Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services

Jiachen Liu
amberljc@umich.edu
University of Michigan

Zhiyu Wu
zhiyuwu@umich.edu
University of Michigan

Jae-Won Chung
jwnchung@umich.edu
University of Michigan

Fan Lai
fanlai@illinois.edu
UIUC

Myungjin Lee
myungjle@cisco.com
Cisco

Mosharaf Chowdhury
mosharaf@umich.edu
University of Michigan

Abstract

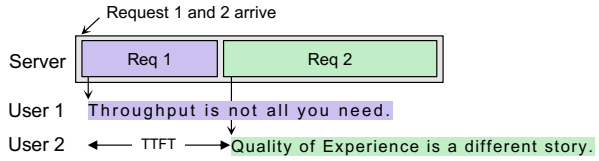
The advent of large language models (LLMs) has transformed text-based services, enabling capabilities ranging from real-time translation to AI-driven chatbots. However, existing serving systems primarily focus on optimizing server-side aggregate metrics like token generation throughput, ignoring individual user experience with streamed text. As a result, under high and/or bursty load, a significant number of users can receive unfavorable service quality or poor Quality-of-Experience (QoE).

In this paper, we first formally define QoE of text streaming services, where text is delivered incrementally and interactively to users, by considering the end-to-end token delivery process throughout the entire interaction with the user. Thereafter, we propose Andes, a QoE-aware serving system that enhances user experience for LLM-enabled text streaming services. At its core, Andes strategically allocates contended GPU resources among multiple requests over time to optimize their QoE. Our evaluations demonstrate that, compared to the state-of-the-art LLM serving systems like vLLM, Andes improves the average QoE by up to 3.2× under high request rate, or alternatively, it attains up to 1.6× higher request rate while preserving high QoE.

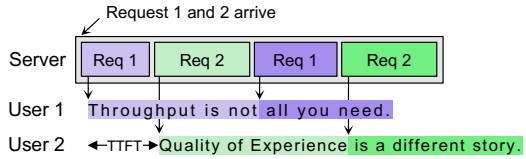
1 Introduction

Large language Models (LLMs) [4, 9, 21, 46, 51] have revolutionized natural language processing. By generating contextually relevant responses, they power a wide range of applications, more than 60% of which are centered around conversational interactions like chatbots, virtual assistants, language translation, and customer support systems [15]. In particular, the meteoric rise of ChatGPT [35] spearheaded the growth of conversational AI services by attracting over 100 million users in just two months after its launch [29].

Conversational AI services, by nature, provide *interactive* conversations between the user and an AI agent. At its core, an LLM generates tokens one by one[1] and streams them back to the user to be digested, be it as written text or speech. As this token-by-token streaming nature is akin to the frame-by-frame streaming nature of video streaming services, we dub such services *text streaming services*.



(a) Existing LLM serving systems are oblivious of QoE. User 2 experiences a long wait time (TTFT) and therefore lower QoE.



(b) A QoE-aware LLM serving system can schedule token generation over time to enhance QoE. User 2's TTFT is drastically improved without affecting User 1's token delivery timeline.

Figure 1. Server-side token generation timeline and user-side response digestion progress. Even if the server generates tokens very fast, users cannot digest them at such a pace.

In this paper, we seek to characterize and enhance the Quality-of-Experience (QoE) of text streaming services (§2.2). We realize that user interaction with LLM responses happens at moments when each new token is delivered (e.g., displayed or spoken) to the user over time. Thus, we define token delivery timeline (TDT), a series of timestamps when each token was delivered to a user, to capture the user's interaction with the service for a single request. The ideal TDT a user expects from a text streaming service can vary significantly based on the type of the service and user demographics. For instance, a chat service that uses a text-to-speech model to read out the LLM's response to users (e.g., voice chat in ChatGPT, real-time speech translation) could be less stringent in terms of its minimum token delivery speed (TDS) compared to a chat service in raw text, because a user's speaking speed is often slower than their reading speed, but it may require smaller time to first token (TTFT) to better resemble real-life

[1] LLMs process and generate text in units of *tokens*. For instance, the word "streaming" may be broken down into two tokens: "stream" and "ing."

verbal conversations. The minimum TDS and TTFT together define the *expected TDT* of a request.

Unfortunately, existing LLM serving systems [20, 25, 30, 50] are designed to optimize aggregated server-side performance metrics such as token generation throughput [25, 50], which are not necessarily aligned with optimizing the QoE of text streaming services (§2.3). More importantly, by realigning the objectives of LLM serving systems towards QoE optimization, a QoE-aware serving system can utilize the same resources more effectively to manage a greater number of concurrent requests while ensuring high QoE, thus reducing the cost per request. To illustrate, we compare existing serving systems with a QoE-aware one, each with a serving capacity of 1, in Figure 1. In Figure 1a, due to the commonly adopted first-come-first-serve (FCFS) scheduling policy [25, 50, 52], User 2 experiences a long initial waiting time (TTFT). In contrast, in Figure 1b, a QoE-aware serving system schedules token generation in a manner that is aware of each user's reading speed, leading to a shorter wait time for User 2 without affecting User 1's interaction with the service. Although the average server-side token generation throughput or latency are the same for the two systems, overall user experience is improved in the QoE-aware system.

We attribute this to the naïve FCFS scheduling policy in existing serving systems, which fails to account for the QoE requirements of individual requests and cannot efficiently utilize resources (§2.4). Consequently, some users may experience extended waiting time during their interaction with the service, especially when the system is under higher request rate or is serving requests with longer context lengths. To preserve good user experience, the service provider must provision more compute resources proportional to the excess request load, leading to higher operational costs.

Designing a QoE-aware LLM serving system, however, is challenging from both conceptual and practical perspectives. Defining the QoE metric to capture the user experience in text streaming services is non-trivial. It should encapsulate the continuous interaction process over time, accounting for factors like TTFT and TDS. Designing a QoE-aware serving system faces several systems challenges as well:

(a) **Dynamic and unpredictable resource demand:** Requests arrive dynamically with varying expected TDT and prompt length and the number of output tokens is not known a priori, making it challenging to implement a one-size-fits-all scheduling strategy such as round-robin.

(b) **Constrained resource supply:** The system has limited GPU memory and computation resources, restricting the number of concurrent in-flight requests. To meet the QoE requirements of individual requests, the system needs to make runtime decisions to allocate resources among requests, which may incur non-negligible overhead.

To this end, we first propose a mathematical definition of QoE for text streaming services (§3.1). Our QoE metric compares the actual TDT of a request with its expected TDT, reflecting the user's experience throughout their *entire* interaction with the service. Then, we propose Andes, an LLM serving system that optimizes the overall QoE of text streaming services (§4). Andes employs a dynamic priority-based preemptive scheduler that operates at the granularity of tokens. Andes strategically allocates system resources to more urgent requests and preempts requests that have already received sufficient service, all to enhance QoE. By satisfying more requests with high QoE using the same amount of resource, Andes eliminates the need for additional resource provisioning, thus reducing LLM serving cost. Andes also co-designs a client-side *token buffer* that temporarily withholds excess tokens and displays them to the user at their expected pace (§5). This design ensures users experience smooth token delivery, oblivious to the intricacies of server-side scheduling or network fluctuations.

Age Group	Reading Speed
18-24 (28.0%)	236 WPM
25-44 (51.9%)	200 WPM
45-54 (11.2%)	192 WPM
55-64 (5.6%)	185 WPM
65+ (3.3%)	175 WPM

Table 1. Reading speed (Word Per Minute) by age group [10, 29].

Language	Speaking Speed
English (79.3%)	150 WPM
Chinese (7.0%)	158 WPM
Korean (6.9%)	150 WPM
French (3.6%)	195 WPM
Spanish (3.2%)	218 WPM

Table 2. Speaking speed (Word Per Minute) by language [8, 29, 36].

We evaluate Andes using the OPT [51] family of models, ranging from 13B to 175B parameters (§6). Compared to vLLM [25], we find that Andes can manage 1.6× higher request rate with high QoE, or alternatively, improve the average QoE by 3.2× given the same amount of resource.

Overall, we make the following contributions in this paper:

1. We identify an emerging category of LLM-based applications (text streaming services) and define a QoE metric for them.
2. We propose Andes, a QoE-aware LLM serving system designed to optimize QoE for text streaming services.
3. We evaluate Andes under different workloads and setups and show that Andes significantly improves QoE with negligible system overhead.

2 Background and Motivation

In this section, we introduce the unique characteristics of LLM serving systems (§2.1) and the user experience of text streaming services (§2.2). We then discuss the opportunities for improving user experience (§2.3) and the limitations of existing solutions (§2.4).

