org/abs/2404.09173.

Prime Intellect. Prime-rl, 2025. URL https://github.com/PrimeIntellect-ai/prime-rl.

Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. Perceiver: General perception with iterative attention, 2021.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention, 2024. URL https://arxiv.org/abs/2407.02490.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers, 2023. URL https://arxiv.org/abs/2305.19466.

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

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020. URL https://arxiv.org/abs/2001.04451.

Vasilis Kontonis, Yuchen Zeng, Shivam Garg, Lingjiao Chen, Hao Tang, Ziyan Wang, Ahmed Awadallah, Eric Horvitz, John Langford, and Dimitris Papailiopoulos. Memento: Teaching llms to manage their own context, 2026. URL https://arxiv.org/abs/2604.09852.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Alex Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks, 2016. URL https://arxiv.org/abs/1610.09038.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, Karina Nguyen, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Robin Larson, Sam McCandlish, Sandipan Kundu, Saurav Kadavath, Shannon Yang, Thomas Henighan, Timothy Maxwell, Timothy Telleen-Lawton, Tristan Hume, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. Measuring faithfulness in chain-of-thought reasoning, 2023. URL https://arxiv.org/abs/2307.13702.

Chen Li, Nazhou Liu, and Kai Yang. Adaptive group policy optimization: Towards stable training and token-efficient reasoning, 2025a. URL https://arxiv.org/abs/2503.15952.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath, 2024a. URL https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf.

Junyan Li, Wenshuo Zhao, Yang Zhang, and Chuang Gan. Steering llm thinking with budget guidance, 2025b. URL https://arxiv.org/abs/2506.13752.

Michael Y. Li, Jubayer Ibn Hamid, Emily B. Fox, and Noah D. Goodman. Neural garbage collection: Learning to forget while learning to reason, 2026a. URL https://arxiv.org/abs/2604.18002.

Tianjian Li, Jingyu Zhang, William Jurayj, Xi Wang, Chuanyang Jin, Mehrdad Farajtabar, Eric Nalisnick, and Daniel Khashabi. Self-compacting language model agents, 2026b. URL https://arxiv.org/abs/2606.23525.

Wenhao Li, Bangcheng Sun, Weihao Ye, Tianyi Zhang, Daohai Yu, Fei Chao, and Rongrong Ji. Ccf: A context compression framework for efficient long-sequence language modeling, 2025c. URL https://arxiv.org/abs/2509.09199.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024b. URL https://arxiv.org/abs/2404.14469.

Yesheng Liang, Haisheng Chen, Zihan Zhang, Song Han, and Zhijian Liu. Paroquant: Pairwise rotation quantization for efficient reasoning llm inference, 2026. URL https://arxiv.org/abs/2511.10645.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024. URL https://arxiv.org/abs/2306.00978.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation : Learning to solve and explain algebraic word problems, 2017. URL https://arxiv.org/abs/1705.04146.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023a. URL https://arxiv.org/abs/2310.01889.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning, 2020. URL https://arxiv.org/abs/2007.08124.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023b. URL https://arxiv.org/abs/2307.03172.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time, 2023c. URL https://arxiv.org/abs/2305.17118.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025. URL https://arxiv.org/abs/2503.20783.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. Moba: Mixture of block attention for long-context llms, 2025a. URL https://arxiv.org/abs/2502.13189.

Ximing Lu, Seungju Han, David Acuna, Hyunwoo Kim, Jaehun Jung, Shrimai Prabhumoye, Niklas Muennighoff, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, and Yejin Choi. Retro-search: Exploring untaken paths for deeper and efficient reasoning, 2025b. URL https://arxiv.org/abs/2504.04383.

Haotian Luo, Li Shen, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Naiqiang Tan, Xiaochun Cao, and Dacheng Tao. O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning, 2025. URL https://arxiv.org/abs/2501.12570.

Mathematical Association of America. Aime, February 2025. URL https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions/.

Michael R. Metel, Yufei Cui, Boxing Chen, and Prasanna Parthasarathi. Thinking long, but short: Stable sequential test-time scaling for large reasoning models, 2026. URL https://arxiv.org/abs/2601.09855.

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

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention, 2024. URL https://arxiv.org/abs/2404.07143.

Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous rlhf: Faster and more efficient off-policy rl for language models, 2025. URL https://arxiv.org/abs/2410.18252.

OpenAI. Learning to reason with llms, September 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025a. URL https://arxiv.org/abs/2508.10925.

OpenAI. Openai gpt-5 system card, 2025b. URL https://arxiv.org/abs/2601.03267.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023.

Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, Kranthi Kiran GV, Jan Kocoń, Bartłomiej Koptyra, Satyapriya Krishna, Ronald McClelland, Jiaju Lin, Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Stanisław Woźniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence, 2024.

