

Stable Token-level Scheduling for Serverless LLM Inference

Hardi Ashishbhai Desai, Balaji N K, Sourav Kanti Addya

Cloud & Smart System Services Lab

Department of Computer Science and Engineering

National Institute of Technology Karnataka, Surathkal, India

Emails: {hardidesai.242cs015, balajink.252is009, souravkaddya}@nitk.edu.in

Abstract—Serverless inference is emerging as an efficient deployment model for Large Language Models (LLMs) due to its ability to elastically scale resources, improve GPU utilization through resource sharing, and reduce infrastructure costs. However, existing approaches primarily optimize time to first token (TTFT) while overlooking inter-token latency (ITL), a key determinant of user experience during response generation. We observe that LLM inference comprises a computation-intensive prefill stage and a bandwidth-bound decode stage, where admitting new requests can disrupt ongoing decoding and increase latency. To address this, we propose Token Window Pacing (TWP), a closed-loop scheduling framework that leverages runtime feedback from ITL, queue depth, and workload characteristics to regulate request admission at the token level instead of the batch level. TWP dynamically adjusts a self-regulating token window and incorporates chunk-based prefill to minimize interference from incoming requests. Experiments on realistic serverless workloads show that TWP delivers stable TTFT, significantly reduces latency spikes, lowers P99 tail latency, minimizes ITL variation, and improves throughput by up to 45% over ServerlessLLM and 58% over conventional schedulers. These results demonstrate that feedback-driven token-level scheduling enables stable, high-throughput serverless LLM inference without modifying the underlying model.

Index Terms—Token Window Pacing, Serverless Scheduling, Inter-Token Latency, Time-To-First-Token, Dynamic Token Scheduling

I. INTRODUCTION

Large language models (LLMs) are becoming popular for modern interactive applications such as conversational systems, programming assistants, and search engines, where strict latency requirements make responsiveness a key component of user experience. To reduce the high cost of provisioning dedicated GPUs, many systems now rely on serverless inference, in which model instances are dynamically instantiated on demand and multiple tenants share a pooled GPU infrastructure under a pay-per-use model. It is cost-effective in that it allows service providers to optimize GPU usage via load sharing without incurring costs related to reserved capacity by end users. The downside, however, is that the practice introduces nontrivial scheduling problems that have yet to be adequately addressed in the LLM-serving literature.

A. How LLM Inference Works

Figure 1 illustrates the execution pipeline of LLM inference. Each request undergoes two phases: *prefill* and *decode*. During

the *prefill* phase, the model processes the entire input prompt in a single forward pass. This step initializes the Key-Value Cache (KV-cache), which stores the attention state associated with prompt tokens and remains resident in GPU memory throughout the lifetime of the request. *Prefill* executes once per request and its cost scales with prompt length.

After *prefill*, the request enters the *decode* phase, where output tokens are generated iteratively, one token at a time. Decode execution is synchronous across all active requests: the system cannot advance to the next decode step until the current step completes for every request in the batch. Consequently, the latency of a decode iteration depends on the aggregate workload of the active batch, making scheduling decisions highly consequential. Admitting a new request introduces an additional prefill operation that competes with ongoing decode execution for GPU resources. The resulting interference delays token generation for all active requests, increasing inter-token latency (ITL) system-wide.

B. The Admission Control Gap

Serverless workloads are inherently bursty, with production traces showing that invocation rates can fluctuate by multiple orders of magnitude over short time intervals, and that a small fraction of functions account for a disproportionate share of total traffic [1]. Such variability is particularly challenging in interactive applications, where request-rate spikes are common and can coincide across many tenants. In addition, cold-start rates exceeding 25% in a large fraction of deployments further amplify instability in request arrival patterns [2]. Under such circumstances, a scheduler without knowledge of the internal state of the decode engine may over-aggressively accept incoming requests during bursts, thus saturating the GPU above its efficient operating region. The result is poor efficiency in terms of serving requests, with an increase in inter-token latency (ITL), which is defined as the latency between two subsequent tokens sent to a user.

This becomes a self-reinforcing issue because higher ITL leads to slower token creation rates. This leads to increased time required for the batch to be emptied. The recovery time becomes extended, and all the users within that session experience poor stream throughput during this entire process.