2.1 LLM Serving Systems

LLM text generation using Transformer-based [47] models is characterized by autoregressive token generation and significant memory usage. First, the LLM generates tokens

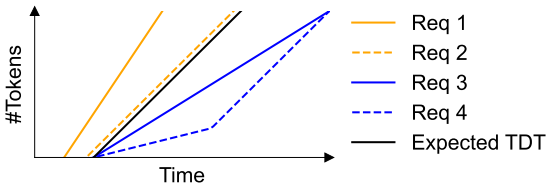


Figure 2. Four requests arrive at $t = 0$. Requests 1 and 2 are equally satisfying. Requests 3 and 4 are frustrating, with request 4 being more so as it delivers fewer tokens earlier on, despite having the same TTFT and average token latency.

sequentially, where the next token is conditioned on the previous tokens. Second, the LLM requires a large amount of memory to store intermediate data for each token in its input prompt and output response, known as *KV cache* [47]. As the number of tokens generated increases, so does the KV cache size. For instance, GPT-3 175B [9] requires 7 GB of GPU memory for a 1000-token request, limiting the number of requests that can be handled concurrently.

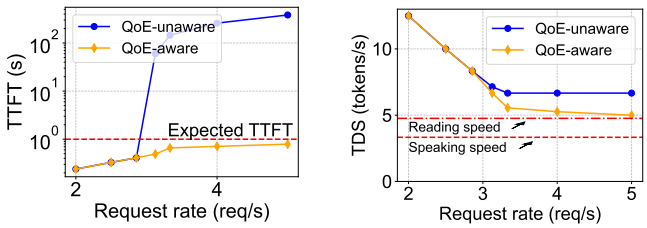
2.2 User Experience of Text Streaming Services

Compared to traditional services that generate entire responses at once, *text streaming services* allow the user to start digesting the response as early as possible. The user experience includes two phases:

Wait Phase. Users wait for the first token to arrive, known as the *time-to-first-token* (TTFT). For web applications, studies indicate that users expect an initial response to arrive within one second, with a significant 32% dropout rate if the response takes longer than three seconds [6].

Digest Phase. Following the first token, users enter the digest phase, which may last for tens of seconds or more [50], Hence, it is a common practice to *stream* tokens to the user on the fly so that they can start digesting the response as early as possible. The expected rate of token delivery, i.e., the Token Delivery Speed (TDS), depends on factors such as application type and user demographics. For example, reading speeds, measured in words per minute (WPM), differ across age groups (Table 1), while speaking speeds vary among languages (Table 2). By translating words to tokens using the average word-to-token ratio [38], we can estimate the average reading speed to 4.8 tokens/s and average speaking speed to 3.3 tokens/s.

Intuition Behind QoE of Text Streaming Services. The expected TTFT and the expected TDS together define the expected *token delivery timeline* (TDT), represented by the black line in Figure 2. Similar to QoE in video streaming, a desired QoE metric should capture the gap between the actual TDT and the expected TDT. Intuitively, users are satisfied when the actual TDT is above the expected TDT, otherwise, they prefer to receive more tokens earlier on, as illustrated in



(a) 90^{th}-p TTFT increases dramatically as the request rate surpasses the server's capacity.

(b) Token generation speed is much faster than the user-expected speed.

Figure 3. System performance under different request rates.

Figure 2. Therefore, the QoE should comprehensively measure the token delivery timeline throughout the *entire* user interaction, going beyond an aggregated number like TTFT or average token latency. We formally define such a QoE metric in Section 3.1.

2.3 Problems and Opportunities

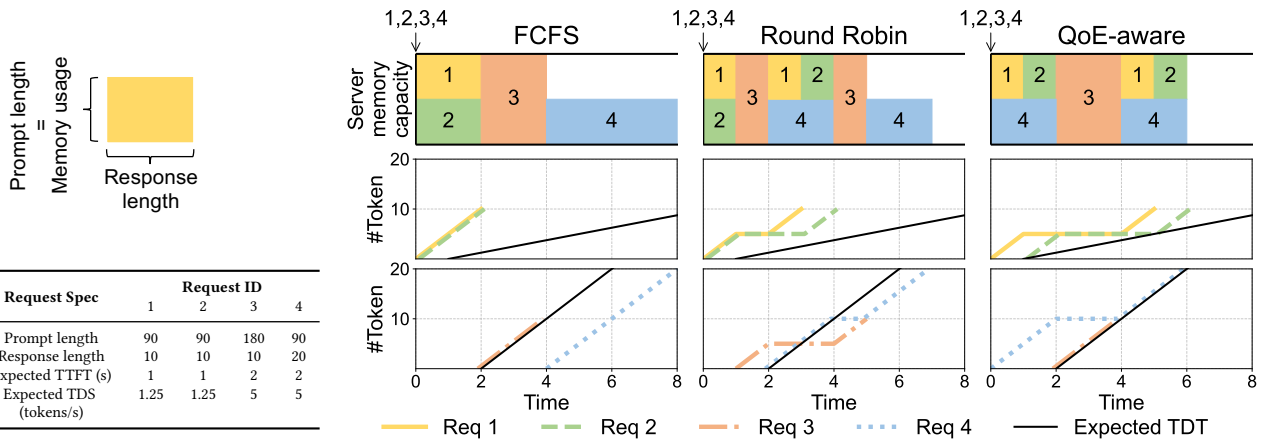
Existing LLM serving systems have primarily focused on optimizing aggregated server-side metrics, and often employ a first-come-first-serve (FCFS) scheduling approach without considering the user experience. In our experiment with ShareGPT [45] on OPT 66B [51] with 4 A100 GPUs, we notice that especially under high request rate, two issues arise: (1) certain users may encounter extended TTFT; (2) conversely, other users might receive tokens at a pace surpassing their digestion ability.

Prolonged TTFT. As depicted in Figure 3a, the 90^{th} percentile TTFT increases dramatically as the server faces more bursty request rates, resulting in a longer queuing delay and degraded user experience. To accommodate such bursty request volumes, service providers often have to over-provision resources, such as by adding more GPUs, which significantly increases operational costs.

Excessively High Token Generation Speed. Conversely, as shown in Figure 3b, we report the token generation speed under different request rates. The observed server-side token generation speed ($\geq$6.6 tokens/s) is much faster than the user-expected speed (3.3 or 4.8 tokens/s), as referenced in Table 1 and Table 2. This discrepancy indicates that the server often generates tokens faster than the user can consume them. While this might seem efficient from the server's perspective, it may overwhelm this user while starving others.

Opportunities. We observe that there is an opportunity to optimize user experience by balancing prolonged TTFT and excessively fast token generation speed. By temporarily pausing the response generation for requests with already sufficient tokens generated, we can spare the limited GPU resources to other pending requests.

The ratio between the expected token generation speed TDS_{expected} and the actual token generation speed TDS_{actual}



Request Spec	Request ID 1	2	3	4
Prompt length	90	90	180	90
Response length	10	10	10	20
Expected TTFT (s)	1	1	2	2
Expected TDS (tokens/s)	1.25	1.25	5	5

Figure 4. Suboptimal user experience from QoE-unaware scheduling policies. In this illustrative toy example, we consider a server that can serve at most 200 tokens simultaneously due to memory constraints. We consider four requests with different prompt lengths, response lengths, as well as different expected TTFT and TDS values, arriving at time 0. The figure shows the serving order (first row) and the cumulative tokens delivered over time for each request (second and third rows). Colored lines represent actual TDT, while the black line indicates the expected TDT. An optimal QoE is achieved when the actual token delivery curve is completely left and/or above the expected token delivery curve.

determines the slack for which a request can be preempted, allowing the system to accommodate more concurrent requests. Thus, with appropriate request preemption and restarting, we can serve $\frac{TDS_{\text{actual}}}{TDS_{\text{expected}}}\times$ concurrent requests than without request preemption, significantly improving user experience. In the example of text-based and voice-based chat services in Figure 3b, we could have increased the serving capacity by $\frac{6.6}{4.8} = 1.38\times$ and $\frac{6.6}{3.3} = 2\times$, respectively. Our evaluation shows that Andes can nearly achieve this theoretical improvement in practice.

2.4 Limitation of Existing Solutions

Let us consider a toy example in Figure 4 to illustrate the limitations of existing QoE-unaware scheduling (FCFS used by vLLM [25] and Round Robin). Under FCFS scheduling, while requests 1, 2, and 3 are served immediately, request 4 suffers from longer TTFT due to queuing delays. Round Robin partially mitigates queuing delay using fair-sharing but still fails to align the token delivery in the later stage of the interaction, leading to suboptimal QoE.

In contrast, the QoE-aware policy manages to meet the QoE requirements for all requests by prioritizing requests based on their QoE requirements and resource demand. It prioritizes requests with stringent TTFT requirements. Meanwhile, it monitors the resource demand of each request to prevent small requests from being starved of necessary resources. As the served requests accumulate enough tokens for the user to digest, the system upgrades the priority of request 3, which then requires more urgent servicing, and serves it. Finally, the system brings back requests 1, 2, and 4 to continue supplying tokens.

In sum, when the server load is below its capacity, all requests can be served promptly and achieve perfect QoE without smart request scheduling. However, when the server is operating at capacity due to unpredictable higher request loads, QoE-aware scheduling can significantly improve the user experience without over-provisioning resources.

3 Overview

In this section, we first introduce a formal definition of Quality-of-Experience (QoE) for text streaming services (§3.1). Then, we provide an overview of Andes, an LLM serving system that optimizes QoE of text streaming services (§3.2).