Hao Peng, Jungo Kasai, Nikolaos Pappas, Dani Yogatama, Zhaofeng Wu, Lingpeng Kong, Roy Schwartz, and Noah A. Smith. Abc: Attention with bounded-memory control, 2022. URL https://arxiv.org/abs/2110.02488.

Charilaos Pipis, Shivam Garg, Vasilis Kontonis, Vaishnavi Shrivastava, Akshay Krishnamurthy, and Dimitris Papailiopoulos. Wait, wait, wait... why do reasoning models loop?, 2025. URL https://arxiv.org/abs/2512.12895.

Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022. URL https://arxiv.org/abs/2112.05682.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling, 2019. URL https://arxiv.org/abs/1911.05507.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. Thinkv: Thought-adaptive kv cache compression for efficient reasoning models, 2026. URL https://arxiv.org/abs/2510.01290.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

Ranajoy Sadhukhan, Zhuoming Chen, Haizhong Zheng, Yang Zhou, Emma Strubell, and Beidi Chen. Kinetics: Rethinking test-time scaling laws, 2025. URL https://arxiv.org/abs/2506.05333.

Rylan Schaeffer, Joshua Kazdan, John Hughes, Jordan Juravsky, Sara Price, Aengus Lynch, Erik Jones, Robert Kirk, Azalia Mirhoseini, and Sanmi Koyejo. How do large language monkeys get their power (laws)?, 2025. URL https://arxiv.org/abs/2502.17578.

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

Yi Shen, Jian Zhang, Jieyun Huang, Shuming Shi, Wenjing Zhang, Jiangze Yan, Ning Wang, Kai Wang, Zhaoxiang Liu, and Shiguo Lian. Dast: Difficulty-adaptive slow-thinking for large reasoning models, 2026. URL https://arxiv.org/abs/2503.04472.

Vasudev Shyam, Jonathan Pilault, Emily Shepperd, Quentin Anthony, and Beren Millidge. Tree attention: Topology-aware decoding for long-context attention on gpu clusters, 2025. URL https://arxiv.org/abs/2408.04093.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Liangtai Sun, Yang Han, Zihan Zhao, Da Ma, Zhennan Shen, Baocai Chen, Lu Chen, and Kai Yu. Scieval: A multi-level large language model evaluation benchmark for scientific research, 2024. URL https://arxiv.org/abs/2308.13149.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. End-to-end test-time training for long context, 2025. URL https://arxiv.org/abs/2512.23675.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference, 2024. URL https://arxiv.org/abs/2406.10774.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Vivek Vajipey, Aditya Tadimeti, Justin Shen, Ben Prystawski, Michael Y Li, and Noah Goodman. Simple, scalable reasoning via iterated summarization. 2025. URL https://openreview.net/pdf?id=uhZLKclfGB.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Guangtao Wang, Shubhangi Upasani, Chen Wu, Darshan Gandhi, Jonathan Li, Changran Hu, Bo Li, and Urmish Thakker. Llms know what to drop: Self-attention guided kv cache eviction for efficient long-context inference, 2025a. URL https://arxiv.org/abs/2503.08879.

Junxiong Wang, Tushaar Gangavarapu, Jing Nathan Yan, and Alexander M. Rush. Mambabyte: Token-free selective state space model, 2024. URL https://arxiv.org/abs/2401.13660.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020. URL https://arxiv.org/abs/2006.04768.

Siyuan Wang, Zhongkun Liu, Wanjun Zhong, Ming Zhou, Zhongyu Wei, Zhumin Chen, and Nan Duan. From lsat: The progress and challenges of complex reasoning, 2021. URL https://arxiv.org/abs/2108.00648.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025b. URL https://arxiv.org/abs/2407.16741.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

Zhiqi Wang, Yichi Zhang, Dongwon Lee, and Yuchen Yang. Lost in compaction: Evaluating side-constraint loss under context compaction, 2026a. URL https://arxiv.org/abs/2608.11242.

Zihan Wang, Cheng Tang, Lei Gong, Cheng Li, Chao Wang, Teng Wang, Wenqi Lou, and Xuehai Zhou. Crystal-kv: Efficient kv cache management for chain-of-thought llms via answer-first principle, 2026b. URL https://arxiv.org/abs/2601.16986.

Jeff Wu, Long Ouyang, Daniel M. Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. Recursively summarizing books with human feedback, 2021. URL https://arxiv.org/abs/2109.10862.

Menghua Wu, Cai Zhou, Stephen Bates, and Tommi Jaakkola. Thought calibration: Efficient and confident test-time scaling, 2025a. URL https://arxiv.org/abs/2505.18404.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. Resum: Unlocking long-horizon search intelligence via context summarization, 2025b. URL https://arxiv.org/abs/2509.13313.

Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. Rlboost: Harvesting preemptible resources for cost-efficient reinforcement learning on llms, 2026. URL https://arxiv.org/abs/2510.19225.

Violet Xiang, Chase Blagden, Rafael Rafailov, Nathan Lile, Sang Truong, Chelsea Finn, and Nick Haber. Just enough thinking: Efficient reasoning with adaptive length penalties reinforcement learning, 2025. URL https://arxiv.org/abs/2506.05256.

Guangxuan Xiao. Why stacking sliding windows can't see very far. https://guangxuanx.com/blog/stacking-swa.html, 2025.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024a. URL https://arxiv.org/abs/2211.10438.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads, 2024b. URL https://arxiv.org/abs/2410.10819.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024c. URL https://arxiv.org/abs/2309.17453.

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention, 2021. URL https://arxiv.org/abs/2102.03902.

Yuchen Yan, Yongliang Shen, Yang Liu, Jin Jiang, Mengdi Zhang, Jian Shao, and Yueting Zhuang. Inftythink: Breaking the length limits of long-context reasoning in large language models, 2025. URL https://arxiv.org/abs/2503.06692.

Yuchen Yan, Liang Jiang, Jin Jiang, Shuaicheng Li, Zujie Wen, Zhiqiang Zhang, Jun Zhou, Jian Shao, Yueting Zhuang, and Yongliang Shen. Inftythink+: Effective and efficient infinite-horizon reasoning via reinforcement learning, 2026. URL https://arxiv.org/abs/2602.06960.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024. URL https://arxiv.org/abs/2312.06635.

Zheng-Xin Yong, M. Farid Adilazuarda, Jonibek Mansurov, Ruochen Zhang, Niklas Muennighoff, Carsten Eickhoff, Genta Indra Winata, Julia Kreutzer, Stephen H. Bach, and Alham Fikri Aji. Crosslingual reasoning through test-time scaling, 2025. URL https://arxiv.org/abs/2505.05408.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Yijiong Yu, Jiale Liu, Qingyun Wu, Huazheng Wang, and Ji Pei. Swaa: Sliding window attention adaptation for efficient and quality preserving long context processing, 2026. URL https://arxiv.org/abs/2512.10411.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025. URL https://arxiv.org/abs/2502.11089.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?, 2025. URL https://arxiv.org/abs/2504.13837.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021. URL https://arxiv.org/abs/2007.14062.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL https://arxiv.org/abs/2512.24601.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, and Chen Ma. A survey on test-time scaling in large language models: What, how, where, and how well?, 2025a. URL https://arxiv.org/abs/2503.24235.

Xuan Zhang, Fengzhuo Zhang, Cunxiao Du, Chao Du, Tianyu Pang, Wei Gao, and Min Lin. Lighttransfer: Your long-context llm is secretly a hybrid model with effortless adaptation, 2025b. URL https://arxiv.org/abs/2410.13846.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023. URL https://arxiv.org/abs/2306.14048.

Haoran Zhao, Yuchen Yan, Yongliang Shen, Haolei Xu, Wenqi Zhang, Kaitao Song, Jian Shao, Weiming Lu, Jun Xiao, and Yueting Zhuang. Let lrms break free from overthinking via self-braking tuning, 2025. URL https://arxiv.org/abs/2505.14604.

Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for llm reasoning via selective rollouts, 2025. URL https://arxiv.org/abs/2506.02177.

Haoxi Zhong, Chaojun Xiao, Cunchao Tu, Tianyang Zhang, Zhiyuan Liu, and Maosong Sun. Jec-qa: A legal-domain question answering dataset, 2019. URL https://arxiv.org/abs/1911.12011.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. Agieval: A human-centric benchmark for evaluating foundation models, 2023. URL https://arxiv.org/abs/2304.06364.

Yang Zhou, Ranajoy Sadhukhan, Zhaofeng Sun, Zhuoming Chen, Souvik Kundu, Saket Dingliwal, Sai Muralidhar Jayanthi, Aram Galstyan, Haizhong Zheng, and Beidi Chen. Sparrow: Sparse rollout for stable and efficient long-context rl of large language models, 2026. URL https://arxiv.org/abs/2606.08446.

Adam Zweiger, Xinghong Fu, Han Guo, and Yoon Kim. Fast kv compaction via attention matching, 2026. URL https://arxiv.org/abs/2602.16284.