A rich body of work has improved individual aspects of LLM serving. Continuous batching [3] eliminates idle time be-

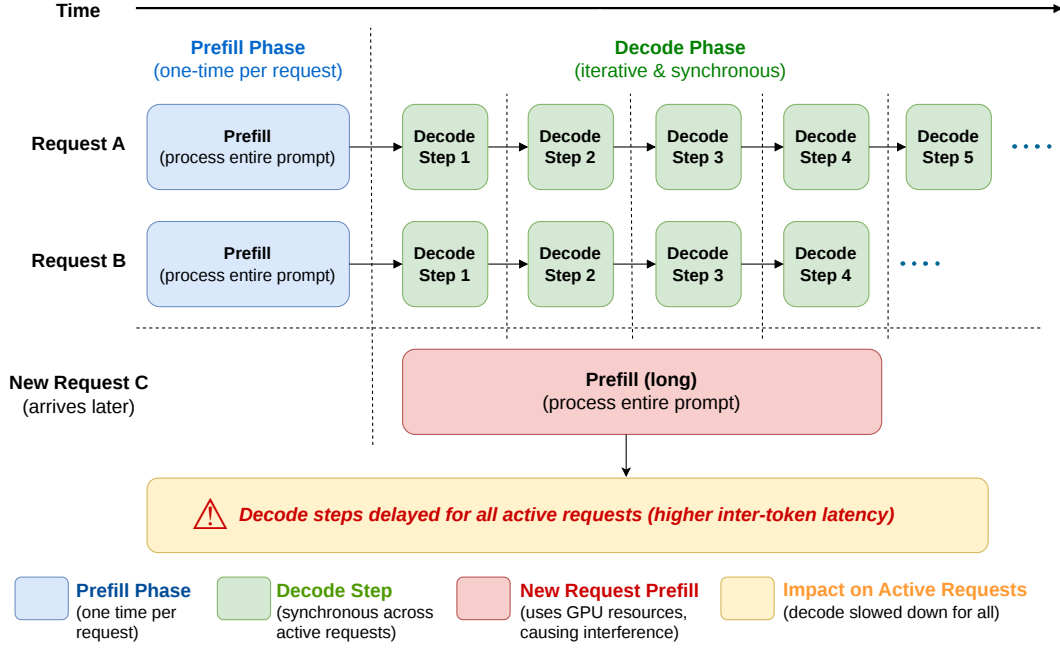


Fig. 1. LLM inference pipeline showing the two-phase execution model.

tween requests. Memory-efficient KV-cache management [4] makes GPU memory use more predictable. Chunked prefill [5] reduces the latency spike caused by large prompt ingestion. ServerlessLLM [2] dramatically cuts cold-start overhead through multi-tier checkpoint loading. A Generative Scheduling framework [6] further advances phase-aware scheduling by jointly optimizing resource allocation for the prefill and decode stages across heterogeneous hardware. Each of these is a genuine contribution to the serving stack. Yet none of them closes the feedback loop between what the decode engine is currently experiencing and how many new requests should be admitted. The admission gate remains open-loop: decisions are made at arrival time, not in response to observed congestion.

C. Our Approach

We propose TWP a scheduler that completes this feedback loop at token granularity. The core insight is that ITL, obtained at the end of every decode stage, directly and with no overhead indicates GPU decode pipeline congestion. ITL is a better indicator compared to queue depth, which can keep growing even when ITL values are low, and also more useful than KV-cache occupancy, which fails to reflect latency cost. By combining ITL and queue depth, as well as KV-cache occupancy in a proportional feedback controller, TWP controls a *dynamic token window* in order to ensure proactive overload prevention.

Contributions:

- i. A **token-granularity, feedback-driven admission framework** for serverless LLM inference, grounded in a formal instability result showing why unconstrained admission fails under burst load (§IV-A).

- ii. An **ITL proof-of-concept** (§III) that characterises ITL as a congestion signal, establishes its theoretical properties, and shows why it is the correct optimisation target for joint TTFT and decode-stability goals.
- iii. The **TWP scheduler** (§V): a two-guard admission gate (token budget and KV-cache headroom), a starvation-safe composite ranking policy, and chunked prefill staging with principled parameter choices.
- iv. An **evaluation** (§VI) against standard baselines on real workloads in a shared serverless container pool.

II. BACKGROUND & PRIOR WORKS

This section provides the background necessary to motivate the design of TWP. We first examine the cost and latency characteristics of LLM inference and the admission challenges that arise in shared serving environments. We then place TWP in the context of prior work on serverless scheduling and LLM serving, highlighting the gap addressed by our approach.

A. Cost Structure, Latency Metrics, and Admission Hazards

The introduction established that LLM inference has two phases and that the decode phase is synchronous across the batch. Here we look at the cost structure of each phase in more detail, define the two latency metrics that TWP optimises, and identify two distinct admission hazards that motivate the design.

Prefill cost: Prefill is expensive and prompt-length-dependent: every input token attends over all preceding tokens, so the compute cost scales quadratically with prompt length. Its output — the *KV-cache* — occupies GPU memory proportional to prompt length from the moment of admission. A long

prompt therefore consumes both a large burst of compute and a lasting block of memory the instant it enters the system. This is relevant for admission because both resources are shared with the ongoing decode work of already-active requests.

