IS A LARGE CONTEXT WINDOW ALL YOU NEED? EXPLORING TIME TO FIRST TOKEN (TTFT)-CONTEXT SIZE TRADEOFF FOR AUTOREGRESSIVE LLMS

Anuran Roy
Alchemyst AI
Bengaluru
anuran@getalchemystai.com

Arnab Sengupta
Alchemyst AI
Bengaluru
arnab@getalchemystai.com

Saptarshi Pani
Alchemyst AI
Bengaluru
saptarshi@getalchemystai.com

ABSTRACT

Recent advancements in auto-regressive large language models (henceforth referred to as LLMs) have significantly expanded context window capacities, with Meta's Llama 4 Scout achieving a 10 million token input length . This expansion is facilitated by techniques like Rotary Position Embedding (RoPE)[1] and YaRN (Yet Another Rope extensioN)[2], which encodes positional information through rotational transformations, enabling models to process longer sequences effectively.

This advancement opens up a host of opportunities for the ubiquitious LLMs. Yet, attention mechanisms barely sub-quadratic in their nature[3] [4] [5]. This means that extending context windows introduces challenges in latencies, especially in scenarios where even sub-second delays can result in catastrophic failures at scale in real-life use cases, many of which can be silent.

This paper examines the trade-offs between context sizes and latencies, highlighting the need for improved context retrieval strategies that do not bloat query sizes to the concerned Large Language Models.

***K*eywords** Information Retrieval · Context Window · Large Language Models · Time To First Token

1 Introduction

In the realm of large language models (LLMs), performance metrics like Time to First Token (TTFT) and Inter-Token Latency (ITL) are pivotal in determining user experience and system efficiency. TTFT measures the duration from when a prompt is submitted to when the model generates its first token, directly impacting perceived responsiveness in applications such as chatbots and real-time assistants. Conversely, ITL assesses the average time between generating consecutive tokens, influencing the fluidity of the model's output. Some business-critical cases of TTFT are listed as below:

1. **Customer Service Calls and Live Agent Assist**: In customer support scenarios, especially voice-based interactions, long TTFTs can lead to *unnatural pauses*, frustrating both agents and customers. For instance, if a call center agent relies on an LLM to surface relevant information or suggest responses during a live call, even a one-second delay can disrupt the flow of conversation and lower *customer satisfaction scores (CSAT)*.
2. **Real-Time Translation and Transcription**: In real-time translation apps or multilingual meeting tools, high TTFT makes the system feel sluggish, impairing usability. A swift TTFT ensures that translation appears in near real-time, which is critical for smooth communication across languages in high-stakes settings like international sales or legal negotiations.
3. **Interactive Virtual Assistants**: Virtual assistants used in healthcare, finance, or IT support must respond quickly to maintain a feeling of *natural dialogue*. If the TTFT is too high, users may perceive the assistant as slow or unhelpful, reducing adoption and trust in the technology.

4. **High-Volume Query Systems**: In environments like e-commerce search engines or recommendation systems, slow TTFT can lead to drop-offs and revenue loss. Every millisecond counts when serving millions of users, and delayed responses can significantly degrade *conversion rates*.

As seen from the above business applications, optimizing TTFT is crucial for business applications where prompt responses are essential. Two primary strategies to mitigate TTFT include:

1. **Accelerated KV-Cache Pre-Filling**: By precomputing and storing key-value pairs during the initial stages of inference, models can bypass redundant computations in subsequent steps, thereby reducing latency.
2. **Enhanced Context/Data Retrieval**: Implementing efficient data fetching mechanisms ensures that relevant information is promptly available to the model, minimizing delays caused by data access bottlenecks.

Addressing these areas is vital for deploying LLMs in environments where speed and efficiency are paramount. But a conundrum arises when you have a time-sensitive use case based on a large organizationational knowledge base. A prime example for that can be customer outreach campaigns at scale via call - which also happens to be one of the most concentrated applications of Generative AI.

Another point to consider is the fact that although a major chunk of applications use the Big Three Generative AI Providers, on-premise deployments of Generative AI are extremely crucial, which require either: *(a)* adequate Graphics Processing Units (abbreviated GPUs) that are at the top of the capability scale (think NVidia H100 or B200s), but increasing scarce and expensive, or *(b)* intelligently managing context windows and prompt sizes (in tokens).