A Extended Related Work

Key-Value (KV) Cache The KV cache in transformers stores information from the past context, which eventually grows prohibitively expensive as generation continues. Thus, many context extensions focus specifically on handling the KV cache footprint. Methods either target the KV-cache from pre-fill, post-fill, or both (Bhaskar et al., 2025). Pre-fill methods seek to reduce the cost of the KV-cache from the prompt (Jiang et al., 2024; Eyuboglu et al., 2025), while post-fill methods deal with the KV cache after processing the prompt (Zhang et al., 2023; Liu et al., 2023c; Li et al., 2024b; Ge et al., 2024; Chen et al., 2024; Wang et al., 2025a; Cai et al., 2025; Hu et al., 2025; Cai et al., 2026; Metel et al., 2026; Ramachandran et al., 2026; Wang et al., 2026b; Chang et al., 2026; Zweiger et al., 2026; Li et al., 2026a). Many post-fill methods use recency eviction methods to discard older parts of the KV cache (Xiao et al., 2024c;b; Fu et al., 2025a; Han et al., 2026; Yu et al., 2026). Other approaches across pre-fill and post-fill optimize the KV cache and memory usage by taking hardware into consideration (Tang et al., 2024; Lu et al., 2025a; Akhauri et al., 2025; Yuan et al., 2025) or using quantization techniques (Lin et al., 2024; Xiao et al., 2024a; Frantar et al., 2023). Importantly, Prefix Sliding does not reduce the pre-fill cost of the KV cache, which may lead to high memory usage with extremely long prefixes. One solution could be to discard information from the prompt that is not needed for the prefix, or to introduce context management techniques (Zhang et al., 2026). Long inputs, such as a relevant book to solve the problem, are likely better suited in a file whose path is provided to the model so it can read it step by step.

Efficient thinking Several works have explored improving the thinking efficiency of reasoning language models. They seek to do so at train-time, test-time, or both. Train-time methods often target the RL algorithm or infrastructure-related issues (Liu et al., 2025; Luo et al., 2025; Dai et al., 2025; Zhao et al., 2025; Shen et al., 2026; Li et al., 2025a; Hou et al., 2025; Xiang et al., 2025; Zheng et al., 2025; Zhou et al., 2026; Wu et al., 2026), while test-time methods seek to work with existing models out-of-the-box that have already been trained (Damani et al., 2024; Li et al., 2025b; Wu et al., 2025a; Liang et al., 2026; Dong et al., 2026; Muennighoff et al., 2025). Prefix Sliding is both a train-time and test-time method: It can be used during RL and can be applied to already-trained models to achieve better efficiency and performance.

B Other sliding window sizes

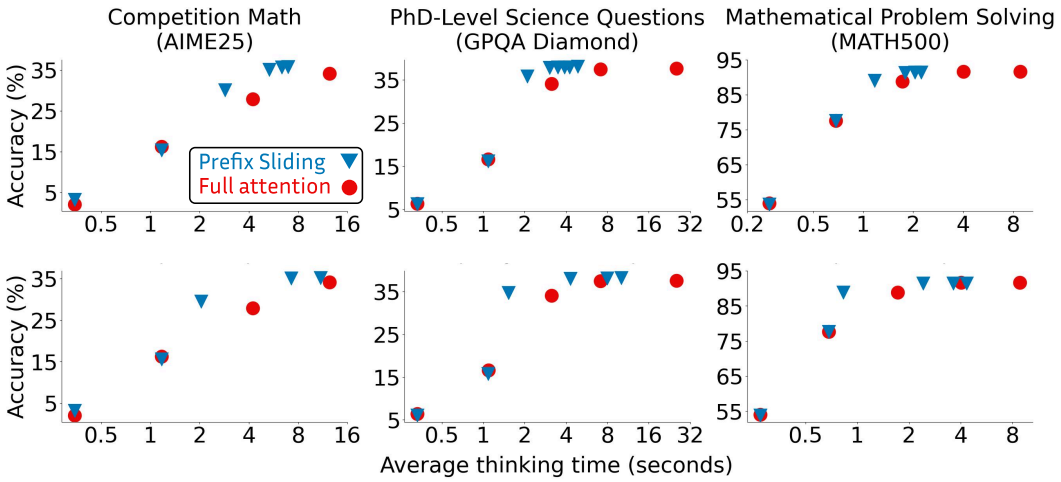


Figure 13: **Figure 1 with sliding window size 8192 (top) and 16384 (bottom).**

C Tabular results

Window size	AIME25 avg@64	AIME25 avglen	GPQA avg@64	GPQA avglen	MATH500 avg@64	MATH500 avglen	Tok/s at 32K	Tok/s at 128K
2048	27.7	47643	35.9	30107	89.8	9310	8973	8737
4096	33.9	29943	37.0	16707	91.5	7069	5479	5224
8192	35.8	19373	38.0	13605	91.4	6229	3291	2788
16384	35.3	19872	38.2	14378	91.5	6160	2441	1420
Full	34.2	19158	37.6	11403	91.7	6056	1477	448

Table 1: **Prefix Sliding achieves performance comparable to full attention at significantly higher speeds**. These performance numbers are used for figures throughout the paper.