Decode cost: Each decode step reads the full KV-cache of every active request and writes one new entry per request. The dominant cost is memory bandwidth: loading KV-cache tensors from GPU high-bandwidth memory (HBM). Up to a saturation batch size B_0 , the GPU’s memory bus amortises this cost across requests and step time stays flat — batching is effectively free in this regime. Beyond B_0 , attention computation becomes the bottleneck and step time rises with each additional request, raising ITL for every active user simultaneously. This piecewise cost structure defines the operating region TWP aims to stay within.

TTFT and ITL: In a serverless deployment, TTFT — the delay from request submission to the first output token — has three components: a *cold-start* penalty if no warm container is available (the model must be loaded from storage before inference can begin), followed by the prefill phase, followed by the first decode step. ITL governs the streaming experience thereafter. Formally:

$$\text{TTFT}_i = \underbrace{cs_i}_{\text{cold-start overhead}} + \underbrace{\sum_{k=t_0}^{t_0 + \lceil p_i/C \rceil} \ell_k}_{\text{chunked prefill in decode steps}}, \quad (1)$$

where cs_i is the cold-start delay (zero for a warm container), t_0 is the admission step, and C is the prefill chunk size. When prefill is staged across decode steps — as TWP does — its cost is paid entirely in units of ITL. Minimising ITL therefore simultaneously reduces TTFT and improves streaming quality, which is why it is the right primary control target.

The noisy-neighbour penalty: A second admission hazard arises at the moment a new request enters the system. Prefill issues a large, bursty memory operation: KV-cache entries for every input token must be written to HBM in the same step that the ongoing decode kernels are reading their own caches. This contention on the shared memory bus transiently raises ITL for all currently active requests — an effect that scales with prompt length and is entirely invisible to a scheduler that makes decisions only at arrival time. We call this the *noisy-neighbour penalty*. A long prompt admitted in one shot can temporarily double active users’ ITL. Spreading ingestion across multiple steps in fixed-size chunks bounds this penalty per step to a small fraction of the full-prompt cost, at a modest TTFT cost we derive in §iii.

B. The Serverless Scheduling Landscape

We survey the existing work relevant to TWP, organised around the distinct problems each body of work addresses, and identify the gap that remains open.

Cold-start reduction: Fu *et al.* [2] introduce Serverless-LLM, which optimizes checkpoint loading across DRAM and NVMe memory tiers to reduce cold-start latency by a factor

of 10 to 200 times. It also implements live migration that only transfers token states between servers, making serverless LLM serving practically feasible. Building on this, Cui *et al.* [7] present TIDAL, a solution that enhances Function as a Service (FaaS) by overlapping model loading with early inference through weight-centric tracing and adaptive state forking. However, they note that CPU-bound initialization remains unaccelerated. Yu *et al.* [8] introduce λ Scale, whose λ Pipe binomial multicast pipeline constructs execution graphs across nodes during loading; the system is currently limited to single-node memory capacity. Zhu *et al.* [9] reduce repeated loading by using Tangram, which shares tensor-level parameters within a unified GPU memory pool and employs affinity-aware scheduling. This approach incurs a slight throughput overhead due to on-demand key-value cache allocation. Lou *et al.* [10] optimize the process by overlapping model loading with container initialization through network-aware placement. Sui *et al.* [11] demonstrate that pre-warming alone is insufficient and propose opportunistic pre-loading when there is idle capacity. Additionally, Lou *et al.* [12] introduce universal GPU workers capable of pre-warming multiple models simultaneously. All of these strategies address the startup component cs_i in Equation (1). Once a container is warmed up, the main challenge becomes regulating the decoding batch to maintain a low inference turn-around time (ITL).

Continuous batching and memory management:

Orca [3] introduces continuous batching, making the decode step the natural scheduling unit and eliminating padding waste—directly motivating TWP’s step-synchronous control. vLLM [4] adds PagedAttention, managing KV-cache in fixed-size pages to eliminate fragmentation; the resulting per-step occupancy measurement underpins TWP’s second admission guard.

Chunked prefill and phase separation:

SARATHI [13] and Sarathi-Serve [5] split prefill into fixed-size chunks merged with decode steps, substantially reducing the noisy-neighbour penalty. These establish the chunked admission mechanism TWP adopts, but neither connects observed ITL back to admission rate. Splitwise [14] separates prefill and decode onto dedicated hardware pools; TWP achieves phase-awareness within a single shared pool.

SLO-aware and token-level scheduling:

Bari *et al.* [15] use deadline-driven criticality scheduling for TTFT and time-between-token SLOs, but require known deadline targets and target single-instance deployments. TokenScale [16] proposes Token Velocity as a predictive autoscaling metric, reinforcing that token-level signals outperform request-level metrics for capacity decisions. LAPS-SD [17] combines Least-Attained-Service with SJF for speculative decoding, highlighting that output-length uncertainty is a fundamental scheduling challenge shared with TWP. Han *et al.* [18] propose TALE, which dynamically adjusts chain-of-thought token budgets via zero-shot prompting or post-training internalisation; though targeting reasoning efficiency rather than serving infrastructure, it reinforces that token counts carry actionable workload-

intensity signals.

