Metron: Holistic Performance Evaluation Framework for LLM Inference Systems

Amey Agrawal[1*] **Anmol Agarwal**[1*] **Nitin Kedia**[2] **Jayashree Mohan**[2] **Souvik Kundu**[3]
Nipun Kwatra[2] **Ramachandran Ramjee**[2] **Alexey Tumanov**[1]

[1]Georgia Institute of Technology [2]Microsoft Research India [3]Intel AI Lab

Abstract

Serving large language models (LLMs) in production can incur substantial costs, which has prompted recent advances in inference system optimizations. Today, these systems are evaluated against conventional latency and throughput metrics (eg. TTFT, TBT, Normalised Latency and TPOT). However, these metrics fail to fully capture the nuances of LLM inference, leading to an incomplete assessment of user-facing performance crucial for real-time applications such as chat and translation. In this paper, we first identify the pitfalls of current performance metrics in evaluating LLM inference systems. We then propose Metron, a comprehensive performance evaluation framework that includes *fluidity-index*– a novel metric designed to reflect the intricacies of the LLM inference process and its impact on real-time user experience. Finally, we evaluate various existing open-source platforms and model-as-a-service offerings using Metron, discussing their strengths and weaknesses. Metron is available at github.com/project-metron/metron.

1 Introduction

The surge in popularity of LLMs has resulted in the proliferation of both proprietary model-as-a-service offerings [10, 2, 4, 5, 1] and active open-source developments aimed at optimizing LLM inference [14, 12, 6, 16]. Given the vast array of available options, a systematic comparison of these frameworks becomes critical to ensure good user experience and cost-effective deployment.

Current evaluation metrics for LLM serving frameworks, such as TTFT (Time To First Token), TBT (Time Between Tokens), normalized latency, and TPOT (Time Per Output Token), fail to capture the full essence of the user experience in real-time LLM interactions. This paper demonstrates that these conventional performance metrics, while valuable, are inadequate and potentially misleading when applied to the dynamic, streaming nature of LLM inference. We argue for a more nuanced approach that considers the temporal aspects of token generation and their impact on perceived responsiveness and overall user satisfaction.

Fine-grained metrics like TTFT and TBT effectively capture latency for individual tokens and tail latency characteristics, however, they fail to represent the overall end-to-end token generation throughput. Conversely, normalized metrics such as TPOT and normalized latency attempt to measure token throughput, but fall short in identifying specific sources of user experience degradation, such as inter-token jitter or scheduling delays – which are similar to buffering time in conventional media streaming settings. This dichotomy highlights the need for a more comprehensive evaluation framework that concisely captures the overall user experience.

To address the limitations of existing metrics, we introduce Metron, a comprehensive framework for evaluating user-facing performance in LLM inference. At its core are two novel metrics: *fluidity-*

*Equal contribution. Correspondence at ameyagrawal@gatech.edu.

Fluidity index: Account for variability in token gen rate, by setting deadlines & measuring fraction of deadlines met.

index and *fluid token generation rate*, designed to capture the nuances of real-time, streaming LLM interactions. We apply this framework to conduct an extensive performance evaluation of both open-source and proprietary LLM inference systems, revealing their strengths and weaknesses.

The design of the *fluidity-index* metric is inspired by the deadline-based evaluation of periodic tasks in real-time systems [20]. We draw an analogy between periodic tasks in the rich literature of real-time systems and the streaming token generation in LLM inference. Ideally, LLM output should maintain a smooth, consistent rate akin to media streaming platforms. However, due to various system challenges, it is difficult to maintain a constant token generation rate. *fluidity-index* accounts this variability by setting token-level deadlines and measuring the fraction of deadlines met within a request. A deadline miss corresponds to a token generation stall due to delayed token generation. On the flip side, in cases of token generation exceeding the required playback rate, there's an opportunity to buffer released tokens to help mitigate future stalls and increase the fluidity. This approach enables precise and quantitative definitions of user experience constraints. For instance, it becomes possible to report the maximum load the system can sustain (capacity), subject to SLOs defined on *fluidity-index*, e.g., 99% of requests achieving a metric of ≥ 0.9.

Devops metrics for LLM inference

fluid token generation rate complements *fluidity-index* by determining the maximum sustainable playback rate that maintains a specified level of fluidity (e.g., *fluidity-index* > 0.9). That way *fluid token generation rate* enables black-box evaluation of LLM inference systems. Combined, these metrics provide a holistic view of LLM inference performance that more closely aligns with real-world user experience.