The latter of the aforementioned methods is a more economical approach and leads to results faster with less blackboxing. In this paper, we present a study showing how efficient context pruning can speed up inference speeds, while also ensuring manageable LLM application costs.

This paper addresses the

2 Literature Review

The Transformer architecture, introduced by Vaswani et al. [6], established self-attention as the core computational primitive in modern large language models (LLMs). However, despite numerous innovations, attention remains fundamentally constrained by $\mathcal{O}(n^2)$ complexity with respect to sequence length n, in both computational and memory terms. This quadratic scaling is not only a theoretical limitation, but a practical one—especially for real-time or large-context applications—where throughput and latency are sensitive to input length. While several methods aim to reduce these costs through approximation or sparsity, dense attention remains the de facto standard in many high-performing models due to its accuracy and generality. Consequently, attention remains an approximate $\mathcal{O}(n^2)$ bottleneck, and efficient deployment increasingly hinges on managing *context size*.

2.1 Sparse Attention and Approximation Methods

Sparse attention variants reduce the number of token interactions per layer. Notable examples include Longformer [7] and BigBird [8], which use fixed or randomized sparsity patterns to achieve sub-quadratic complexity. These patterns, often in the form of sliding windows or block-global connections, reduce attention operations to $\mathcal{O}(n \cdot w)$ for a window size $w \ll n$. Despite theoretical efficiency, GPU inefficiencies in handling irregular access patterns limit the speedups in practice. Tay et al. [9] found that such methods only marginally outperform dense attention at moderate lengths ($n \leq 4$K), shifting the burden back to hardware-aware optimizations.

2.2 FlashAttention: Memory-Efficient Exact Attention

FlashAttention [3] re-engineers the exact attention computation for modern GPU hardware. While it retains $\mathcal{O}(n^2)$ time complexity, it reduces memory complexity to $\mathcal{O}(n)$ by computing attention in tiled chunks that exploit fast on-chip memory and avoid storing the full attention matrix. Empirical benchmarks report up to $3\times$ speedups and $10\times$ reductions in memory usage on long sequences. Importantly, FlashAttention achieves this without approximation, making it a practical default in many modern LLMs. However, it does not eliminate the fundamental quadratic scaling; instead, it pushes the quadratic cost further out, allowing longer sequences within the same memory footprint.

2.3 Linear and Kernel-Based Attention

Performer [10] and Linformer [11] aim to reduce attention complexity to $\mathcal{O}(n)$ or $\mathcal{O}(n \cdot k)$ via approximation. Performer uses kernel approximations of softmax attention, while Linformer projects key and value matrices into a low-rank

subspace. These methods demonstrate strong theoretical properties and linear memory growth, yet their adoption in mainstream LLMs remains limited. This is partly due to performance variability and degradation in downstream tasks, and partly because the approximations may not generalize well to novel or extremely long contexts.

2.4 Hardware and Context-Size Bottlenecks

Despite efforts to reduce attention complexity, most state-of-the-art LLMs—including GPT-4, Claude, and Mistral—rely on modified but fundamentally quadratic attention mechanisms. For instance, Mistral employs sliding window attention in conjunction with FlashAttention to accelerate inference [12], but still uses dense local computation. These methods delay the scaling barrier rather than remove it.

As a result, the primary bottleneck in LLM efficiency has shifted to *context size*. Attention's approximate $\mathcal{O}(n^2)$ cost means that doubling the sequence length can still quadruple memory or compute needs, even under optimized implementations. In latency-sensitive use cases—such as real-time inference or document-level understanding—efficient management of context becomes critical. Techniques like multi-query attention [13] or position interpolation help mitigate this pressure by reducing overhead or improving context extrapolation, but the fundamental challenge persists.

3 Methodology

3.1 Intuition

In an organizational context, context involved prompting is atleast a few orders of magnitude larger than in day-to-day use cases, while self-hosted or on-premise GPUs can handle only so much. Attention Mechanism is approximately a quadratic operation with respect to the computational and memory complexity of the size of query received (by tokens).

We hypothesize here for our experiment that reducing the context window dynamically before sending the query to the LLM (hereby referred to as the *context-pruning* process) would impact the inference latency from the LLM.