D Continue vs Reset PE

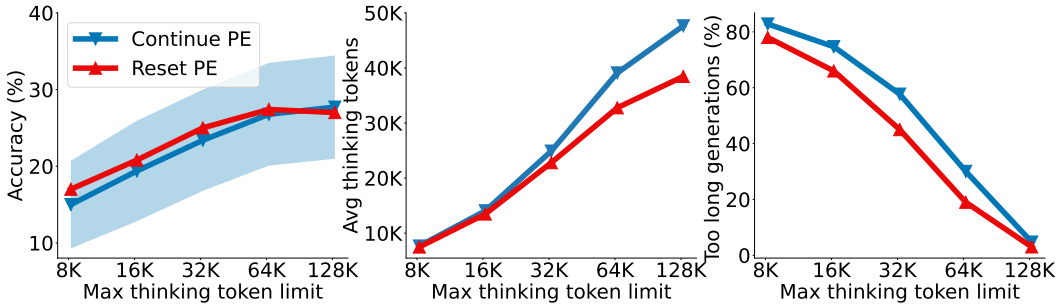


Figure 14: **Continue PE performs similarly to reset PE.** The benchmark is AIME25 and the shaded area represents the standard error. The sliding window size is 2048.

E Truncated backpropagation validation

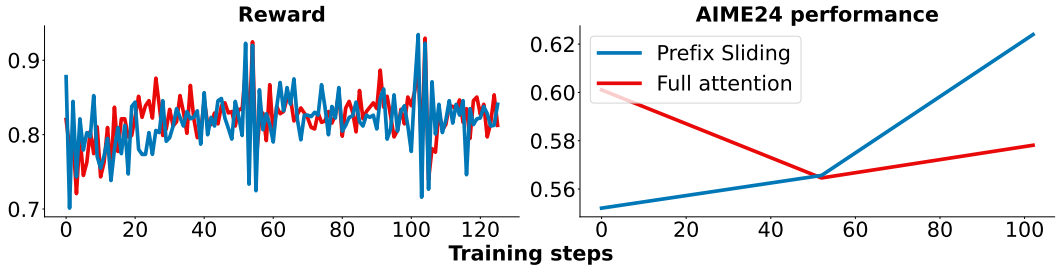


Figure 15: **Prefix Sliding with truncated backpropagation can match full attention.**

Figure 15 depicts a short experiment with DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI et al., 2025a) using the prime-rl codebase for asynchronous reinforcement learning (Intellect, 2025). For both models, 32768 tokens are passed to the trainer, but for Prefix Sliding only 8,192 of them are backpropagated using truncated backpropagation (section 2) with a multiplier of 4. The window size for Prefix Sliding is 8,192. We find that performance is comparable, but highlight that more experiments at even larger scales are necessary in the future. Therefore, we stick with truncated backpropagation for our experiments (e.g., Figure 7), but larger-scale runs with significantly longer chains may require chunked backpropagation (section 2).

F Training Dataset

For our dataset for reinforcement learning training we combine public sources, specifically SkyWork (He et al., 2025a) and s1 (Muennighoff et al., 2025) (s1 further sources from NuminaMATH (Li et al., 2024a), MATH (Hendrycks et al., 2021), OlympicArena (Huang et al., 2024), OmniMath (Gao et al., 2024), AGIEval (Zhong et al., 2023; Ling et al., 2017; Hendrycks et al., 2021; Liu et al., 2020; Zhong et al., 2019; Wang et al., 2021), OlympiadBench (He et al., 2024), TheoremQA (Chen et al., 2023), JEEBench (Arora et al., 2023), GPQA (Rein et al., 2023), SciEval (Sun et al., 2024)). We also decontaminate against test data using the s1 setup.

To filter problems, we rely on three criteria: guessability, verifiability, and difficulty. For guessability, we remove samples where small models write the correct solution on any of 8 tries without thinking (Kimi et al., 2025). For verifiability, we remove any samples that contain a set of words such as "How", "Explain", as such questions may have answers that cannot be easily objectively verified. For difficulty, we score models on all samples multiple times. We then remove those always solved by weak models, as well as those never solved by strong models, as they may be impossible to solve.