Fluid token gen rate: Max playback rate that maintains certain token generation rate

We have open-sourced Metron at github.com/project-metron/metron, aiming to establish a standard for user-centric performance evaluation in the rapidly evolving landscape of LLM inference systems and frameworks.

2 Background

In this section, we describe the typical LLM inference process, commonly used metrics to characterize inference performance, and an overview of open-source and proprietary inference solutions.

2.1 LLM Inference Process

There are two distinct phases in LLM inference – a prefill phase followed by a decode phase. During prefill phase, the user's input prompt is processed and first output token is produced. Next, during the decode phase, output tokens are generated one at a time, where the token generated in one step is passed through the model to generate a new token in the next step until a special *end-of-sequence* token is generated. Decode phase also requires access to KV (key and value) pairs associated with all previously processed tokens during its attention phase. Contemporary LLM inference systems store activations in KV-cache to avoid repeated re-computation during each step [30, 3, 25].

2.2 Performance Metrics for LLM Inference

Conventional performance metrics for LLM inference performance are the following:

- **TTFT** : Time To First Token (TTFT) [32, 7] is the latency between the request arrival and the first output token generated by the system for the request. It includes the scheduling delay (time elapsed from request arrival to start of prompt processing) and the prompt processing time. Minimizing TTFT is crucial for real-time interactions to maintain a responsive user experience. In contrast, longer TTFT is acceptable in offline or batch processing contexts.
- **TBT** : Time Between Tokens (TBT) [17] is the latency of every subsequent token generation in the decode phase. This metric directly influences the perceived speed of the model by users. If we assume the average English reading speed is 250 words per minute then a TBT of roughly 6 tokens per second is required. Optimizing TBT enhances the user experience by ensuring rapid and fluid response generation.

Min TBT → 6 tok/s

- **TPOT** : Time Per Output Token (TPOT) [32, 7] is closely related to TBT. It is the average time to generate an output token in the decode phase. It is calculated as the total decode time of a request normalized by the number of decode tokens generated.

- **Normalized Latency** : This is defined as the total execution time of a request normalized by the number of decode tokens. It includes the scheduling delay, prompt processing time and time to generate all the decode tokens. Median Normalised Latency has been used in [30, 25] to compare system throughput. Lower normalised latency at a given load (queries-per-second) is desirable.
- **Capacity** : This is defined as the maximum request load (queries-per-second) a system can sustain while meeting certain latency targets (SLOs). It has been used in [15, 17]. Higher capacity is desirable because it reduces the cost of serving.

2.3 LLM Inference Framework Evaluation

We now discuss what it means to evaluate the user-facing performance for LLM serving frameworks, in open-source [14, 17, 6, 13] as well as public model offerings [10, 4, 5, 2].

Open-source frameworks. Evaluating the performance of open-source frameworks like vLLM [14], Sarathi-Serve [17], LightLLM [6], Text-Generation-Inference [13], etc. is challenging due to their numerous configurable parameters. At the same time, accurate performance assessment is crucial during deployment to determine the maximum sustainable load for a given cluster while meeting specific latency targets (SLOs).

Proprietary model service offerings. Companies like OpenAI [10], Azure AI Studio [2], Fireworks AI [4], and Groq [5] provide model-as-a-service solutions that typically restrict end-user configurability regarding system performance. Consequently, users and developers are limited to passive performance evaluations based on their specific workload. In this constrained environment, performance comparisons across different services rely primarily on observable metrics such as latency and cost. This highlights the need for methods that can effectively guide users in selecting the most efficient and cost-effective service for their specific applications.

3 Motivation

3.1 Pitfalls of Existing Metrics

While the conventional latency and througput metrics described in §2.2 appear adequate in evaluating the performance of LLM inference systems, they fail to provide a comprehensive view of the user experience. Below, we discuss specific shortcomings identified in the current metrics.

Time To First Token (TTFT) is oblivious of prompt length. TTFT, which measures prefill efficiency, includes both the scheduling delay (which depends on the system load, routing policy, batching policy [30, 25, 17], etc.) and the actual prompt processing time which depends on the prompt length. Naively comparing two systems on their TTFT does not reveal the individual contribution of these components to the prefill time. Moreover, since TTFT is highly dependent on the prompt length (quadratic), as shown in Figure 1, defining a static Service Level Objective (SLO) on TTFT as a measure for user-facing responsiveness of the system is not practical. A naive alternative would be to normalize TTFT by the prompt length; but this normalizes the scheduling delay as well and would penalize shorter input requests disproportionately compared to longer ones.