Some simple math to back up the hypothesis is as follows:

We assume Attention to be quadratic here, for the convenience of numbers (since beyond 4k tokens, it almost becomes the same[9].)

Now let us consider 3 paragraphs C_1, C_2, C_3 with token consumption $n(C_1) = n(C_2) = n(C_3) = 10,000$

Now, let us consider two pipelines:

- **Naive append**: Directly dump the context into the LLM, appending them one after another in some form - popular techniques include appending with newline characters, etc.
- **Context Compression**: Compress each context and then recombine them together recursively.

3.1.1 Naive Append

In this case, we create a single large query $Q_1 = C_1 + C_2 + C_3$ such that $n(Q_1) = n(C_1) + n(C_2) + n(C_3) = n(C_1 + C_2 + C_3)$

Thus, Time complexity involved for computing results from the LLM is $Q_1 \propto n(Q_1)^2 = 30000^2 = 9 \times 10^8 = \zeta_{\mathbf{1}}$ (say)

3.1.2 Context Compression

We define a process:

$$f_m : \mathbb{R} \to \mathbb{R}/m\mathbb{Z}$$

such that:

$$n(f_m(C_1)) \propto \frac{n(C_1)}{m}$$

, which basically means that the context is summarized to approximately $\frac{1}{m}$th of the original size.

Now, via one pass of the context compression procedure, assuming $m = 10$, we have:

$$n(f_m(C_1) + f_m(C_2) + f_m(C_3)) = 1000 + 1000 + 1000 = 3000$$

Now this is sent to the LLM.

If f_{10} involves an LLM-based compression procedure, the total complexity looks like:

$$[n(C_1)]^2 + [n(C_2)]^2 + [n(C_3)]^2 + [n(f_{sum}(C_1))]^2 + [n(f_{sum}(C_2))]^2 + [n(f_{sum}(C_3))]^2$$
$$= 10000^2 + 10000^2 + 10000^2 + 3000^2 = 3 * 10^8 + 9 * 10^6 = \zeta_{\mathbf{2}}(\text{say})$$

3.2 Setup

3.2.1 Infrastructure

- A vLLM Kubernetes cluster with 2 L4 GPUs per node, running a 7B long context LLM (example, Qwen-2.5-7B-Instruct-1M)
- A proxy that contains an LLM-based compression, compatible with OpenAI API format, calling the LLM in the vLLM cluster
- A benchmarking script in Python
- A long context, multi-turn conversational dataset based on combining Paul Graham's essays.

3.2.2 Data

We required emulating context sizes comparable to organizational context, while pushing for common edge cases like "needle-in-a-haystack" and uncorrelated context, which may either induce added latency and/or worse, induce context poisoning, whereby the Generative AI-powered agent might be led astray from the intended response - which is something we also need to measure. For this, we needed to serve up mixed contents that are partly correlated, while the remaining parts are uncorrelated. To make this test even more realistic, we needed to introduce edge cases where sometimes uncorrelated context seep across multi-turn conversations.

So we generated the test data with the following steps:

1. **Get long context chunks of data**: We took the dataset on Paul Graham's Essays[14] (thanks to the team behind RULER[15] for making our life easier!)
2. **Get multi-turn conversations**: We used HotPotQA[16], SQuAD[17], and MuTual[18] datasets.
3. **Generate multi-turn contextual conversations**: We generated multi-turn contextual conversations by chunking the dataset of Paul Graham's essays, recombining them with random samples of coherent and incoherent chunks, and then transforming into part-contextual, part-random turns in different conversations, where the questions are hand crafted, by carefully considering *NIAH* and context poisoning.
4. **Combine generated conversation segments with other turn-based datasets**: Now we combine segments of these generated conversations with the other turn-based conversations, and sample them into a partly-contextual, partly poisoned conversations, that may or may not have multi-turns.

This is a pretty realistic guesstimate of user-AI conversations across multiple sessions, involving mixed contexts. We then filter out the longest 50% of the conversations, and then sort them by length.

3.3 Execution

We run the LLM calls for each sample, one through the proxy, and one directly to the vLLM cluster, without the proxy. We measure response times, as well as account for errors where LLM timeouts happened.