Serverless scheduling, LoRA serving, and large-scale systems: Thompson *et al.* [19] derive competitive online scheduling algorithms for serverless systems with setup delays, staying within a $(1 + \epsilon)$ cost factor of optimal, but assume uniform costs and make no distinction between prefill and decode phases. Ni *et al.* [20] address multi-adapter cold-start with Predictive-LoRA, prefetching hot LoRA adapters via an LSTM predictor and eliminating fragmentation through page-based memory management, though accuracy degrades under dynamic patterns. Sui *et al.* [21] propose ServerlessLoRA, sharing model backbones across isolated functions via CUDA IPC with contention-aware batching and dynamic GPU offloading, at the cost of duplicated CUDA context overhead. Hu *et al.* [22] present DeepServe, a production serverless platform with a microkernel serving engine and NPU-fork for fast weight transfer across nodes; specialised hardware interconnects are required and true zero-instance cold starts remain unsupported.

Memory-constrained inference and adaptive batching: Alizadeh *et al.* [23] load only sparsity-selected weights from flash into DRAM using a low-rank predictor, with windowing and row-column bundling for throughput; the work reinforces that KV-cache and weight residency must be managed jointly, a principle TWP operationalises in its per-step memory guard. Ali *et al.* [24] propose BATCH, tying batch size to profiled latency targets for non-autoregressive serverless inference; the insight that batching decisions should be grounded in observed latency aligns with TWP, though BATCH uses static thresholds and ignores KV-cache dynamics and the prefill–decode distinction.

The open gap: No existing system effectively closes the feedback loop between the observed decode-phase Inference Time Latency (ITL) and the request admission gate in a shared serverless Large Language Model (LLM) pool. DynamicBatch employs a static batch-size limit without mechanisms to prevent saturation before this limit is reached. ServerlessLLM’s rapid loading unintentionally leads to batch saturation by accepting requests faster than the decode phase can handle them. The root of the problem lies in the structure of the system: all existing schedulers make admission decisions at the moment a request arrives, without considering any information about the status of the current batch from the decode loop. Our system, TWP, addresses this issue by making admission decisions during the decode step, using the ITL measured in the previous step to inform these choices.

III. ITL AS A CONGESTION SIGNAL

We establish the formal and empirical case for ITL as the primary feedback signal in TWP. All characterisation in this section is derived from measurements on our experimental platform.

A. Definition and User-Visible Meaning

Definition 1 (Inter-Token Latency). *Let t_k be the wall-clock time at which decode step k completes. The inter-token latency*

at step k is $\ell_k = t_k - t_{k-1}$.

Because the decode loop is synchronous, every active request experiences the same ℓ_k at each step. From a user’s perspective, ℓ_k is the gap between successive tokens in the response stream. A stable, low ℓ produces smooth streaming; a high or volatile ℓ produces visible stuttering regardless of how fast the total response eventually arrives.

B. The ITL–Batch-Size Relationship

The duration of a decode step is governed by either KV-cache transfers from GPU high-bandwidth memory (HBM) or attention computation. For small batches, memory transfers dominate and the GPU can load KV-caches for multiple requests without increasing total transfer time. Consequently, step duration remains constant at a minimum value $\ell_{\min}$ until the batch size reaches a saturation point B_0 .

Beyond B_0 , attention computation becomes the bottleneck. Each additional request increases the computation required to process its KV-cache, causing step duration to grow approximately linearly with batch size. The resulting ITL model is

$$\ell(B_t) = \begin{cases} \ell_{\min}, & B_t \leq B_0, \\ \ell_{\min} + \alpha(B_t - B_0), & B_t > B_0. \end{cases} \quad (2)$$

where B_0 denotes the transition point between bandwidth-bound and compute-bound execution, and α captures the marginal increase in decode-step latency per additional request. These parameters are obtained through a one-time offline profiling sweep that measures step duration across different batch sizes. Figure 2 illustrates the characteristic behaviour and the inflection at B_0 .

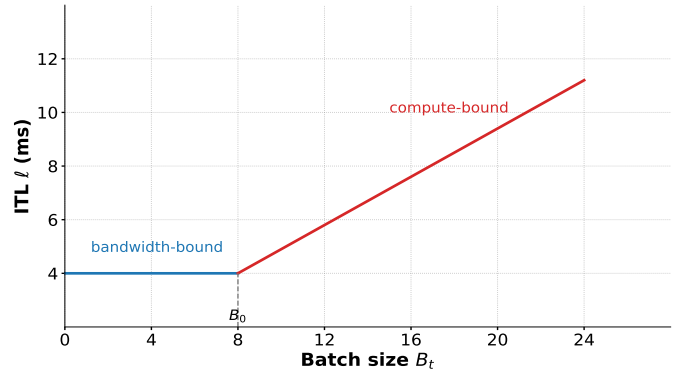


Fig. 2. TTFT comparison across different scheduling policies.