TFT does not measure prefix time accurately

Normalized latency hides scheduling delay. Normalized latency normalizes the request end-to-end time by the total number of decode tokens (which is the visible output of the system). However, this ends up hiding specifics about metrics such as scheduling delay. For example, consider the example illustrated in Figure 2. Here, the scheduling delay is above 25s for almost 60% of the requests in vLLM, compared to Sarathi-Serve which has a maximum scheduling delay of 15s. However, the normalized latency for these systems differs only be a few hundred milliseconds! This is a result of the normalization by decode tokens (the median decode tokens in Arxiv-Summarization [18] is 228).

Don't accurately measure impact of jitters during decoding

Time Per Output Token (TPOT) and normalised latency hides jitters in token generation. Both these metrics normalise the latency by the number of decode tokens in the request. This normalization can mask the jitters that occur as intermittent stalls during token generation. As shown in Figure 3a, vLLM suffers a long stall of $10s$ (this can happen due to a long prefill request which is onboarded into the ongoing batch). While this will result in a very bad user experience, the impact of this stall on the TPOT or normalized latency metric will be numerically small due to the normalization by

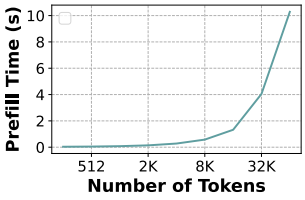
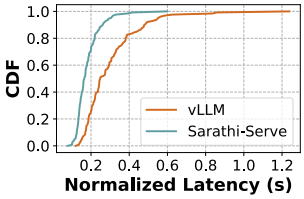
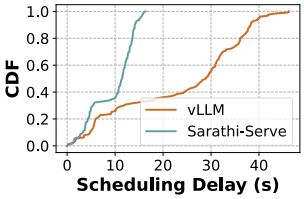


Figure 1: Increase in prefill latency with prompt length (Yi-34B on 2-H100) makes it infeasible to operate with fixed TTFT SLOs, especially for models with long context support.

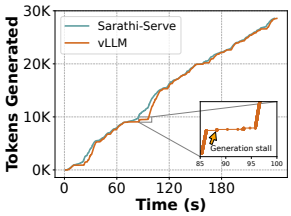


(a) Normalized Latency

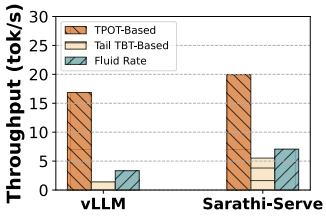


(b) Scheduling Delay

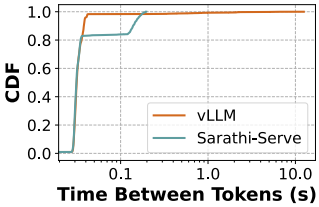
Figure 2: Normalized latency metric could be misleading as it obfuscates scheduling delay. On *arxiv_summarization* trace, 1.5 QPS, Yi-34B on 2-H100, while the scheduling delay is above 25s for 60% requests in vLLM, the normalized latency only differs by few hundred ms.



(a) Generation stalls in output tokens generation.



(b) Token throughput derived using various latency metrics.



(c) Decode token latency distribution.

Figure 3: (a) Decode tokens can be intermittently stalled due to prefills from incoming requests. (b) Naively normalizing total decode latency in TPOT, hides these latency spikes and overestimates the system token throughput. (c) Simply observing tail latency does not capture the nuances in the latency distribution. P85 latency for Sarathi-Serve is higher compared to vLLM while it has much lower P99 latency. Performance evaluations with *fluid token generation rate* accounts for all these variations and provides an accurate and balanced view of system performance. Here for *fluid token generation rate*, we enforce that 99% of the requests meet deadlines at least 90% of the time (*fluidity-index* > 0.9).

(typically) large number of decode tokens. Tail TBT latency, as employed by Sarathi-Serve [17], can highlight these generation stalls However, tail latency does not reveal the complete profile of magnitude and frequency of stalls at request level as shown in Figure 3c.