We pass the generated conversations as messages, and set LLM caching parameter to true for the OpenAI SDK client calls, to mimic Generative AI deployments in production. We record the times of the LLM call for each conversation, first without proxy, and then with proxy. The context-pruning proxy (hereforth referred as "proxy" in some cases) here is simply a relay that has an OpenAI compatible chat completions API format, and does context pruning before sending it to the real vLLM cluster.

4 Results

Mathematically, it sounds beneficial. Let us denote superposition as:

$$f_m^n = f_m(f_m(f_m(...\text{n times})))$$

.

So now, for the time complexity we can state:

$$n(f_m^1(C)) = \Sigma_{i=1}^{m} \left(\frac{n(C)}{m}\right)^2 + \Xi(C)$$

where $\Xi(C)$ is an overhead involving other costs (network bandwidth, cache movement in GPUs, etc).

Thus,

$$\lim_{x \hookrightarrow \infty} \frac{f_m^1(x)}{x^2} = \lim_{x \hookrightarrow \infty} \frac{\Sigma_{i=1}^{m} \left(\frac{n(x)}{m}\right)^2 + \Xi(C)}{x^2} \approx \frac{1}{m}$$

With one pass itself, we can get an inverse relation itself, which can be seen when $n(C) = 30000$, as in our aforementioned example.

Thus, we can actually establish a relation:

$$f_m^n(x) \propto \left(\frac{1}{m}\right)^n$$

Practically, speeds exceeded our expectations, as revealed in the following figures:

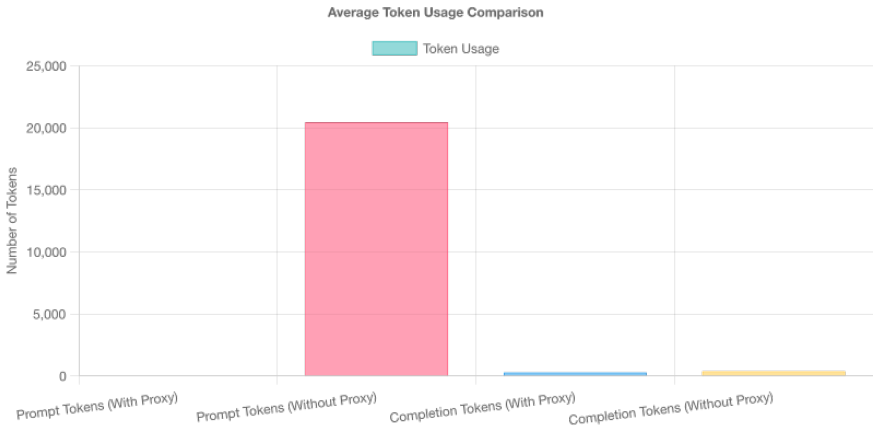

Figure 1: Tokens usage comparison over large contexts

Figure 2: Response times comparison - Naive Append without proxy vs Context Pruning with proxy

Our experimental evaluation comprised 20 test cases comparing LLM performance with and without context pruning through a proxy mechanism. The results demonstrate significant improvements in both response time and token efficiency. With the proxy-enabled setup, we observed an average response time of 7.61 seconds compared to 12.39 seconds without the proxy, representing a 38.5% reduction in latency.

The response time measurements showed considerable variation, with proxy-enabled queries ranging from 4.64 to 10.86 seconds, while non-proxy queries spanned 5.69 to 22.05 seconds. The maximum improvement in response time reached 12.00 seconds, though we noted occasional cases with minimal performance overhead (maximum of 329ms). This suggests that the proxy mechanism provides more consistent and predictable response times, effectively mitigating the worst-case scenarios observed in direct queries.

Most notably, the system achieved remarkable token efficiency with an average compression ratio of 0.997, corresponding to a 99.73% reduction in prompt tokens. The token savings ranged from 12,745 to 50,096 tokens per query, with a total of 407,941 tokens saved across all test cases. The compression performance was consistently high, with even the minimum reduction percentage reaching 99.59%. Interestingly, the proxy-enabled responses maintained comparable quality while being more concise, with average output lengths of 1,423 characters compared to 1,880 characters in non-proxy responses.

Figure 1 illustrates how stark the savings in token costs are, considering network bandwidth besides obvious token costs - which is understandable because of the context pruning.