C. Instability Without Admission Control

At steady state, tokens arrive at rate $\lambda \bar{L}$ and are processed at rate $B^*/\ell(B^*)$. For $B^* > B_0$, the service rate satisfies $B^*/\ell(B^*) \rightarrow 1/\alpha$ as $B^* \rightarrow \infty$. Thus, the maximum achievable token processing rate is bounded by $1/\alpha$.

When $\lambda \bar{L} > 1/\alpha$, the token arrival rate exceeds the maximum service rate. The backlog therefore grows continuously, implying that no finite steady-state batch size exists. In the

absence of an explicit bound on token load, B^* diverges without limit.

Even when $\lambda\bar{L} < 1/\alpha$, transient bursts can temporarily raise the effective arrival rate above $1/\alpha$. During such periods, the system enters the compute-bound regime, causing the active batch size to grow. Recovery occurs only after the excess queued requests are processed, which may require many decoding steps. As a result, all active users experience elevated ITL throughout the recovery period. Admission control through a token window prevents unbounded batch growth and limits the duration and severity of these overload episodes.

D. Why ITL Dominates Alternative Signals

Queue depth reflects demand but not congestion. The queue can grow while $B_t < B_0$ and ITL remains at $\ell_{\min}$. Throttling on queue depth alone would delay admission during periods where the system has available headroom.

KV-cache occupancy reflects memory pressure but not compute pressure. A batch of many short-output requests may have high occupancy while still residing comfortably in the bandwidth-bound regime. Memory occupancy is a necessary guard against hard out-of-memory failures but is not sufficient as the sole control signal.

ITL responds within one decode step to any change in B_t . It requires no additional measurement infrastructure: the step wall-clock time is already tracked by the serving engine as a standard bookkeeping value. It directly encodes the cost being imposed on every active user by the current batch composition. And from Equation (1), it is the quantity that both TTFT and streaming latency are built from — making it the single most actionable signal for an admission controller that cares about both metrics.

IV. TWP AND SYSTEM ARCHITECTURE

This section presents the TWP framework and the overall architecture. We first formalise the system model and the token-based control mechanism used to regulate admission pressure during inference. We then describe the architectural components of the system and how they interact to adapt scheduling decisions based on runtime feedback.

A. System Model

TWP models the serving system as a discrete-time process advancing one decode step at a time. Each request i carries a prompt length p_i and a target output length L_i . Once admitted, it joins the *active set* A_t and generates one token per step until complete.

Definition 2 (Token Load and Batch Size). *The token load $K_t = \sum_{i \in A_t} r_i(t)$ is the total remaining output tokens across all active requests at step t . The batch size $B_t = |A_t|$ is the count of active requests.*

Token load is the quantity TWP controls. Unlike batch size, which counts requests, token load captures how much work remains: a batch of requests near completion imposes little future pressure, while the same batch size with requests

just admitted imposes far more. Bounding K_t therefore gives a more precise handle on where the system sits relative to saturation than bounding B_t directly.

TWP maintains a dynamic token budget H_t , referred to as the *token window*, as a soft upper bound on token load such that $K_t < H_t$. Since $B_t \approx K_t/\bar{r}_t$, keeping K_t below H_t indirectly keeps B_t below the saturation threshold B_0 and thereby keeps $\ell(B_t)$ close to $\ell_{\min}$.

The window is not fixed. It adapts each step using the normalised lowdown $\sigma_t = \hat{\ell}_t/\ell_{\min}$, which measures how far the current ITL has risen above the efficient baseline. The target window is:

$$H_t^* = H_0 \cdot \frac{1 + \min(1, q_t/Q)}{\max(0.5, \sigma_t)}, \quad (3)$$

where q_t is current queue depth and H_0 is the nominal window set at system startup. When ITL is low and the queue is growing, the window expands to absorb more work. When ITL rises above baseline, the window contracts to reduce admission pressure. The window is updated smoothly each step:

$$H_{t+1} = H_t + \gamma(H_t^* - H_t), \quad (4)$$

with gain γ controlling how quickly the window tracks its target.

B. System Design

This subsection presents the architectural design of TWP and the interaction between its core components. We describe the overall system organisation, the feedback control mechanism and the per-step scheduling process that governs request execution during inference.