TBT CDFs do not reveal the magnitude and temporal characteristics of stalls. Due to the autoregressive nature of LLM inference, a delay in one token generation delays all subsequent tokens. As a result, a high tail TBT could have potentially occurred at the start of token generation, disrupting the user experience at the start itself. This is not captured by the TBT CDF. Also, as the query load increases, the frequency of stalls in systems like vLLM [25] and Orca [30] can go up because of prefill requests interleaved with ongoing decodes. While the tail of the TBT distribution gives some information about stalls, it does not reveal request level metrics, such as stall duration for each request, frequency and timing (say towards the start or end of its decodes) of stalls per request, etc.

TBT fails to account for non-uniform token generation strategies. Techniques such as speculative decoding [26] can generate multiple decode tokens for the same request in a single iteration. Suppose 3 decode tokens d_i, d_{i+1} and d_{i+2} are generated in one iteration in time T_i. Conventionally, the TBT for d_i will be T_i and zero for both d_{i+1} and d_{i+2}. Next, say the token d_{i+3} is generated in time $3 * T_i$. Naively, the TBT for d_{i+3} will be attributed as $3 * T_i$. However, the user actually saw 4 tokens generated in $4 * T_i$ time, and the last token generation delay can be easily hidden by the user-facing client which could have shown each of the 4 tokens arriving at a uniform rate. This observation inspires our deadline based latency metric.

Example. Consider a scenario where an end-user, evaluates two serving systems, vLLM and Sarathi-Serve, by passively observing their response to 1000 requests. Initially, when comparing the TPOT throughput, as illustrated in Figure 3b, both systems appear to perform similarly. However, analysis

using TBT reveals that vLLM suffers from a significantly longer tail TBT of 1 second, although it outperforms Sarathi-Serve between the 80th and 98th percentiles (see Figure 3c). Solely examining tail latency disproportionately penalizes vLLM, although both systems have comparable median TBTs. Thus, while TPOT downplays the discrepancies between the systems, tail-TBT overstates them. Neither metric accurately reflects the true system throughput under constraints of quality of service or user experience. Our proposed metric, *fluid token generation rate*, instead measures the actual token generate rate achievable by a system while meeting constraints on the user experience, for example, a constraint requiring a minimum percentage (e.g., 99%) of user requests meet a token generation deadline with at least *fluidity-index* of 0.9.

3.2 Desirable Properties of Evaluation Framework

Having identified the pitfalls of existing metrics in evaluating LLM serving frameworks, we articulate the essential attributes of an ideal evaluation metric. First, we need an evaluation framework that is blackbox (can evaluate any API endpoint), and workload agnostic (e.g., not impacted by variance in prompt lengths in the workload). Second, given the complexity of inferring system performance from a collection of metrics, there is a pressing need for a unified metric that not only simplifies analysis but also accurately reflects the user-facing performance of LLM serving systems, while incorporating the unique dynamics of the inference process. Lastly, the metric should comprehensively capture the frequency, duration, and timing of stalls within the system, addressing one of the most critical aspects affecting user experience.

4 Metron: Design and Implementation

Let us assume that we have the ideal TTFT and TBT for a given application based on some expectation on user behavior. The current TBT based SLO metrics treats each token generation independently, which may not capture the user experience well. For example, take a concrete example where the desired TBT is 100ms. Then, a system which produces 10 tokens at TBT of 10ms and the 11^{th} token at TBT of 150ms, will see the same TBT miss rate as a system which produces 10 tokens at TBT of 100ms and the 11^{th} token at TBT of 150ms. Clearly the first system is much more superior than the second, but the TBT miss rate itself does not capture that. What we propose instead is to incorporate a notion of *deadline* for each token's generation. Let us analyze our example in more detail. For the first system, if the token generation started at $t = 0$s, the first 10 tokens would then have been generated by $t = 100$ms while the 11^{th} token would have been generated at $t = 250$ms. If the reading speed of the user is say one token per 100ms, they would have ample time by the time they reach to the 11^{th} token (at $t = 1000$ms). Thus, the extra delay in the 11^{th} token generation would not be perceived by the user. In the second system, however, the user will actually perceive a delay while reading the 11^{th} token. This is very similar to the case of video streaming, where TTFT corresponds to the initial delay in video playback (this includes the load times, buffering, etc.), while TBT corresponds to the delay in generation of each frame and should be below $1/fps$, where fps is the video's frames per second. In the case of video playback, even if a frame is available earlier than desired, the client actively delays the frame playback to $1/fps$. Although this is not required in the case of LLM decode, the client may decide do display the tokens at TBT rate for a consistent experience even if they are available earlier. Based on this motivation, we propose a deadline based TBT acceptance rate metric, which we call *fluidity-index*.