3.1 Quality-of-Experience (QoE) in Text Streaming

Text streaming services allow the developer to specify the expected token delivery timeline (TDT) in a request. We derive the QoE of a request by comparing its actual TDT with the expected TDT, considering the entire token delivery process.

Informed by the distinctions between superior and inferior service depicted in Figure 2, the formulation of our QoE metric is guided by a set of principles that reflect user expectations and experiences throughout their interaction:

1. **Perfect Satisfaction:** Users are satisfied when the actual token delivery perfectly aligns with or exceeds the expected delivery, resulting in maximum QoE (QoE = 1). We normalize QoE $\in [0, 1]$ for generality across applications.
2. **Excess Token Delivery:** At any given time, delivering tokens faster than the user's digest speed does not add

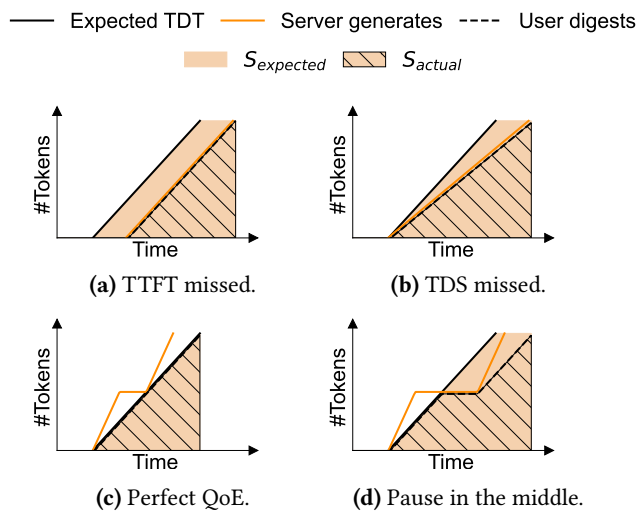


Figure 5. QoE example. The slope of the actual token delivery curve on the user side is capped by the expected TDS.

value to the user experience, as the user cannot digest all tokens at once. So the QoE remains unchanged.

3. **Early Token Delivery:** Users prefer receiving more tokens earlier to start processing the response sooner. In scenarios where perfect satisfaction is not achieved, the QoE is higher for scenarios where more tokens are delivered earlier. For example, the QoE is worse for a longer TTFT with the same TDS, and similarly, the QoE is worse for a slower TDS with the same TTFT.

Following these principles, we formalize the QoE metric by comparing two curves: (a) The expected token delivery curve $T(t)$ that is defined by expected TTFT and TDS. Specifically, $T(t) = TDS_{\text{expected}} \cdot (t - TTFT_{\text{expected}})$ represents the ideal timeline at which tokens should be delivered to the user (black lines in Figure 5). (b) The actual token delivery curve $A(t)$ reflects the timeline of how tokens are digested by the user over time (black dotted lines in Figure 5), with its slope at any time capped by the expected TDS.

To quantify the QoE of a request with response length l, we measure the area under both curves up to the actual time to the last token (TTLT). We then define QoE as the ratio of the actual and expected areas, as shown in Figure 5:

$$QoE = \frac{S_{\text{actual}}}{S_{\text{expected}}} = \frac{\int_0^{TTLT} A(t)dt}{\int_0^{TTLT} \min(T(t), l)dt} \quad (1)$$

This formulation focuses on the relative QoE relationship between services, but Andes allows the service provider to prioritize specific aspects. For example, to stress a shorter TTFT, the provider can add a penalizing term on the defined QoE as $\alpha^{TTFT_{\text{actual}} - TTFT_{\text{expected}}} \cdot \frac{S_{\text{actual}}}{S_{\text{expected}}}$, where $\alpha \in [0, 1]$. In this paper, we will use the QoE definition in Equation 1 by default.

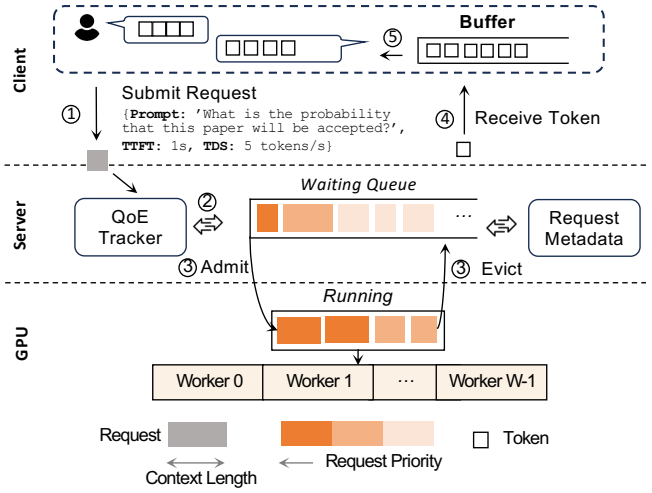


Figure 6. Andes Overview.

3.2 Andes Overview

The workflow of Andes is shown in Figure 6. ① The interaction begins with the user submitting a request to the server. The request comes with its QoE requirement, which is prespecified by the application developer. ② Upon receiving the request, the QoE tracker assigns a scheduling priority and puts it in the waiting queue. ③ At each scheduling iteration, the QoE tracker refreshes the priorities of all requests, both in the waiting and running queues. Then Andes reschedules the requests based on their priorities by admitting high-priority waiting requests to GPU workers and evicting low-priority running requests back to the server. For these evicted requests, their states (e.g., KV cache) are stored in the request metadata store on CPU RAM for future retrieval. ④ During each inference iteration, each running request generates one token, which is then sent to the client. ⑤ As tokens are delivered to the client, a token buffer is responsible for storing excess tokens and displaying them at the expected speed, ensuring smooth token delivery.

4 QoE-Aware Scheduling

In this section, we describe how Andes schedules token generation across multiple requests to maximize the total QoE. Section 4.1 formulates the scheduling problem as a Knapsack variant, and Section 4.2 introduces an efficient solution.

4.1 Problem Formulation

The core of Andes is an online preemptive scheduling algorithm for token generation, which requires designing three elements: (1) How often to make scheduling decisions (time quantum), (2) which requests to serve (scheduling objective), and (3) how many requests to serve at a time (batch size).

Time Quantum. At the beginning of each time quantum, the scheduler inspects both queued and running requests, and determines which ones to admit and preempt. Following the

continuous batching used in existing systems [25, 50], Andes invokes its scheduler at the beginning of each iteration.

Scheduling Objective. Just like any other online serving system, it is impractical to perfectly plan execution into the future. Therefore, Andes serves the set of requests that maximizes the scheduling objective in the upcoming time frame of length Δt. The parameter Δt cannot be too short, as scheduling decisions will become shortsighted, or too long, as the actual system state would deviate too far from estimations. We find that setting it as the average request completion time is reasonable, and show in Section 6.5 that Andes is not sensitive to the setting of Δt.

Andes supports various scheduling objectives including max average QoE and max-min QoE by designing its scheduling objective function appropriately. For the sake of presentation, we will focus on maximizing average QoE here (See Appendix A for alternative objectives). The objective function for request i is defined as:

$$Q_{\text{serve},i} - Q_{\text{wait},i} \tag{2}$$

where $Q_{\text{serve},i}$ and $Q_{\text{wait},i}$ are the QoE of request i after Δt if it is served and not served, respectively. In simple terms, Equation 2 is the amount of *QoE gain* when we decide to serve request i compared to when it is not served, and we naturally want to serve more of the requests that give us large QoE gains when served.

Batch Size. The number of requests picked to run in the upcoming quantum, or batch size, is limited by two factors.

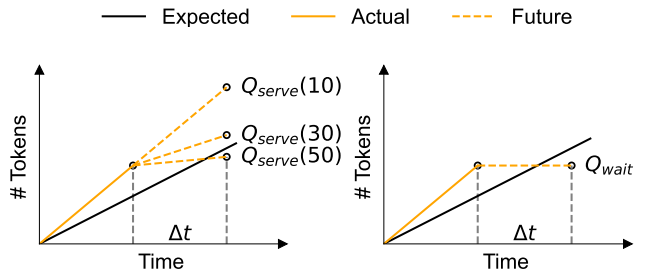
First, each token in a request's context (prompt plus all generated tokens) consumes one entry in the LLM serving system's KV cache [9], whose size is bounded by GPU memory. Thus, we have the following constraint:

$$\sum_{i=1}^{N} l_i x_i \le M \tag{3}$$

where there are N requests in total (queued or running), l_i is request i's context length, x_i is an indicator variable that is 1 if request i is served and 0 otherwise, and M is the total number of tokens that can fit in GPU memory.

Furthermore, Andes must take into account the latency to generate one token. That is, while a large batch size may increase server-side token generation throughput, the increase in the amount of compute will inflate the latency to generate one token from the perspective of each request, potentially hurting their QoE by delaying TTFT or failing to meet the expected TDS. On the other hand, a small batch size will be able to deliver tokens faster to each running request, but in turn more requests will not be served at all, again potentially hurting their QoE. Thus, the right intermediate batch size will have to be chosen in order to maximize average QoE.