1) *Architecture*: Figure 3 shows the TWP architecture. Incoming requests queue until the scheduler admits them; admitted requests move to the decode engine, which generates one token per active request per step and streams output to clients. After each step, a monitor reads the step latency and memory state, and a headroom controller uses these to update the token window before the next step begins. This feedback path is the core contribution: the system continuously observes the cost of the current batch and adjusts admission pressure accordingly.

A key property is that the entire feedback path executes within the decode-step boundary: the monitor reads after step t completes, the controller updates H_{t+1} , and the scheduler uses H_{t+1} for the admissions that precede step $t+1$. There is no asynchronous thread or polling delay. Every admission decision is informed by the state one step ago, which is the most recent state available.

All schedulers in the evaluation share the same serverless container pool. Containers lose warmth over idle time, and a cold container incurs a startup delay before serving its first request. The token window operates only within the decode phase of running containers and does not interact with container provisioning.

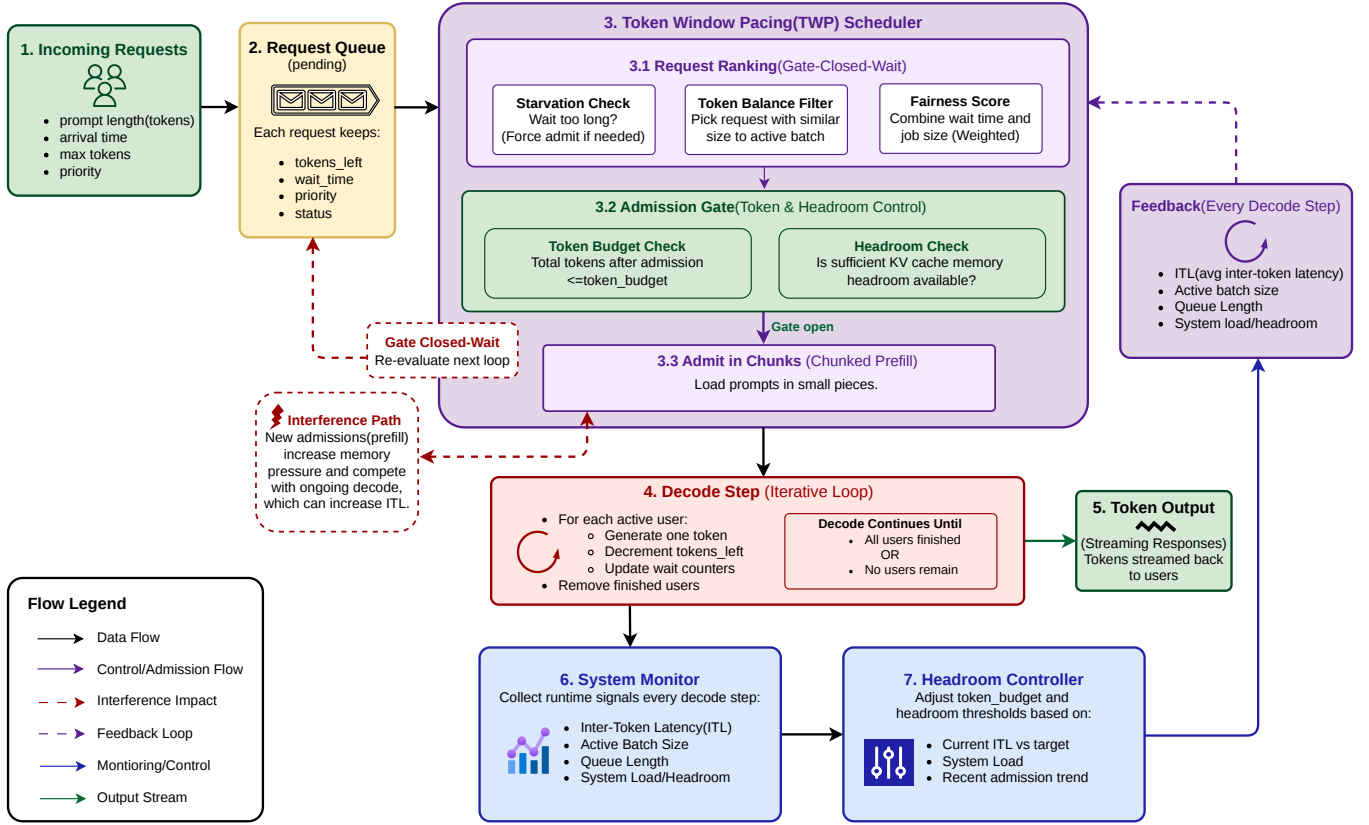


Fig. 3. System-level design of the Token Window Pacing (TWP) scheduler. The scheduler regulates token-level admission using feedback from decode execution.

2) *Feedback Control*: The monitor maintains a smoothed ITL estimate $\hat{\ell}_t$ by applying an exponential filter to the raw step latency, attenuating transient spikes from events unrelated to batch load while still responding to genuine load changes within a few steps. The headroom controller uses $\hat{\ell}_t$ to compute σ_t and update the token window via Equations (3)–(4).