G Other methods

G.1 Last k hyperparameters

We select $k = 256$ for our runs in Figure 9 based on a sweep in Table 2. This means that whenever the model runs out of context, exactly 256 tokens from the end of the generation are taken and prepended to the thinking in the next generation of the model. We do not consider proper sentence endings; thus, the tokens are likely to be cut off; see Figure 21 for an example. To keep the total thinking tokens the same after the model has received the last k tokens of the first turn, it generates k fewer tokens from the second turn onward. The optimal value of last k likely depends on the context window and the number of allowed passes, so the setup in Figure 9 may have a different optimum. However, in practice one cannot predict the number of passes ahead of time, but generally needs to use the same last k value across setups.

k **tokens**	**MATH500**	**AIME25**
64	58.2	3.2
128	59.7	3.5
256	**60.8**	**4.2**
512	60.3	**4.2**
1024	54.6	2.4

Table 2: **Ablating last k.** Results are avg@64 with a 2048-token context and one pass. We select $k = 256$, which performs best while keeping the carried context small.

G.2 Summary hyperparameters

We set a maximum summary length of $k = 256$ based on subsection G.1. We treat the summary as a tool call and use summary forcing: If k tokens are left in the context window, and the summary tool has not yet been invoked, we force-insert the tool call into the reasoning of the model so it generates a summary. We use the model itself as the summary model. We ablate prompts in Table 3. Adding an example of how to use the tool and context (prompt 2) raises performance; possibly it leads to better summaries and their usage. Explaining the context further (prompt 3) did not help. Figure 20 shows an issue with the summary approach where the model seemingly ignores its summary, or maybe it uses it without mentioning it in its thinking (Lanham et al., 2023; Chen et al., 2025). Thus, we stick with prompt 2 for Figure 9. For summary, we do not subtract the summary length from the tokens the model may generate in its second turn onward; thus, it keeps slightly more tokens in memory than Prefix Sliding or last k. This may give the summary approach a slight advantage in Figure 9. We also tried inserting the summary in the model's thinking, which led to worse performance; possibly it was very out-of-distribution for the model.

Prompt	AIME25	
	Accuracy	Coverage
1: Tool only (Figure 16, Figure 17)	23.2	33.3
2: Tool/context examples (Figure 18, Figure 17)	26.4	53.3
3: Tool/context examples with info (Figure 18, Figure 19)	25.8	46.7

Table 3: **Summary prompt ablations.** The first figure in the brackets is the system prompt; the second is the user prompt after a summary is produced (the first turn is unmodified). cov@64 is coverage across 64 samples, i.e., if any of the 64 are correct, it is counted as correct.

```
# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "pass", "description": "Passes the
task to the next model to solve the problem from where you left off. Useful
when thinking gets too long.", "parameters": {"type": "object", "properties":
{"context": {"type": "string", "description": "Information to pass to the next
model. E.g., ideas already tried, key results, next steps to perform..."}},
"required": ["context"]}}}
</tools>
For each function call, return a json object with function name and arguments
within <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

Figure 16: **System prompt framing summary as a tool.**

```
<|im_start|>user
{problem}

Context:
{context}<|im_end|>
<|im_start|>assistant
```

Figure 17: **User prompt inserting context simply.**

{Figure 16}

Below is an example of using the pass tool:
<|im_start|>user
There exist real numbers x and y, both greater than 1, such that $\log_x(y^x) = \log_y(x^{4y}) = 10$. Find xy.<|im_end|>
<|im_start|>assistant
<think>
Okay, so I need to solve this problem where there are real numbers x and y, both greater than 1, such that log base x of (y^x) equals log base y of (x^4y) equals 10. And I need to find the product xy. Hmm, let me think.

First, let me recall some logarithm properties. The equation says that two different logarithms are equal to 10. So, $\log_x(y^x)$ = 10 and $\log_y(x^{4y})$ = 10. Maybe I can convert these logarithmic equations into exponential form to make them easier to handle.

Starting with the first one: $\log_x(y^x) = 10$. Using the logarithm power rule, which says that $\log_b(a^c) = c * \log_b(a)$. So, applying that here, $\log_x(y^x) = x * \log_x(y) = 10$. So, x * $\log_x(y)$ = 10.

But $\log_x(y)$ is the same as $1/\log_y(x)$, by the change of base formula. Wait, but maybe I can express $\log_x(y)$ in terms of something else. Let me
<tool_call>
{"name": "pass", "arguments": {"context": "Maybe convert logarithmic equations into exponential form to make them easier to handle. Via logarithm power rule $\log_x(y^x) = x * \log_x(y) = 10$. Noting that $\log_x(y)$ is the same as $1/\log_y(x)$. Maybe try express $\log_x(y)$ in terms of something else."}}
</tool_call>

When context is passed to you, evaluate its correctness first and then use it if correct.
<|im_start|>user
There exist real numbers x and y, both greater than 1, such that $\log_x(y^x) = \log_y(x^{4y}) = 10$. Find xy.<|im_end|>
<|im_start|>assistant
<context>
Maybe convert logarithmic equations into exponential form to make them easier to handle. Via logarithm power rule $\log_x(y^x) = x * \log_x(y) = 10$. Noting that $\log_x(y)$ is the same as $1/\log_y(x)$. Maybe try express $\log_x(y)$ in terms of something else.
</context>
<think>
Okay, so I need to first evaluate whether the context is correct and then use it to solve the problem.

The context suggests converting logarithmic equations into exponential form and using the logarithm power rule, which is a valid approach. It also notes that $\log_x(y)$ can be expressed in terms of $\log_y(x)$, which is correct. It suggests expressing $\log_x(y)$ in terms of something else.

Wait, do we really need to express $\log_x(y)$ in terms of something else? Maybe we can instead first use the logarithm power rule for the second equation as well. Let me do that...

Figure 18: **System prompt with tool-usage and context-usage examples.**