Knapsack Formulation. Putting these together, we observe that the problem setting resembles that of the classic knapsack problem [23]. The goal is to select items (requests) to put in a knapsack (GPU) so that total item value (QoE gain) is maximized and total weight (l_i) does not exceed the knapsack's capacity (M).



(a) $Q_{\text{serve, i}}(B)$ **(b)** $Q_{\text{wait},i}$

Figure 7. Visualization of $Q_{\text{serve, i}}(B)$ and $Q_{\text{wait},i}$. The former depends on batch size B whereas the latter is a constant. With batch size 50, request i no longer has perfect QoE.

However, our problem setting deviates from that of the classical knapsack because the value of each item depends on how many items there are in the knapsack. This is because, as noted above, the number of requests in the knapsack (batch size) affects token generation latency, which in turn means that $Q_{\text{serve},i}$ is actually a function of batch size B.[2] Figure 7 visualizes this. When B is just 10 or 30, the request maintains perfect QoE by always running ahead. However, when B is 50, the computation time of one iteration becomes longer and slows down token generation, degrading the request's QoE by failing to meet its TDS expectation. On the other hand, $Q_{\text{wait},i}$ does not depend on the batch size because it simply sits in the queue, waiting to be served.

Thus, for a specific batch size B, we would like to solve:

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{N} \left(Q_{\text{serve},i}(B) - Q_{\text{wait},i}\right) \cdot x_i \\ \text{s.t.} \quad & x_i \in \{0,1\},\ i \in 1,\ldots,N \\ & \sum_{i=1}^{N} x_i = B \\ & \sum_{i=1}^{N} l_i x_i \le M \end{aligned} \tag{4}$$

where the optimization variable x is a length N array of x_is. The second constraint ensures that exactly B many requests are chosen, whereas the final constraint ensures that the GPU memory capacity is not exceeded. Equation 4 should be solved for each possible batch size B and the solution that yields the best objective value should be selected.

[2] More precisely, token generation latency is a function of batch size and the total number of tokens in the batch, but batch size and total number of tokens are nearly perfectly correlated, allowing us to eliminate the latter and only leave batch size. See Appendix B for more detailed analysis.

4.2 Solution Design

In this section, we discuss the hardness of the problem formulated in the previous section in terms of algorithmic hardness and systems overhead. Then, we propose efficiency optimizations and a greedy algorithm that gives an approximate solution with low systems overhead.

Algorithmic Hardness. As Andes must solve its optimization problem repetitively online to determine the set of requests to solve, an efficient algorithm is needed. However, Equation 4 is a variant of the knapsack problem called the *Exact K-item Knapsack*, which is weakly NP-Hard [23].

We give an optimal 3D dynamic programming solution to the problem that runs in pseudo-polynomial time $O(M \cdot N^2)$ in Appendix C. However, such an algorithm is also too slow in our case as the number of requests N and the maximum number of tokens that can fit in memory M are easily in the order of hundreds and thousands, respectively. Furthermore, we need to solve Equation 4 for each possible batch size $B \in [1, N]$, which is clearly intractable.

Preemption Overhead. When some requests that were running in the previous time quantum are not selected to run on the next, such requests are preempted. This is the core mechanism that reduces TTFT inflation from head-of-line blocking. For this, Andes supports two preemption mechanisms: swapping and recomputation. The former moves the request's KV cache entries between the GPU and CPU memory, whereas the latter drops all entries on preemption and recomputes them when the request restarts. If Andes runs out of host memory for storing KV cache, the preemption mechanism will automatically switch to recomputation.

Preemption is not free – in general, the latency overhead of swapping is similar to one token generation iteration (See Appendix D for detailed benchmarking). Frequent preemption may slow down token generation and delay token delivery, potentially degrading request throughput and QoE. Therefore, our scheduling algorithm must make preemption decisions that strike a good balance between reaping QoE gains and causing slowdowns.

Optimization #1: Selective Triggering. We observe that Equation 4 only needs to be solved when batch size is limited either by memory capacity or computation time. The former case can be detected easily by monitoring the KV cache occupancy and having a high-memory watermark (e.g., 90%). For the latter case, Andes monitors token generation latency and detects when it begins to exceed the most minimum token delivery speed requirement of the most stringent request. In all other cases, Andes does not trigger the optimization problem solver and serves every request.

Optimization #2: Batch Size Search Space Pruning. In order to reduce the number of times Equation 4 needs to be solved, we reduce the search space of batch size B from $[1, N]$ to $[B_{\min}, B_{\max}]$. First, there is no point in exploring very large batch sizes that cannot be realized. Thus, $B_{\max}$ is determined by adding to the batch requests with the shortest context lengths until the total number of tokens in the batch reaches M, at which point the batch size is the largest that can be realized. On the other hand, very small batch sizes that can generate tokens faster than the expected TDS of *any* request are also suboptimal. This is because going that fast does not increase the QoE of requests that are served, but on the other hand will serve a smaller number of requests, potentially degrading the QoE of requests that are left waiting. Thus, $B_{\min}$ is set as the largest batch size that generates tokens faster than the most stringent TDS among all requests.

Algorithm 1 Greedy packing algorithm for Equation 4

Inputs:
Number of requests N and KV cache capacity M
Request context length array $l[N]$
Request QoE gain array $q[N]$
Target batch size B
Output: Solution array $x[N]$

Initialize priority array $p[N]$ with all zeros
for $i = 0$ **to** $N - 1$ **do**
  $p[i] = \frac{q[i]}{l[i]}$ ▷ Priority of request i
$M_{\text{current}} = 0$
$N_{\text{current}} = 0$
Initialize solution array $x[N]$ with all zeros
for all $i \in [0, N-1]$ in descending order of $p[i]$ **do**
  if $M_{\text{current}} + l[i] \leq M$ and $N_{\text{current}} + 1 \leq B$ **then**
    $x[i] = 1$ ▷ Serve request i
    $M_{\text{current}} = M_{\text{current}} + l[i]$
    $N_{\text{current}} = N_{\text{current}} + 1$
  else
    break
return x

Optimization #3: Greedy Packing for Knapsack. A direct solution to the exact k-item knapsack problem in Equation 4 is computationally too heavy. Instead, Andes designs an efficient algorithm that computes each request's *priority* and greedily packs requests in that order. In designing the priority function, we have three goals:

(a) **Reflecting merit:** Requests that yield high QoE gain and consume less resource should have high priority.
(b) **Preventing starvation:** Requests should be automatically deprioritized as they receive service.
(c) **Reducing preemption:** Selecting high priority requests should reduce the need for preemption.

In light of these goals, request i's priority is defined as:

$$\frac{Q_{\text{serve},i}(B) - Q_{\text{wait},i}}{l_i} \tag{5}$$

This priority function meets our goals. (a) A higher QoE gain will increase the request's priority, but simultaneously discounted by the amount of GPU memory it will use. (b) As

a request receives service, its context length (l_i) will increase, automatically deprioritizing itself. In contrast, requests will have higher QoE gain the more they wait, automatically boosting their priorities. (c) Finally, a request with long context length (l_i) will be preempted first, freeing enough GPU memory to potentially bring in *more than one* waiting requests.[3] This reduces the number of preemptions required to alleviate head-of-line blocking.

The whole procedure is given in Algorithm 1. The greedy packing algorithm offers time complexity $O(N \log N)$. We empirically show in Section 6.5 that this greedy solution can achieve performance comparable to the 3D DP algorithm while greatly reducing scheduling overhead.

Optimization #4: Preemption Cap. We have discussed that preemption is not free and can potentially degrade QoE. However, we can empirically and theoretically show that Andes commonly does not result in excessive preemptions/thrashing that may cause average QoE to degrade. Empirically, Andes consistently maintains an average preemption frequency below 1 per request, even under a high server load (§6.2.3). Theoretically, the number of preemptions needed to optimize the QoE of requests is contingent upon the excessive request load. Assume the serving system can handle r_0 requests per second and the actual request rate is $k \cdot r_0$ requests per second, where $k \geq 1$. Thus, there would be $(k-1) \cdot r_0$ requests whose QoE might be degraded due to the queuing delay. To mitigate this, we need roughly one preemption to accommodate each of these requests. Sometimes, a single preemption of a long request can allow multiple new requests to be served, which further reduces the number of preemptions needed. Therefore, the average preemption frequency needed is bounded by $k-1$, which is small as long as the load is not excessively high.

Nevertheless, in order to safeguard against thrashing that may happen in the worst case request pattern, Andes supports setting a cap P on the average number of preemptions a request can experience throughout its lifetime. Too high a P will not be able to act as a safeguard, whereas too small a P will prevent even absolutely necessary preemptions from happening. We find that setting $P = 1$, i.e., a request on average experiences at most one preemption during its lifetime, is a good default (Section 6.5).

5 Implementation

The two core elements of Andes are its QoE-aware scheduler and a client-side token buffer.