In addition to ITL, the controller tracks the rate of KV-cache block consumption. When memory is being depleted rapidly, the KV-cache admission threshold δ_{kv} is tightened to slow intake before the engine runs out of pages mid-step. When memory is stable, it relaxes. This mechanism operates on the memory signal independently of the ITL signal; both can act simultaneously.

The token window is clamped between a minimum and maximum value. The minimum ensures the system always keeps enough requests active to use available GPU parallelism; the maximum limits how far the window can expand during a quiet period, preserving headroom to absorb a subsequent burst without crossing B_0 .

3) *Per-Step Scheduling*: At each decode step, the scheduler ranks all pending requests, evaluates the admission gate, and stages admitted requests via chunked prefill.

- Request Ranking**: Pending requests are sorted by a composite score that balances three objectives. First, a *starvation priority* proportional to how long a request has

waited: once a request exceeds a maximum wait threshold $T_{\max}$, it is force-admitted unconditionally, bypassing the gate. This gives a hard worst-case latency bound regardless of window pressure. Second, a *token balance* score that favours requests whose expected output length is close to the mean remaining length of the active batch, expressed as:

$$T(r) = \exp\left(-\frac{|\hat{L}(r) - \bar{r}_t|}{\bar{r}_t + 1}\right). \quad (5)$$

Admitting requests that match the current batch composition reduces the fraction of steps where a few long-running requests hold up an otherwise finished batch. Third, a *fairness score* that normalises wait time by job size, preventing a stream of short requests from indefinitely displacing larger ones. The three scores are combined with weights $w_1 > w_2 > w_3$, prioritising the starvation guarantee first, batch efficiency second, and fairness third.

- Admission Gate**: Candidates are considered in ranked order. A request is admitted only if two conditions hold simultaneously: the resulting token load stays within the window ($K_t + r_i \leq H_t$), and available KV-cache memory exceeds the threshold δ_{kv} . If either condition fails, the gate closes and remaining candidates wait for the next

Algorithm 1 Token Window Pacing (TWP) Scheduler

```

1: Initialize token window  $H_0$ 
2: Initialize active set  $A \leftarrow \emptyset$ 
3: for each decode step  $t$  do
4:   // Step 1: Monitor system state
5:   Measure inter-token latency  $\ell_t$ 
6:   Observe queue length  $q_t$  and batch size  $B_t$ 
7:   // Step 2: Update token window (feedback control)
8:   Compute slowdown  $\sigma_t = \ell_t / \ell_{\min}$ 
9:   Compute target window  $H^*$  based on  $\sigma_t$  and  $q_t$ 
10:   $H_t \leftarrow H_{t-1} + \gamma(H^* - H_{t-1})$ 
11:  // Step 3: Request selection
12:  Rank queued requests using:
13:    (i) starvation priority (waiting time)
14:    (ii) token balance (similar size to active set)
15:    (iii) fairness score
16:  // Step 4: Admission control
17:  while  $K_t < H_t$  and queue not empty do
18:    Select next request  $r$ 
19:    if adding  $r$  does not exceed token window then
20:      Admit  $r$  (possibly in chunks)
21:      Add  $r$  to active set  $A$ 
22:    else
23:      break
24:    end if
25:  end while
26:  // Step 5: Decode execution
27:  for each request  $i \in A$  do
28:    Generate one token
29:     $r_i \leftarrow r_i - 1$ 
30:  end for
31:  Remove completed requests from  $A$ 
32:  Update total tokens  $K_t = \sum_{i \in A} r_i$ 
33: end for

```

step. The gate does not skip to smaller requests that might fit within residual headroom, because doing so would bypass the starvation ordering. Force-admitted requests are exempt from both checks.

- iii. **Chunked Prefill:** Admitted requests do not immediately submit their full prompt for prefill. Instead, the prompt is broken into fixed-size chunks processed one per decode step. This bounds the noisy-neighbour penalty per step to a fixed fraction of $\ell_{\min}$ regardless of prompt length. The chunk size C is chosen at the point where further reduction yields negligible additional ITL improvement while TTFT continues to increase — identified during offline characterisation as described in §V.
- iv. **Algorithm:** Algorithm 1 shows the complete per-step execution. The five steps run in strict sequence so that each admission decision uses the most recently measured system state.

V. IMPLEMENTATION

TWP is implemented as a scheduling layer between the request queue and an existing LLM serving backend. At each decode step, it observes the step latency, active batch size, and KV-cache occupancy, updates the token window, and determines which queued requests to admit before the next step. The backend remains unchanged and continues to handle model execution and KV-cache management. As a result, TWP can be integrated into any serving framework that exposes a step-level decode interface, without modifications to model weights, kernels, or GPU execution primitives.