SLO
TBT = 100ms

4.1 *fluidity-index* metric

Let the desired TTFT and TBT for a given application be D_p and D_d, respectively. Note that D_p will be a function of the number of prompt/prefill tokens. We then define the *deadline* for the generation of the i^{th} token as $D_i = D_p + i \times D_d$. As long as all tokens are generated within their deadline, D_i, the user will not perceive any delay or generation stall, even if some individual tokens see a delay of more than D_d between consecutive token generation. We then define a *deadline-miss* as an event when the actual token generation time of the i^{th} token exceeds D_i. Note that in the event of a *deadline-miss*, depending on the length of the stall, many subsequent tokens may miss their deadlines defined as above. This can be misleading as a single stall can amount to 10s of *deadline-misses*. To account for this, if there was a *deadline-miss* at the s^{th} token, we reset the deadlines for all subsequent

$\text{Deadline}_{\text{for token } i} = D_{\text{TTFT}} + i \times D_{\text{TBT}}$

$\text{Deadline miss} = \text{Actual time} > \text{Deadline}_{\text{for token } i}$

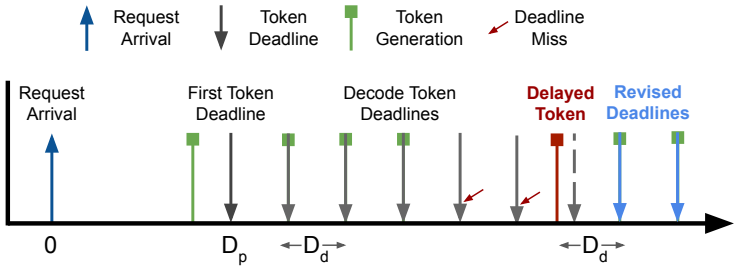


Figure 4: When a request arrives in the system, Metron sets the deadlines for all future tokens. If a token is produced before the set deadline, the slack is carried forward and serves as a buffer for future tokens. When a token arrives late, the system gets penalized for all the missed deadlines, and the subsequent deadlines are reset to account for autoregressive decoding process.

tokens to be $D_i = t_s + (i - s) \times D_d$, where t_s is the actual generation time of the s^{th} token, and compute the *deadline-miss*es of subsequent tokens based on these refreshed deadlines.

Algorithm 1 *fluidity-index* computation

Require: `inter_token_times:T, tbt_deadline:D_d, prefill_tokens:P, scheduling_slack:SD`
Ensure: *fluidity-index*

```
# Calculate the prefill deadline D_p
D_p ← predict_prefill_time(P) + SD
total_deadlines ← 0
missed_deadlines ← 0
slack ← 0
for i ← 0 to length(T) - 1 do
    t ← T[i]
    # Determine the appropriate deadline D
    D ← D_p if i == 0 else D_d
    if t ≤ D + slack then
        # Deadline met:  adjust slack and increment total deadlines
        slack ← slack + D − t
        total_deadlines ← total_deadlines + 1
    else
        # Deadline not met:  calculate deadline misses
        misses ← (t − slack − D) // D_d + 1
        missed_deadlines ← missed_deadlines + misses
        total_deadlines ← total_deadlines + misses
        slack ← 0
    end if
end for
return (total_deadlines - missed_deadlines) / total_deadlines
```

The metric. The detailed algorithm of how *fluidity-index* measures the percentage of accepted deadlines is as described in Algorithm 22. At a high level, we first set the deadline for all future tokens based on the prefill, decode latency target, while accounting for a small scheduling slack determined empirically. If a token is generated well before deadline, the slack is added to the future token generation, whereas in case of a missed deadline, we reset all future deadlines to start from the completion time of the token that missed the deadline as depicted in Figure 4.

Picking the larget latencies. Two important constants in the *fluidity-index* metric is the target latencies for the first generated token (prefill) and subsequent (decode) tokens denoted by D_p and D_d above. As discussed in §3.1, the prefill time increases quadratically in the size of input prompt. Therefore, setting a static value for D_p is impractical as the prompt length varies significantly in production scenarios. LMSys-Chat-1M [31] has 417 median number of prefill tokens and 1418 prefill tokens at 90^{th} percentile. Therefore, in Metron, we propose D_p to be a function of prompt length.