Server-Side QoE-Aware Scheduler. Andes's scheduling algorithm can work with any LLM serving system that supports continuous batching and at least one preemption mechanism (swapping or recomputation). We note that an LLM serving system that implements Paged Attention [25] is likely to also support at least one preemption mechanism to prevent the system from running out of memory. As a reference, we implemented Andes's scheduling algorithm on top of vLLM [25]. The scheduler only manages requests coming into the vLLM instance it is integrated with, assuming that cluster-level load balancing and fault tolerance are done separately.

[3]The overhead of preemption depends on how much memory was freed, not the number of requests. Therefore, for the same amount of memory freed from preemption, it's better to free a smaller number of requests.

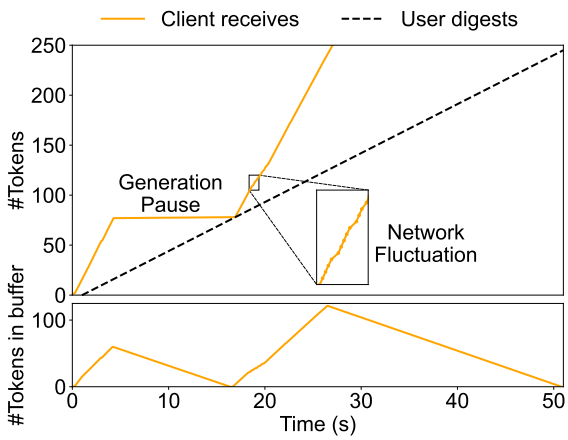


Figure 8. The client-side token buffer holds excess tokens sent from the server to absorb token generation fluctuations and paces token delivery based on the user's expected TDS.

Client-Side Token Buffer. The server sends tokens to the buffer as soon as they are generated, even if they were generated at a pace that exceeds the user's expected TDS. Then, the token buffer smooths out the token delivery timeline to pace tokens at the user's expected TDS. The token buffer can also naturally smooth out some fluctuations in network latency, for instance in crowded mobile networks. The buffer should be implemented appropriately depending on the destination of streaming – e.g., TypeScript for web frontend, Python for API use.

Figure 8 visualizes the token buffer in action. With an initial burst generation faster than the user's expected TDS, the buffer withholds excess tokens and paces token delivery, thus growing in size. The server is fully aware of the token buffer, and preempts the request to serve other requests. During this time, the buffer drains at a rate that matches the user's expected TDS. Finally, the server brings back the request and starts generating tokens again, and together with the token buffer, perfect QoE was achieved.

6 Evaluation

We evaluate the performance of Andes under different workloads. We demonstrate that:

1. Andes improves the average QoE up to 3.2× when the system experiences high/bursty load (§6.2.1).

Model size	13B	30B	66B	175B
GPUs	A100	4×A100	4×A100	4×A100
GPU Memory	80 GB	320 GB	320 GB	320 GB
Precision	FP16	FP16	FP16	8-bit [14]
Model Memory	26 GB	60 GB	132 GB	180 GB

Table 3. OPT model family and GPU specifications used.

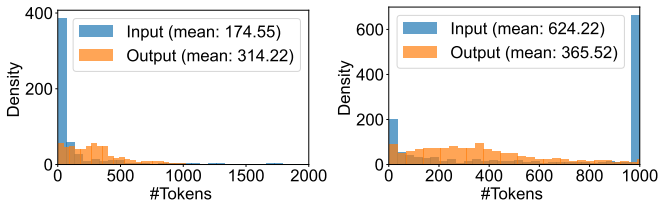
2. Andes can handle up to 1.6× higher request rates while preserving high QoE without additional resources, significantly reducing the serving cost(§6.2.2).
3. Andes maintains similar token generation throughput as the baseline, with a minor drop (≤ 10%) in throughput as the request rate increases (§6.2.3).
4. Andes significantly improves TTFT, while maintaining TDS above user expected speed (§6.3).
5. Andes outperforms the baselines across different workloads (§6.4) and setups (§6.5).

6.1 Experiment Setup

Model and Server Configurations. Following state-of-the-art LLM serving systems [25], we evaluate Andes using the OPT [51] series with 13B, 30B, 66B, and 175B parameters, with the 175B model employing INT8 quantization. We run all experiments on NVIDIA A100 GPUs in Chameleon Cloud [22], and use tensor parallelism to deploy the models, using the default configuration in vLLM [25]. We use swap as the preemption mechanism and set the CPU swap space to 240 GB in total. Detailed hardware specifications are provided in Table 3.

Workloads. We experiment on ShareGPT [45], a dataset that gathers conversations shared by users with ChatGPT [35], including multiple rounds of input prompt and output response. By concatenating multiple rounds of conversations into one input while limiting its length to 1k tokens to fit the model's maximum context length, and setting the final response as the output, we create the Multi-Round ShareGPT dataset for longer conversations. As shown in Figure 9, Multi-Round-ShareGPT has about 3× longer input than ShareGPT, while both datasets have similar output length distribution.

We generate request arrival traces using Poisson distribution with different arrival rates. The request's QoE requirement trace is created with different expected TTFT and TDS. TTFT is set to 1 second for all, while TDS is based on user reading speeds (Table 1), and is translated from words to tokens using the average word-to-token ratio for ChatGPT [38]. In real applications, QoE requirements should be set depending on the application's specific use case. For instance, reading speed (and thus expected TDS) may be measured using screen scrolling [18] or eye-tracking [3, 34]. Another potential use case is to introduce API price tiering,



(a) ShareGPT. **(b)** Multi-Round ShareGPT.

Figure 9. Input and output length distributions of datasets.

where a higher per-token price provides faster TDS, and API users can select the tier suitable for downstream digestion.

Baselines. We compare Andes with vLLM (version 0.2.7). vLLM uses first-come-first-serve (FCFS) scheduling policy by default. We implement another scheduling policy, Round-Robin (RR), atop vLLM for more informed comparison, which is designed to guarantee equal service to requests through cyclic request preemption. For RR, we set the service interval to 50 inference iterations, maximizing its QoE performance.

Metrics. We focus on the following metrics in evaluations:

- *Average QoE*: We set the threshold to 0.9 as the minimum acceptable average QoE. The QoE of 0.9 corresponds to a 5% delay in TTFT, a 10% slowdown in TDS, or something in the middle.
- *System capacity*: It measures the maximum request rate that the system can handle while maintaining an average QoE above the threshold.
- *System throughput*: It measures how many tokens the system generates per second.

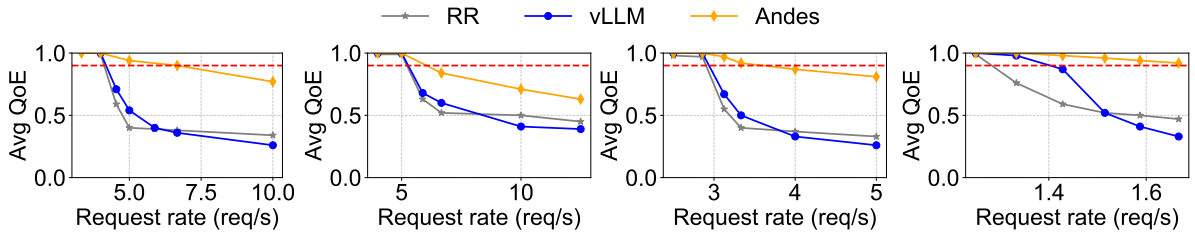
We also report normalized latency, which is used by vLLM[25] and Orca[50], in Appendix E.

6.2 End-to-End Experiments

In this section, we report the performance of Andes in terms of average QoE (§6.2.1), system capacity (§6.2.2), and system throughput (§6.2.3) under different setups.

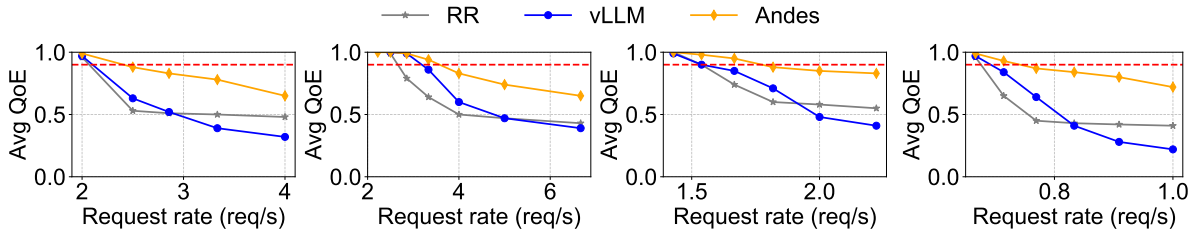
6.2.1 Improvement on Average QoE. We evaluate the performance of Andes on all four models and two datasets. Figure 10 and Figure 11 show the result on the ShareGPT dataset and Multi-Round ShareGPT dataset respectively. As the request rate increases, Andes maintains a high average QoE, outperforming the baseline whose average QoE sharply decreases. In other words, Andes can serve more concurrent requests without compromising user experience.

For ShareGPT dataset, Andes increases average QoE up to 3.1× at the same request rate, while maintaining an average QoE of 0.9, all with the same resources. For Multi-Round ShareGPT dataset, Andes improves average QoE up to 3.2×. For OPT-30B model, the improvement is less significant, as the model is less resource-constrained when compared to the OPT-66B model.