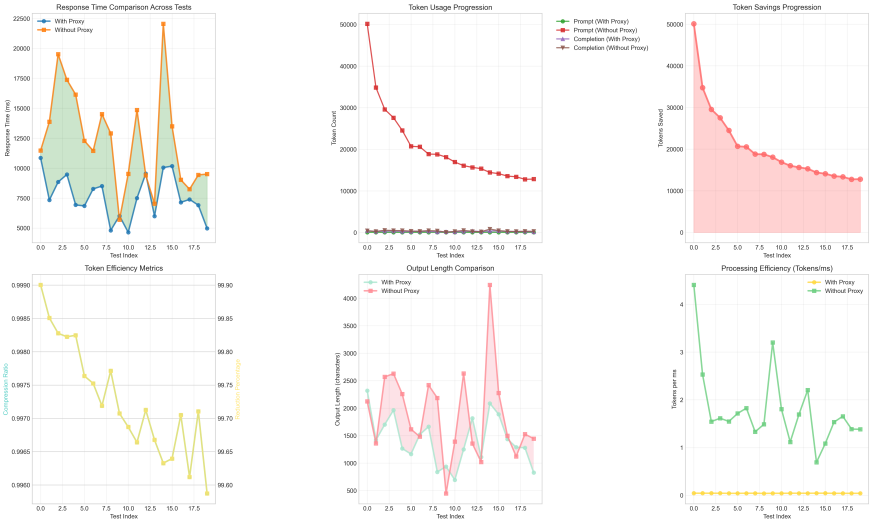

Figure 3: Trends on token consumption and latencies - Naive Append without proxy vs Context Pruning with proxy

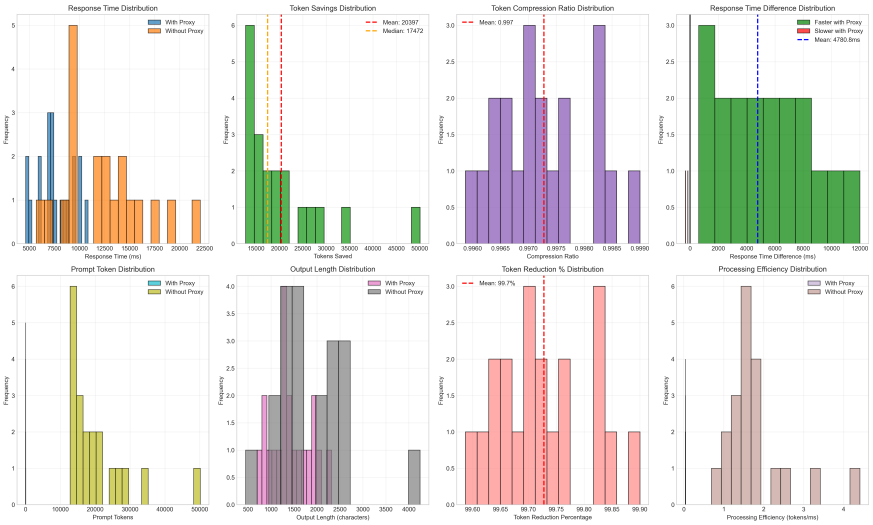

Figure 4: Comprehensive comparisons - Proxy vs non-proxy LLM calls

But counterintuitively, we can see a different pattern arise from figures 2, 3, 4, 5 - that beyond a certain threshold of tokens, LLM latencies actually decrease. Caching might mitigate the latencies, but for one miss of the cache (which is an extremely frequent occurence for an LLM catering to multiple use cases), the latency shoots up.