We use a reasonable open-source system like vLLM [14] as baseline to benchmark the prefill time as a function of input length by profiling the request in isolation, in the absence of any other variables like scheduling delays. We repeat this for 10 requests (tunable) and fit a curve through the observed points to obtain a D_p target as a function of input length. Note that, this is a recommendation on how to set the prefill latency target; different systems may observe different performance trends due to their implementation, optimizations, and kernels used. Therefore, the process of identifying this prefill latency curve should be repeated on the system being evaluated to set practical latency targets.

Picking the decode latency target D_d is fairly intuitive and straightforward. It depends on the application being evaluated. We define three targets in Metron– a strict target for interactive applications like chat (25ms), medium target for medium-priority users (50ms) and a relaxed target for low-priority users (100ms). These numbers in Metron can be picked based on production needs.

4.2 Evaluation workflow and implementation

Metron, provides a standardized evaluation workflow for both proprietary and open-source LLM inference frameworks. Metron provides two evaluation recipes as described below:

Black-box Evaluation. For an LLM inference API endpoint, Metron performs black-box evaluation by hitting the server with a set of requests with diverse prompt lengths, and tracks checkpoints such as the timestamps when each output token got generated, which allows it to calculate several metrics such as TTFT, TBT, and TPOT. Moreover, given a target threshold value for *fluidity-index* metric, such as – 99% of the requests have their *fluidity-index* of at-least 0.9, Metron infers the minimum TBT deadline that can satisfy this constraint. This allows us to obtain the maximum stable token throughput (*fluid token generation rate*) that can be served to the user in glitch-free manner.

Capacity Evaluation. Typically, while deploying an LLM inference service, the operator needs to determine the minimum number of GPUs required to serve the expected userbase with predefined service quality requirements. To aid this process, Metron provides a capacity evaluation module, which evaluates a system with different request loads to identify the maximum capacity each replica can provide while meeting the *fluidity-index* SLO requirements.

Implementation. We create a fork of open-source LLMPerf [8] which supports calling numerous LLM APIs, specifically, OpenAI compatible APIs [11]. We extend the codebase to support interacting with open-source LLM Inference Framework vLLM [14], calculating existing metrics discussed in §2.2, *fluidity-index* metric proposed in §4.1 and capacity search.

5 Evaluation

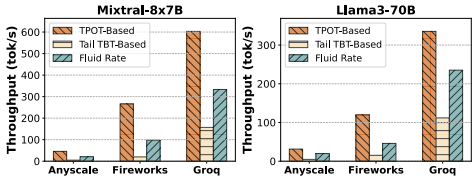
In this section we demonstrate the effectiveness of Metron in holistically evaluating the performance of different LLM inference systems both open source frameworks and proprietary offerings.

5.1 Evaluating Public APIs

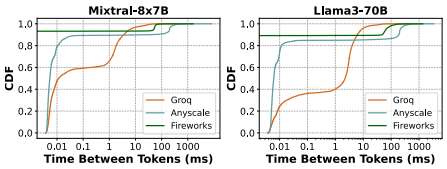
In this section, we demonstrate the effectiveness of Metron to benchmark public API endpoints. Metron performs black-box analysis on these systems to characterize their performance under various configurations. We evaluate three proprietary systems with API-only access: Anyscale [1], Groq [5], and Fireworks [4], across two models – a dense model LLaMA3-70B [9], and a MoE model Mixtral-8x7B [24]. We use a custom workload with varying prefill length (between 256 and 8k and maximum tokens to generate set to 256. Since the performance of public APIs can change throughout the day depending on request traffic and other factors, we run Metron once every hour for 24 hours to accommodate varying nature of traffic throughout the day.

Possible OS opportunity to evaluate Together.AI

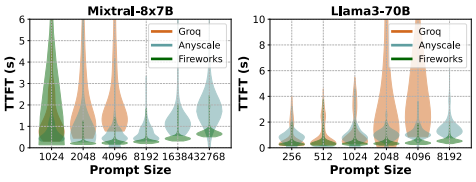
Results. Figure 5a plots the throughput of the three systems using three metrics – TPOT, tail TBT, and *fluidity-index*. For the first two, we plot the inverse of the observed mean TPOT across all requests, and the inverse of the 99th percentile TBT. For *fluid token generation rate*, we find the minimum TBT SLO D_d such that 99% percent of the requests have *fluidity-index* of at-least 0.9. The inverse of this is the *fluid token generation rate*.