(a) OPT-13B **(b)** OPT-30B **(c)** OPT-66B **(d)** OPT-175B.

Figure 10. Average QoE for different request rates using the ShareGPT dataset.



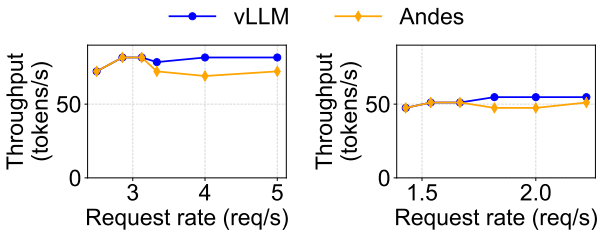
(a) OPT-13B. **(b)** OPT-30B. **(c)** OPT-66B. **(d)** OPT-175B.

Figure 11. Average QoE for different request rates using the Multi-Round ShareGPT dataset.

These improvements can be attributed to Andes's QoE-aware scheduling policy, which dynamically prioritizes resources for urgent requests that risk falling below their expected QoE, preempting those that have been sufficiently served. In contrast, under higher load, traditional FCFS scheduling policy suffers from head-of-line blocking, leading to significant queuing delay. Although the RR policy mitigates head-of-line blocking by preemptions, frequent preemptions introduce significant overhead and degrade the average QoE.

6.2.2 Improvement on Server Capacity. As shown in Figures 10 and 11, the horizontal dotted lines represent the average QoE threshold of 0.9. For ShareGPT dataset, Andes can manage 1.2×−1.6× higher request rate than vLLM while maintaining an average QoE above the threshold. Specifically, for the OPT-66B model, Andes can handle 1.25× higher request rate than vLLM, nearing the 1.38× theoretical improvement suggested in Section 2.3, showcasing Andes's ability to optimize resource allocation and average QoE effectively. For Multi-Round ShareGPT dataset, Andes can serve 1.1 × −1.3× higher request rate. Additionally, by serving higher request rates with the same resources, Andes effectively reduces the resource cost per request.

6.2.3 Impact of Andes on System Throughput. We report the token generation throughput and the preemption frequency of Andes on OPT-66B with both datasets, as shown in Figure 12 and Figure 13. In both datasets, Andes maintains the same token throughput as vLLM when the request rate is moderate, and experiences a minor drop (≤ 10%) in throughput as the request rate increases. This demonstrates that



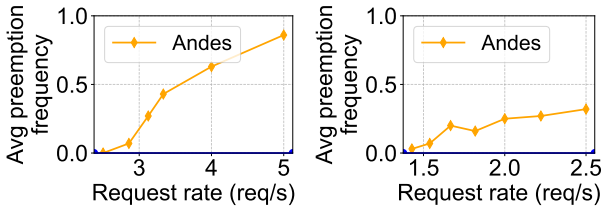
(a) ShareGPT. **(b)** Multi-Round ShareGPT.

Figure 12. Token generation throughput with OPT-66B under different request arrival rates.

Andes marginally impacts system throughput. The throughput decrease can be attributed to the overheads introduced by request preemption. Despite the active request scheduling, the frequency of preemptions per request remains low (≤ 0.5) under reasonable average QoE as shown in Figure 13, minimizing the impact of overheads on throughput; Despite the minor decrease in throughput, the up to 60% improvement in server capacity offered by Andes can compensate for this, effectively reducing the resource cost per request while maintaining a satisfactory user experience.

6.3 Breakdown Analysis

To understand Andes's performance in detail, we conducted a breakdown analysis focusing on QoE, time to first token (TTFT), and token delivery speed (TDS), as shown in Table 4. We report Andes's performance on OPT-66B and ShareGPT dataset with a request rate of 3.3, where Andes achieved an average QoE of 0.92. With these breakdown analyses, we can



(a) ShareGPT. **(b)** Multi-Round ShareGPT.

Figure 13. Preemption frequency with OPT-66B under different request arrival rates.

Metric	Percentile	Method	
		vLLM	Andes
QoE	10^{th}	0.05	0.77
	50^{th}	0.39	1.00
	90^{th}	1.00	1.00
TTFT (s)	10^{th}	0.33	0.35
	50^{th}	56.73	0.47
	90^{th}	144.95	0.66
TDS (tokens/s)	10^{th}	6.05	5.32
	50^{th}	6.45	5.44
	90^{th}	7.84	7.02

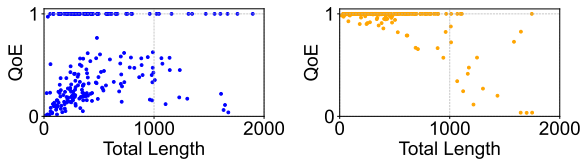
Table 4. Andes significantly improves QoE and TTFT, while maintaining TDS above user expected speed.

provide granular insights into individual user satisfaction under this level of QoE.

QoE distribution. Andes significantly improves the lower and median user experiences, with the 10th percentile rising from 0.05 to 0.77 and the 50th percentile achieving a perfect score of 1, compared to 0.39 in vLLM. In order to understand how Andes handles requests with different request lengths, we present a scatter plot of QoE across different total lengths as shown in Figure 14. We observe Andes slightly starves a small fraction of longer requests, as they consume more resources or take longer time to complete. In contrast, FCFS starves lots of shorter requests that are blocked by longer requests.

Token delivery timeline. Andes greatly enhances initial responsiveness, reducing median TTFT from 56.73 seconds in vLLM to just 0.47 seconds, and similarly improving the 90th percentile from 144.95 seconds to 0.66 seconds. This improved performance is attributed to Andes's QoE-aware scheduling, which effectively mitigates head-of-line blocking and reduces queuing delays.

Additionally, we analyze the percentile distribution of the average TDS observed by users, excluding TTFT. While Andes slightly slows the average TDS, it remains above the user's expected speed, ensuring balanced delivery that neither overwhelms nor starves users.



(a) vLLM. **(b)** Andes.

Figure 14. QoE distribution across different total lengths.

6.4 Robustness to Diverse Workloads

We evaluate the robustness of Andes under diverse settings including different hardware, arrival patterns, and QoE traces. We observed similar trends in diverse settings; therefore, we report our results with OPT-66B and ShareGPT.

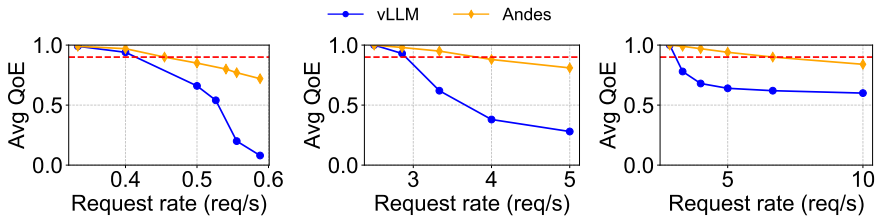
Hardware. We evaluate Andes on the NVIDIA A40 GPU with 46 GB RAM, as shown in Figure 15a. Andes improves average QoE up to 7× under a higher request rate and serves 1.1× higher request rate while maintaining an average QoE of 0.9. The reason for the smaller improvement on server capacity is that the A40 has a lower computational capability than the A100, leading to a slower average token generation speed. Consequently, the gap between the expected TDS and actual TDS on the A40 is smaller than on the A100, providing less opportunity for request scheduling and improving average QoE. However, as newer generations of GPUs are becoming more powerful in terms of computational capability, the potential improvement of Andes will be more significant.

Bursty Arrival Process. We use a Gamma arrival process with the same request rate and a coefficient of variation of 3 to simulate the burst arrival of user requests. Figure 15b indicates that under bursty workload, the average QoE for FCFS policy begins to decrease at a lower request rate compared to the Poisson arrival, due to increased queuing delays. In contrast, Andes sustains a high average QoE, achieving up to a 2.7× improvement on average QoE at the same request rate and serves 1.3× higher request rate, showing Andes's adaptability to bursty workload.

Different QoE Traces. Due to the unique QoE requirements of different applications, we evaluate Andes's performance under a voice chat QoE trace, with expected TTFT at 1 second and slower expected TDS adjusted according to the speaking speed outlined in Table 2. As shown in Figure 15c, both Andes and baseline achieve better average QoE even on higher request rates, attributed to the less strict TDS requirements. Nevertheless, Andes improves average QoE up to 1.25× and manages 2× request rate, which approaches the theoretical maximum improvement of 2× as discussed in Section 2.3.

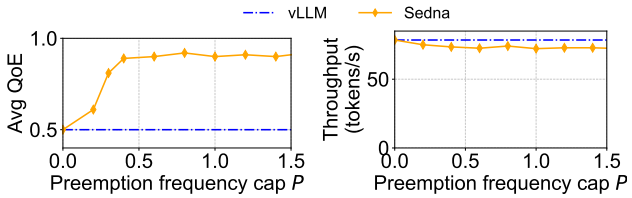
6.5 Sensitivity Analysis

All experiments in sensitivity analysis are conducted on OPT-66B with the ShareGPT dataset and a request rate of 3.3.