```
<|im_start|>user
{problem}

Context from your prior reasoning attempt that may be useful but could also be
wrong:
{context}<|im_end|>
<|im_start|>assistant
```

Figure 19: **User prompt inserting context with extra info.**

G.3 H_2O

H_2O (Zhang et al., 2023) retains recent tokens and "heavy hitters" that accumulate high attention scores, while evicting less influential tokens from the KV cache. Thus, prefix tokens that have not received sufficiently high attention can be discarded once they leave the recency window. H_2O is not integrated with FlashAttention/vLLM (https://github.com/vllm-project/vllm/issues/3532), making a fair efficiency comparison difficult.

However, we think combining H_2O with Prefix Sliding could be promising. Both retain recent tokens, but they differ in that Prefix Sliding is "forward-looking" while H_2O is "backward-looking". Prefix Sliding preserves prefix tokens, which we know may become important later. For example, a tool definition may get little attention for thousands of tokens but must be available when the model eventually uses that tool. In contrast, H_2O retains tokens based on their importance thus far. Preserving the prefix, heavy-hitting intermediate tokens, and a recency window may outperform Prefix Sliding alone. However, it may be challenging to make such a method work for reinforcement learning.

H Analyzing outputs

<|im_start|>user
An isosceles trapezoid has an inscribed circle tangent to each of its four sides. The radius of the circle is 3, and the area of the trapezoid is 72. Let the parallel sides of the trapezoid have lengths r and s, with $r \neq s$. Find $r^2 + s^2$
Context from your prior reasoning attempt that may be useful but could also be wrong:
I need to compute r² + s² where $r = 12 + 6\sqrt{3}$ and $s = 12 - 6\sqrt{3}$.<|im_end|>
<|im_start|>assistant
<think>
Okay, so I need to find r² + s² where r and s are the lengths of the two bases of an isosceles trapezoid that has an inscribed circle with radius 3 and an area of 72. Hmm, let me recall what I know about trapezoids with inscribed circles.
First, I remember that for a trapezoid to have an inscribed circle (be tangential), the sum of the lengths of the two bases must equal the sum of the lengths of the two legs. Since it's an isosceles trapezoid, the legs are equal in length. Let me note that down:
Let the two bases be r and s, with $r \neq s$. Then, the legs (which are equal) must satisfy:
r + s = 2 * leg_length.
But I also know that the area of a trapezoid is given by the formula:

Figure 20: **Summary duplicate processing problem on AIME sample.** The model ignores its prior derivations and starts the task anew.

<|im_start|>user
An isosceles trapezoid has an inscribed circle tangent to each of its four sides. The radius of the circle is 3, and the area of the trapezoid is 72. Let the parallel sides of the trapezoid have lengths r and s, with $r \neq s$. Find $r^2 + s^2$<|im_end|>
<|im_start|>assistant
<think>
$+s)/2 * h = 24/2 * 6 = 12 * 6 = 72$. Correct. The radius is 3, which is $h/2 = 6/2 = 3$. Correct.
Then, r² + s²: $(12 - 6\sqrt{3})^2 + (12 + 6\sqrt{3})^2$.
Let me compute $(a - b)^2 + (a + b)^2$ where $a = 12, b = 6\sqrt{3}$.
Which is (a² - 2ab + b²) + (a² + 2ab + b²) = 2a² + 2b² = 2(a² + b²).
So $2 * (12^2 + (6\sqrt{3})^2) = 2 * (144 + 36 * 3) = 2 * (144 + 108) = 2 * (252) = 504$.
Wait, but the answer is 504? But let me check again.
Wait, the problem says the trapezoid has an inscribed circle. For a trapezoid to have an inscribed circle (be tangential), the sum of the lengths of the two bases must equal the sum of the lengths of the two legs. So, in this case, since it's an isosceles trapezoid, the legs are equal. So, let me check if that's the case.
Let me recall that for a trapezoid to be tangential, the sum of the lengths of the two bases equals the sum of the lengths of the two legs. So, if the bases are r and s, and the legs are both of length l, then r + s = 2l.
Also, the area is given by the average of the bases times the height. So, (r + s)/2 * h = 72. And the radius of the inscribed circle is h/2 = 3, so h = 6. Therefore, (r + s)/2 * 6 = 72 => (r + s)/2 = 12 => r + s = 24.
So, we have that r + s = 24 and that r + s = 2l => l = 12. So, the legs are 12 each.
Now, the trapezoid is isosceles, so the legs are equal, and the height is 6.
Now, we need to find r² + s². But how can we find r and s? We know that r + s = 24, and we need another equation. Let's think about the geometry.
In an isosceles trapezoid with bases r and s, and legs of length l = 12, and height h = 6.
The difference between the bases is r - s, and when you drop the perpendiculars from the ends of the shorter base to the longer base, you form two right triangles on either side. The base of each triangle is (r - s)/2, and the height is 6, and the hypotenuse is l = 12.
So, (r - s)/2 = sqrt(l² - h²) = sqrt(144 - 36) = sqrt(108) = 6*sqrt(3).
Therefore, $(r - s)/2 = 6\sqrt{3} \Rightarrow r - s = 12\sqrt{3}$.
So now we have:
r + s = 24
$r - s = 12\sqrt{3}$
Adding these equations: $2r = 24 + 12\sqrt{3} \Rightarrow r = 12 + 6\sqrt{3}$
Subtracting the second equation from the first:
$2s = 24 - 12\sqrt{3} \Rightarrow s = 12 - 6\sqrt{3}$
Therefore, r² + s² = $(12 + 6\sqrt{3})^2 + (12 - 6\sqrt{3})^2$
Let me compute each term:
First term: $(12 + 6\sqrt{3})^2 = 12^2 + 2 * 12 * 6\sqrt{3} + (6\sqrt{3})^2 = 144 + 144\sqrt{3} + 36 * 3 = 144 + 144\sqrt{3} + 108 = 252 + 144\sqrt{3}$
Second term: $(12 - 6\sqrt{3})^2 = 12^2 - 2 * 12 * 6\sqrt{3} + (6\sqrt{3})^2 = 144 - 144\sqrt{3} + 108 = 252 - 144\sqrt{3}$
Adding them together: $(252 + 144\sqrt{3}) + (252 - 144\sqrt{3}) = 252 + 252 = 504$
So the answer is 504. Therefore, the value of r² + s² is 504.

Figure 21: **Last k duplicate processing issue on AIME sample.** We mark the repetition and where the carried-over 256 tokens end.