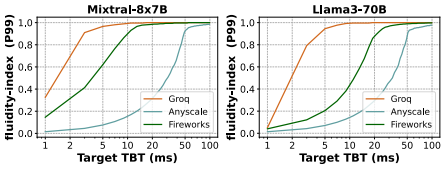
(a) Token Generation Rate.



(b) TBT Latency Distribution.



(c) TTFT Distribution.



(d) Acceptance rate as a function of TBT target.

→ Better overall picture

Figure 5: Evaluation of proprietary serving offerings for Mixtral-8x7B and Llama3-70B performed over duration of 24 hrs. (a) shows the token throughput as estimated by different decode latency metrics, (b) presents the overall decode latency distribution across all requests, (c) shows the TTFT for different prompt lengths and (d) provides a full characterization of the system by showing the *fluidity-index* as a function of target TBT.

We observe that Groq has the highest throughput of 600 tokens/s based on TPOT, a value that service providers oftentimes report. However, the tail TBT metric shows a 4× lesser throughput potentially due to jitters and stalls between decode iterations. The former is too relaxed and ignores generation stalls while the latter over penalizes the tail latency spikes. The throughput computed using *fluid token generation rate* lays a fair ground and shows the throughput that the system can sustain while providing a good user experience.

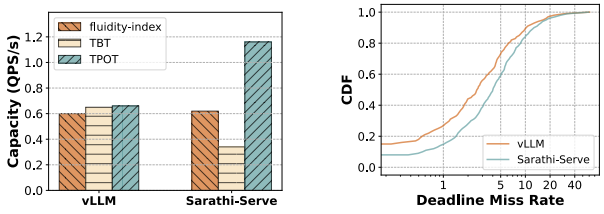
We make some interesting observations from the TBT distribution in Figure 5b. First, in Fireworks [4], roughly 90% of the decodes arrive together; hints at the potential use of speculative decoding. Second, all the offerings exhibit the S-shaped curve – indicative of prefill-decode interference due to long context length requests. Next, the TTFT distributions in Figure 5c shows that Fireworks consistently has the lowest TTFT as well as variance for all prompt lengths. The minimum TTFT for Anyscale overlaps with Fireworks indicating that their system can achieve similar prompt processing efficiency. However, the wide spread of TTFT in Anyscale indicates potentially high scheduling delays either due to high load or underprovisioning. Unlike decodes, Groq has the worst prefill efficiency.

Finally, the deadline miss rate plots in Figure 5d clearly highlight the differences in TBT across the three systems. Drawing a horizontal line at a desired miss rate (say 10%), we see that for both Mixtral-8x7B and LLaMA3-70B, Groq [5] has the best TBT, followed by Fireworks and Anyscale. While this was difficult to interpret in Figure 5b, *fluidity-index* highlights this difference.

5.2 Evaluating Open Source Systems

We now demonstrate the effectiveness of Metron in setting SLOs for deployment operators and capacity planning. We evaluate vLLM [14] and Sarathi-Serve [17] (via vLLM with *chunked-prefill* feature turned on). on LLaMA3-8B [9, 29] on a H100. We use rope-scaling to support prompt token lengths longer than 8192. Requests are randomly sampled from the Arxiv-Summarization [18] dataset that represents long context workloads.

Results. Earlier in §3.1 we compared the vLLM and Sarathi-Serve at a high load, where we observed that the tail TBT-based throughput for vLLM is 3× worse than Sarathi due to huge generation stalls, while the TPOT based throughput shows these systems at par. *fluidity-index* shows the true difference in throughput between these systems. Next, we compare these systems under a strict TBT SLO. We use *fluidity-index* to define the service SLO as – 99% of requests should have less than 10% deadline miss rate with 25ms target TBT. Metron then finds the maximum request load (QPS) at which the SLOs can be maintained. We also consider TPOT and P99 TBT based SLOs as baselines with the target latency of 25ms. Here, Sarathi has a 2x lower token throughput compared to vLLM as



(a) System Capacity. (b) Deadline Miss Rate CDF.