(a) NVIDIA A40. **(b)** Burst request arrival. **(c)** Voice chat QoE trace.

Figure 15. Robustness analysis on OPT-66B with ShareGPT dataset.



(a) Average QoE. **(b)** Throughput.

Figure 16. Tuning preemption frequency cap P.

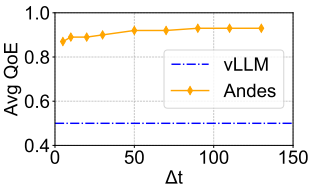


Figure 17. Tuning Δt.

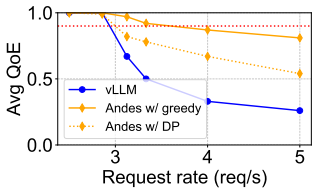


Figure 18. Different solver.

Preemption Frequency Cap P. Increasing preemption frequency cap P can lead to finer-grained scheduling, potentially enhancing average QoE, but at the cost of increased overhead and reduced throughput. Figure 16a shows the average QoE under different P. Improvements in QoE are observed as P increases up to 0.4 preemptions per request, stabilizing beyond this point. Conversely, Figure 16b illustrates a slight decrease in system throughput with increased P, stabilizing beyond 0.4 preemption per request. These observations suggest a trade-off between average QoE and system throughput, indicating the current setting of P nearly optimizes QoE while maintaining satisfactory throughput.

Prediction Timeframe Δt. We evaluate how different Δt influences average QoE to understand its effect on system performance. Figure 17 illustrates that the average QoE remains roughly consistent for Δt values greater than 50, and significantly outperforms the baselines, indicating that Andes is not sensitive to the setting of Δt.

Different Knapsack Solution. We compare the performance of Andes with different knapsack solutions between greedy and dynamic programming (DP). Figure 18 shows that the greedy consistently surpasses the DP solution, while both solutions outperform the baselines. The lower performance of the DP is due to its substantial computational overhead, which delays the inference process and degrades the average QoE. This suggests that the greedy approach is a more practical and efficient solution for Andes.

7 Related Work

General Model Serving Systems. A variety of model serving systems have emerged, ranging from general-purpose, production-level frameworks like TensorFlow Serving [33] and NVIDIA Triton [31] to specialized systems such as Clipper [11], which sets application-level SLOs. Recent systems including Nexus[42], DeepRecSys [17], Clockwork [16], INFaaS [40], SuperServe [24] and AlpaServe [26] have introduced features like serving pipelines, hardware platform diversity, advanced scheduling, dynamic model selection, and model parallelism to boost resource efficiency. However, these general systems neglect the unique characteristics of LLM inference, leaving potential avenues for optimization.

LLM Serving Systems. Numerous model serving systems are proposed to address the unique challenges of LLMs. Orca [50] introduced an iteration-level scheduling policy to enhance the throughput of batching inference, and vLLM [25] developed a PagedAttention to reduce the memory usage of LLMs. Splitwise [37], DistServe [52], TetriInfer [19] and Sarathi-Serve [1, 2] optimize the computation of prefill and decode phases through disaggregating or merging them. Some other systems focus on GPU kernel optimization and kernel fusion[5, 12, 32], model parallelism [5, 39], batching algorithm [13, 43, 50], KV-cache management [27, 28, 44] and parameter-sharing [53]. However, these systems focus on optimizing aggregated server-side performance and simply adopt a FCFS scheduling policy, which fail to address the queuing delay problem under higher request load. Finally, shortest remaining processing time [41] is a preemptive scheduling policy, but it does not consider the QoE of individual requests and requires knowledge of the response length of requests. To the best of our knowledge, Andes is the first to define and optimize QoE of text streaming services.

Video Streaming and QoE. The concept of text streaming draws inspiration from video streaming but encounters unique challenges and has a different QoE definition. While video streaming services are primarily limited by network bandwidth and latency [7], text streaming services are mainly constrained on computational resources [48]. Additionally, the QoE in video streaming is often measured by metrics like buffering ratio, resolution stability, and playback smoothness [7], while the QoE in text streaming primarily considers the token delivery timelines (TDT).

8 Conclusion

In this paper, we define and optimize the Quality-of-Experience (QoE) for text streaming services, a critical aspect often overlooked by existing serving systems. We propose a QoE-aware LLM serving system, Andes, which is able to serve more concurrent requests while meeting their QoE requirements, significantly reducing the cost per request. We demonstrate the effectiveness of Andes through extensive experiments on various real-world datasets and LLMs, showing that Andes can handle up to 1.6× higher request rate while preserving high QoE, or enhance QoE by up to 3.2× without additional resource expenditure.

References

[1] 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. *arXiv preprint arXiv:2403.02310*, 2024.

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[3] Seoyoung Ahn, Conor Kelton, Aruna Balasubramanian, and Gregory Zelinsky. Towards predicting reading comprehension from gaze behavior. In *ETRA (Symposium on Eye Tracking Research and Applications)*, 2020.

[4] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. 2022.

[6] Daniel An. Find out how you stack up to new industry benchmarks for mobile page speed. In *Google Research Blog*, 2018.

[7] Nabajeet Barman and Maria G Martini. Qoe modeling for http adaptive video streaming–a survey and open challenges. *Ieee Access*, 7, 2019.

[8] Dom Barnard. Average speaking rate and words per minute. https://virtualspeech.com/blog/average-speaking-rate-words-per-minute#:~:text=According%20to%20the%20National%20Center, speech%20rates%20for%20different%20activities.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[10] Marc Brysbaert. How many words do we read per minute? a review and meta-analysis of reading rate. *Journal of memory and language*, 2019.

[11] Daniel Crankshaw, Xin Wang, Guilio Zhou, Joseph E. Franklin, Michael J. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.

[13] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. 2021.

[14] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training compression for generative pretrained transformers. In *ICLR*, 2023.

[15] Grand View Research. Large language model (llm) market size, share & trends analysis report by component, by application, by enterprise size, by end-use, by region, and segment forecasts, 2023 - 2030. Grand View Research, 2023.

[16] 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 *OSDI*, 2020.

[17] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *ISCA*. IEEE, 2020.

[18] Hannah Harvey and Robin Walker. Reading comprehension and its relationship with working memory capacity when reading horizontally scrolling text. *Quarterly Journal of Experimental Psychology*, 71(9):1887–1897, 2018.

[19] 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.

[20] HuggingFace. Text generation inference. https://github.com/huggingface/text-generation-inference.

[21] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[22] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *ATC*, 2020.

[23] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer Berlin Heidelberg, 2004.

[24] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. Superserve: Fine-grained inference serving for unpredictable workloads. *arXiv preprint arXiv:2312.16733*, 2023.

[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. In *SOSP*, 2023.

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, 2023.

[27] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[28] 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. *NeurIPS*, 36, 2024.

[29] Viraj Mahajan. 100+ incredible ChatGPT statistics & facts in 2024. https://www.notta.ai/en/blog/chatgpt-statistics.

[30] NVIDIA. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[31] NVIDIA. Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

[32] NVIDIA. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

[33] 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. *arXiv preprint arXiv:1712.06139*, 2017.

[34] Kristien Ooms, Arzu Coltekin, Philippe De Maeyer, Lien Dupont, Sara Fabrikant, Annelies Incoul, Matthias Kuhn, Hendrik Slabbinck, Pieter Vansteenkiste, and Lise Van der Haegen. Combining user logging with eye tracking for interactive and dynamic applications. *Behavior research methods*, 47:977–993, 2015.

[35] OpenAI. ChatGPT. https://chat.openai.com.

[36] Tara Parachuk. Speaking rates comparison table. https://www.voices.com/blog/languages-in-usa-speaking-rates-per-minute/#speaking-rates-comparison-table.

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[38] Aleksandar Petrov, Emanuele La Malfa, Philip Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages. *Advances in Neural Information Processing Systems*, 2024.

[39] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, James Devlin, Jacob ans Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *MLSys*, 2023.

[40] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *ATC*, 2021.

[41] Linus Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 1968.

[42] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *SOSP*, 2019.

[43] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. 2024.

[44] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. D\'ej\avu: Kv-cache streaming for fast, fault-tolerant generative llm serving. 2024.

[45] ShareGPT Team. ShareGPT. https://sharegpt.com/.

[46] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[48] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, et al. Efficient large language models: A survey. *Transactions on Machine Learning Research*, 2024.

[49] Wikepedia. Weak NP-completeness. https://en.wikipedia.org/wiki/Weak_NP-completeness.

[50] 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 *OSDI*, 2022.

[51] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[52] 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. In *OSDI*, 2024.

[53] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. Pets: A unified framework for parameter-efficient transformers serving. In *ATC*, 2022.

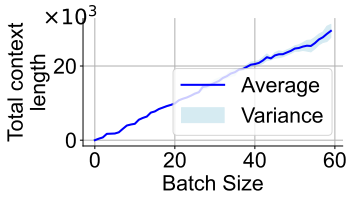


Figure 19. Total context length distribution under different batch sizes using the ShareGPT dataset and OPT 66B. Plotting is stopped once GPU memory saturation is reached.