```
...
Let me think of the following code:
def can_remove_k(targetIndices, source, pattern, k):
    removed = []
    for i in range(len(targetIndices)):
        if len(removed) >= k:
            break
        # Check if removing this index allows the pattern to be a subsequence
        # Simulate the pattern as a subsequence, skipping this index
        # So, we need to check if the pattern is a subsequence after removing this
index
        # To do this, we can simulate the pattern as a subsequence, but skipping
this index
        # Let's create a list of indices in source that are not removed
        # But since the targetIndices are in order, and we are checking for each
index in order, we can simulate this
        # So, we can simulate the pattern as a subsequence, with the current index
being removed
          # So, we can create a list of the source's characters, and for each
character, if it's not the current index, we check
        # But this is O(n) per check
        # So, the code would be:
        # simulate the pattern as a subsequence, with the current index removed
        # if the pattern is a subsequence, then we can remove it
        # So, we need to check if the pattern is a subsequence of source, with the
current index removed
        # Let's implement this
        # We can create a list of the source's characters, and for each character,
if it's not the current index, we check
        # But how to do this efficiently
        # Let's create a copy of the source, and remove the current index
        # But this is O(n) for each check
        # However, for large n, this is not feasible
        # So, we need a more efficient way to check this
        # Here's a way to do it:
        # We can simulate the pattern as a subsequence, but with the current index
removed
        # So, we can have a pointer for the pattern, and iterate through the source,
skipping the current index
        # So, the code would be:
        # current pattern pointer
        p = 0
        for idx in range(len(source)):
            if idx == targetIndices[i]:
                continue
            if p < len(pattern) and source[idx] == pattern[p]:
                p += 1
        if p == len(pattern):
            removed.append(targetIndices[i])
    return len(removed) >= k
Wait, but this code is not correct. Because for each check, we are trying to remove
the current index, but in the code above, we are checking whether the pattern is
a subsequence after removing the current index.
...
```

Figure 22: **Reasoning in comments issue on LiveCodeBench sample.**