Figure 6: Capacity evaluation of open source systems vLLM and Sarathi-Serve performed on H100 for Llama3-8B. (a) shows the overall capacity achieved obtained by using different decode lat. metrics - TBT, TPOT, and *fluidity-index*. (b) captures the distribution of deadline miss rate at capacity point for the two systems.

measured by tail TBT-based metric, in contrast to Figure 3. At a budget of 25ms, almost all the mixed batches created by Sarathi-Serve with prefill chunks violate the latency SLO threshold. Figure 6b, shows the distribution of deadline miss rate when both the systems are operating at the capacity. Also, note that in this setting, the *fluidity-index* based capacity is the same at 0.6 QPS. We find that vLLM has a higher deadline miss rate at lower percentile. This is expected because, vLLM will have to ingest prefills in their entirety from time to time and whenever it does so, the TBT deadline is breached. In contrast, with chunked prefills Sarathi-Serve achieves fewer deadline misses and higher fluidity.

Maybe DUE to continous batching?

6 Discussion

Metron provides an evaluation framework for LLM inference using *fluidity-index* metric that tracks the missed deadlines per request. We now discuss the challenges that we leave to future work.

The *fluidity-index* metric requires setting a deadline for every token – for the first token of every request, this is the target latency for the prefill phase. In this paper we discuss a potential mechanism for selecting prefill latency target based on observing the prefill processing curve across varying prompt sizes. However, picking a deadline for a given prompt length is challenging for proprietary systems as we cannot accurately characterize their prefill performance; the observed prefill time can include scheduling delays which may offset the expected trends in prefill processing. We leave it to future work to explore alternate ways of prefill latency target selection for propreitery system evaluation.

Next, we observe that we need to provide a small scheduling slack in deadline computations as discussed in §4.1. We pick this value based on our empirical observations; we leave it to future work to systematically set a scheduling slack. Finally, open-source systems have various performance tuning knobs; for e.g., chunk size in Sarathi-Serve, block size in vLLM etc. Metron currently does not explore or auto-tune such parameters; it expects the users to set the configuration parameters while evaluating across two systems.

7 Related Work

Machine learning inference systems have been studied extensively over the last decade. TensorFlow-Serving [27], Clipper [19], BatchMaker [21], and Clockwork [22] propose various caching, placement, and batching strategies to improve general model serving. More recently works including Orca [30], vLLM [25], Sarathi [16] primarily addresses the dedicated challenges faced in auto-regressive transformer inference using efficient memory management and scheduling. SplitWise, DistServe and TetriInfer [28, 32, 23] have presented options to disaggregate the prefill and decode phases to eliminate the interference between them.

8 Conclusion

Evaluating LLM inference systems is a challenging problem due to the unique characteristics of autoregressive decode process. We presented a detailed analysis of existing evaluation metrics and their pitfalls. To address their shortcomings, we introduce Metron– a holistic LLM evaluation framework that instantiates a novel approach comprised of a novel *fluidity-index* based approach to evaluating LLM inference systems in a user-facing manner. We then show how Metron can be leveraged to evaluate both open-source and proprietary model serving systems. Metron is aimed to serve as a standard evaluation suite for LLM inference systems.

References

[1] Anyscale. https://www.anyscale.com.

[2] Azure ai studio: A unified platform for developing and deploying generative ai apps responsibly. https://azure.microsoft.com/en-in/products/ai-studio.

[3] Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

[4] Fireworks: Generative ai for product innovation! https://fireworks.ai/.

[5] Groq. https://groq.com.

[6] Lightllm: A light and fast inference service for llm. https://github.com/ModelTC/lightllm.

[7] Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices.

[8] Llmperf: A tool for evaulation the performance of llm apis. https://github.com/ray-project/llmperf.

[9] Meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/.

[10] Openai. https://openai.com/.

[11] Openai api. https://platform.openai.com/docs/api-reference/introduction.

[12] Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM.

[13] Text generation inference. https://huggingface.co/text-generation-inference.

[14] vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm.

[15] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara*, 2024.

[16] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.

[17] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.

[18] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[20] Jeremy P. Erickson and James H. Anderson. *Soft Real-Time Scheduling*, pages 233–267. Springer Nature Singapore, Singapore, 2022.

[21] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[23] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR, 23–29 Jul 2023.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[28] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng

Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[30] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[31] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

[32] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.