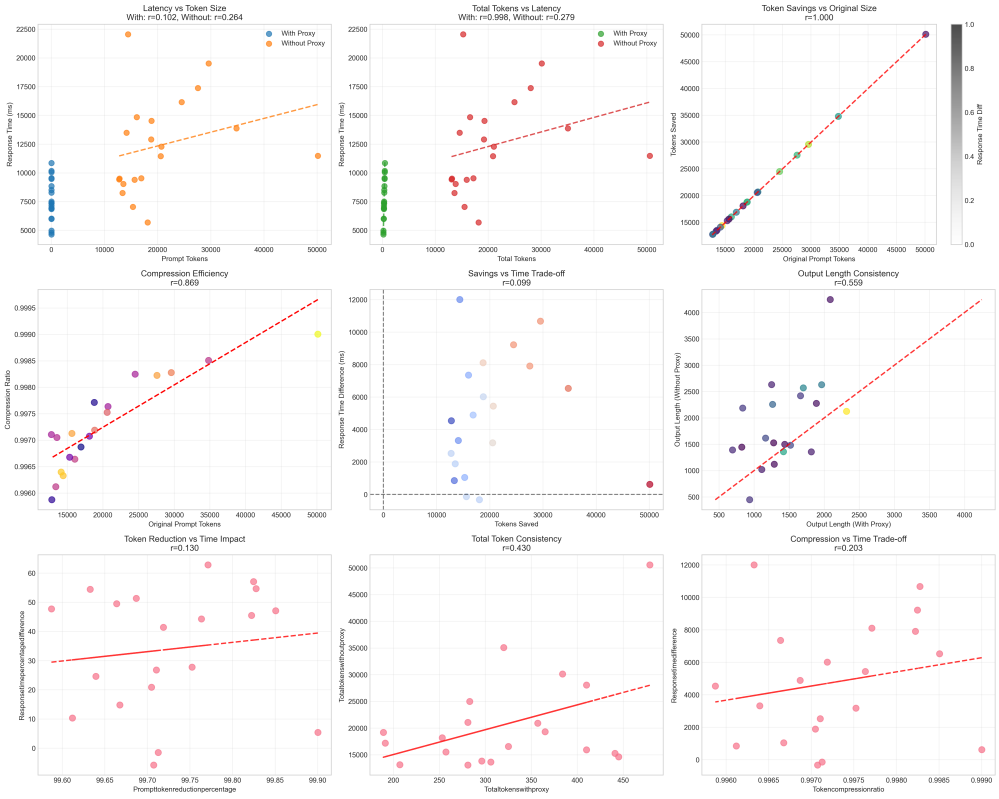

Figure 5: Correlation Analyses between tokens, latency and context sizes

Figure 6: Relations between tokens saved and latency

The ramifications of these in business contexts can be drastic - think of reply latencies shooting up so high when the customer is in the midst of a voice call with the AI Agent - that the customer mistakenly assumes that the phone has hung up, and hangs up from his side. Although the LLM is pre-occupied in processing the request, and generating the response, this response now has no business value, but is obstructing other requests waiting in queue to be processed. Even worse, this frequently leads to a domino effect, which can then spiral out of control into drastic CSAT score drops *en masse*, potentially underdelivering, or worse, actively harming business prospects from a Return of Investment perspective.

5 Conclusion

This study by our team ponders on the challenges of often hardware-constrained inferences in production while boosting inference speeds as well as saving tokens. With ever-evolving GPUs and ever-increasing token windows, the problem in time-sensitive use cases gets even more pronounced.

This study also shows that scaling AI and AI context is not causal, albeit with a positive correlation. This can open up further studies on how context is processed spanning an organization's knowledge base, which may indicate that we need to rethink on how data is processed differently for human and AI consumption. Context window size and hardware accelerators are something that are ever evolving and will be required, but thoughtless context dumping would have visible impact on performance in time-sensitive scenarios which would not be mitigated anytime soon.

Acknowledgments

All of our authors have contributed equally to this work. This work is also supported in part by the Alchemyst Platform Engineering Team, without whose support the stable platform and infrastructure for testing would have been very difficult to set up. Special thanks to the research circles who validated our work with their own observations as well.

References

[1] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[2] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2023.

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

[4] James Vo. Sparseaccelerate: Efficient long-context inference for mid-range gpus, 2024.

[5] Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Hannaneh Hajishirzi, and Daniel Khashabi. When not to trust language models: Investigating effectiveness and limitations of parametric and non-parametric memories. *arXiv preprint*, 2022.

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

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

[8] 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, 2020.

[9] 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, 2020.

[10] 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, 2020.

[11] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

[12] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[13] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[14] Samarth Goel. paul _graham _essays (revision 0c7155a), 2024.

[15] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

[16] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[17] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[18] Leyang Cui, Yu Wu, Shujie Liu, Yue Zhang, and Ming Zhou. Mutual: A dataset for multi-turn dialogue reasoning. In *Proceedings of the 58th Conference of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.