A Alternative Scheduling Objectives

In Section 4.1, we presented Andes in terms of maximizing the average QoE across all requests. However, text streaming services can have different quality goals under various deployment circumstances. More importantly, our defined QoE metric and proposed solution can be seamlessly adapted to different QoE objectives. In this section, we explore alternative scheduling objectives.

Maximizing the Minimum QoE. To maximize the minimum QoE across all requests (i.e., max-min QoE), the gain (item value in knapsack) of request i can be formulated as:

$$\max(Q_{\text{min}} - Q_{\text{wait},i}, 0), \tag{6}$$

where Q_{min} is the minimum QoE across all requests. This function prioritizes requests that, if not served within Δt, would further degrade the minimum QoE. By prioritizing these urgent requests, the overall QoE floor can be lifted, ensuring a more uniformly satisfying user experience.

Maximizing the Number of Requests with Perfect QoE. To optimize the number of requests that achieve perfect QoE, the gain (item value in knapsack) of request i can be formulated as:

$$[\mathbb{1}(Q_{\text{serve, i}} = 1) - \mathbb{1}(Q_{\text{wait, i}} = 1)] \cdot \mathbb{1}(Q_{\text{current},i} = 1), \tag{7}$$

where $\mathbb{1}(\cdot)$ is 1 if the given condition is true and 0 otherwise, and $Q_{\text{current},i}$ is the request's current QoE. The intuition behind this approach is that (1) there is no point in serving a request whose QoE is not perfect at the moment, and (2) if a request with currently perfect QoE will degrade QoE if not served for Δt, the request must be prioritized.

B Modeling Token Generation Latency

In order to solve our knapsack formulation in Section 4.1, we need to be able to anticipate the QoE of a request after Δt if served (i.e., $Q_{\text{serve},i}$), which requires us to model token generation latency as a function of system state. The latency to run one LLM decoding iteration for a given LLM architecture is known to depend on batch size and the total number of tokens in the batch.

Figure 19 shows the relationship between the batch size and the total number of tokens across all requests in the batch (i.e., total context length). We use the ShareGPT dataset and OPT-66B model with request rate 2.5 req/s on A100 using vLLM to generate this data. It can be seen that batch size and total context length are nearly perfectly correlated, with Pearson correlation coefficient being 0.997. In other words, while fixing the batch size and varying total context length will change token generation latency, in a live serving system, a specific batch size typically leads to a very consistent total context length. Therefore, we can drop total context length and model token generation latency simply as a function of batch size B.

Algorithm 2 Dynamic programming solution to Equation 4

Input:
Number of requests N and KV cache capacity M
Request context length array $l[N]$
Request QoE gain array $q[N]$
Target batch size B
Output: Solution array $x[N]$.

Initialize $dp[N+1][B+1][M+1]$ with $-\infty$
Initialize $choice[N+1][B+1][M+1]$ with 0
$dp[0][0][0] = 0$
for $i = 1$ to N **do**
  for $b = 0$ to $\min(i, B)$ **do**
    for $m = 0$ to M **do**
      if $dp[i][b][m] < dp[i-1][b][m]$ **then**
        ▷ Request i is not served.
        $dp[i][b][m] = dp[i-1][b][m]$
        $choice[i][b][m] = 0$
      if $b \geq 1$ & $m \geq l[i]$ **then**
        if $dp[i-1][b-1][m-l[i]] + q[i] > dp[i][b][m]$ **then**
          ▷ Request i is served.
          $dp[i][b][m] = dp[i-1][b-1][m-l[i]] + q[i]$
          $choice[i][b][m] = 1$
$Q_{\text{max}} = \max(dp[N][B][:])$
$m_{\text{current}} =$ Index of Q_{max} in $dp[N][B]$
$b_{\text{current}} = B$
Initialize $x[N+1]$ with zeros
for $i = N$ **downto** 1 **do**
  $x[i] = choice[i][b_{\text{current}}][m_{\text{current}}]$
  if $x[i] == 1$ **then**
    $m_{\text{current}} = m_{\text{current}} - l[i]$
    $b_{\text{current}} = b_{\text{current}} - 1$
return $x[1:]$

C Dynamic Programming Solution

In Algorithm 2, we give a 3D dynamic programming solution to Equation 4. The time complexity of the algorithm is $O(M \cdot N^2)$ as the largest batch size B is N in the worst case, and the problem needs to be solved for all feasible batch sizes B to find the optimal set of requests to serve. We note for clarity that our knapsack problem is *weakly NP-Hard* and the 3D DP

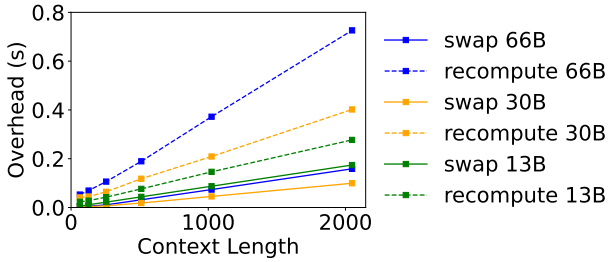
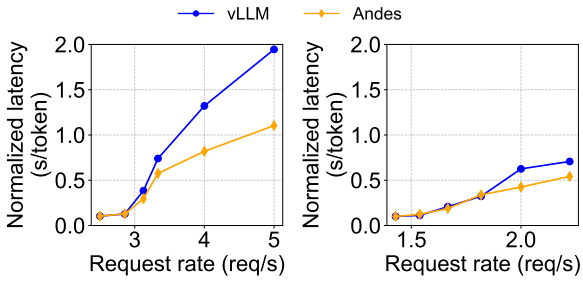


Figure 20. Swapping and recomputation overhead on A100.



(a) ShareGPT. **(b)** Multi-Round ShareGPT.

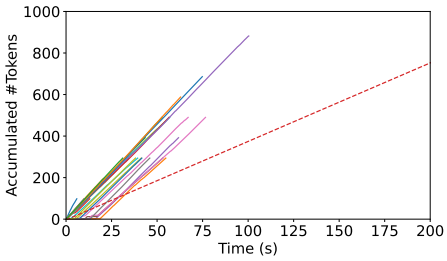
Figure 21. Normalized token latency with OPT-66B under different request arrival rate.

algorithm is *not* polynomial time with respect to problem size (number of bits required to represent the problem). That is, when the problem size (number of bits) is scaled in terms of the number of requests N by adding more requests, runtime grows quadratically. However, when the problem is scaled in terms of available memory M by increasing the number of bits needed to represent M, the value of M and thus algorithm runtime grows exponentially. Therefore, the solution runs in *pseudo-polynomial* time, which is effectively exponential time. For more details on weak NP-Hardness and pseudo-polynomial runtime, we direct the reader to [49].

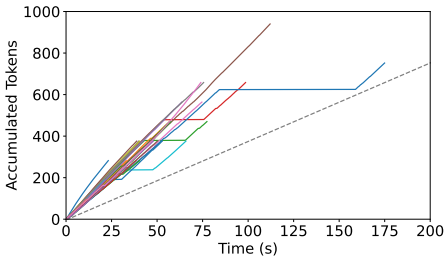
D Swapping and Recomputation Overhead

Figure 20 shows the swapping and recomputation overhead of OPT models on A100-PCIe. We find that swapping consistently leads to lower overhead on these GPUs. Our results complement that of vLLM [25] Section 7.3 in that our node configuration has NVLink connecting GPUs 0 and 1, and 2 and 3. This led to increased tensor parallelism communication overhead compared to nodes with full-mesh NVLink connectivity, increasing the overhead of recomputation.

Swapping and recomputation are semantically identical operations but stress different parts of the system. Therefore, users should measure the overhead of preemption techniques on their production environment and select the one with lower overhead.



(a) FCFS.



(b) Andes.

Figure 22. TDT Visualization.

E Analysis on normalized latency

vLLM [25] and Orca [50] use normalized latency to indicate system throughput, which is defined as the average of every request's end-to-end latency divided by its output length.

We measure the normalized latency of Andes on OPT-66B with two datasets, as shown in Figure 21. In each scenario, Andes exhibits a comparable normalized latency at lower request rates—where maintaining a perfect QoE is feasible for all scheduling policies—while showing significantly lower normalized latency at higher request rates in contrast to the baselines. This is attributed to Andes's QoE-aware scheduling, which promptly serves incoming requests to meet expected TTFT, preventing shorter-output requests from being delayed by earlier, longer-output ones.

F Visualization of Improved TDT

To illustrate how Andes effectively manages TDS, we randomly sample 3.3% of requests who have the same QoE requirement and plot the accumulated tokens over time, as shown in Figure 22. We offset the starting points of all requests to uniformly track the progression of token generation. In the figure, the dashed line represents the expected Token Delivery Timeline (TDT), while the colored lines indicate the actual accumulation of tokens for each request. The figure demonstrates that nearly all requests in Andes meet the expected TDT, with most colored lines appearing above the dashed line, indicating a fluent token delivery.

In contrast, the FCFS policy fails to align with the expected TDT, primarily due to the head-of-line blocking problem, which prevents achieving the expected TTFT.