Table I summarises the notation used throughout the design.

TABLE I
NOTATION USED IN TWP.

Symbol	Meaning
H_t	Token window at step t
H_0	Initial token window
K_t	Total tokens remaining across active requests
B_t	Active batch size
$\ell_{\min}$	Baseline decode step latency (bandwidth-bound)
B_0	Saturation batch size
σ_t	Normalised ITL slowdown $= \hat{\ell}_t / \ell_{\min}$
γ	Token window control gain
C	Prefill chunk size (tokens)
q_t	Queue depth at step t
$T_{\max}$	Starvation force-admit threshold (steps)
$\hat{\ell}_t$	Smoothed ITL estimate

TWP requires three deployment-time parameters. The initial token window is set to $H_0 = B_0 \bar{L}$, where B_0 is the measured saturation batch size and $\bar{L}$ is the expected mean output length. The control gain γ determines the responsiveness of the feedback controller, while the prefill chunk size C controls the trade-off between TTFT and decode interference. In our implementation, we use $H_0 = 400$ tokens, $\gamma = 0.15$, and $C = 64$ tokens. These values are obtained through offline characterization and remain fixed throughout all experiments.

TWP controls request admission, batching, and execution ordering, but does not modify the underlying model architecture or execution stack. This design enables deployment on existing LLM serving systems with minimal integration effort.

VI. EVALUATION

We evaluate TWP against conventional scheduling policies and *ServerlessLLM* [2] to assess its ability to reduce latency while maintaining throughput under bursty workloads.

A. Experimental Setup

Implementation and Environment: TWP is implemented in Python as a scheduling layer integrated into a custom LLM serving pipeline supporting request-level and token-level scheduling. Experiments are conducted on NVIDIA Tesla V100 GPUs (32GB) with CUDA 11.8. Unless otherwise stated, all results use a single GPU serving the Meta-Llama-3.2-3B model [25] in FP16 precision.

Workload: Requests are generated from the GSM8K benchmark [26]. Arrivals follow a Poisson process with rate $\lambda = 18$ req/s over a 60-second interval, producing up to 250 requests per run. Prompt lengths are heterogeneous: 60% short (30–80 tokens), 30% medium (120–250 tokens), and 10% long (350–512 tokens). The saturation batch size is $B_0 = 8$.

Baselines: We compare TWP against representative scheduling policies spanning classical queueing, batching, and serverless execution strategies. ServerlessLLM is reimplemented within our runtime to ensure a fair comparison under identical resource and execution conditions. Table II summarizes the evaluated schedulers.

TABLE II
BASELINE SCHEDULERS AND THEIR ADMISSION STRATEGIES.

Scheduler	Policy
FCFS [27]	Admit in arrival order; no concurrency bound.
Round-Robin [28]	Route requests to containers in rotation.
Priority [29]	Admit in shortest-prompt-first order.
DynamicBatch [30]	Cap batch size when ITL exceeds a static target threshold; no token-level feedback.
ServerlessLLM [2]	Implemented within our runtime using FCFS-style admission with locality-aware startup scheduling, fast checkpoint loading, and live migration support.

Metrics: We report Time-to-First-Token (TTFT), end-to-end latency (P50/P95/P99), throughput (tokens/s), and inter-token latency (ITL). To quantify generation stability, we additionally report the ITL coefficient of variation (CV).

B. Results

This section presents the experimental results obtained from the evaluation setup described earlier. Using the defined baselines and evaluation metrics, we compare TWP against existing scheduling policies across various metrics and further analyse its behaviour under diverse workload patterns.

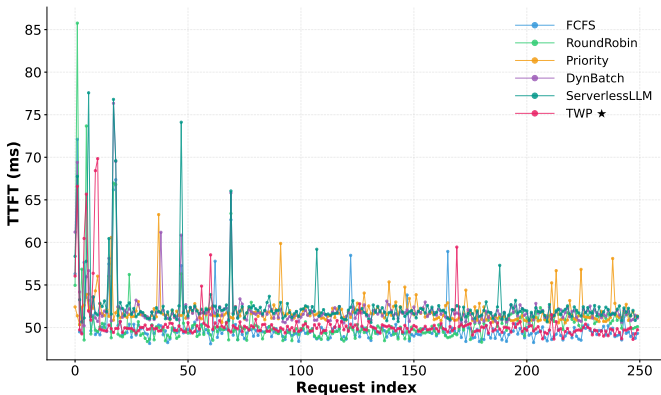


Fig. 4. TTFT comparison across different scheduling policies.

1) *Time-To-First-Token (TTFT)*: Figure 4 shows TTFT for individual requests. While Round Robin and ServerlessLLM experience latency spikes of up to 85 ms and 77 ms, respectively, TWP maintains TTFT near 50 ms throughout the workload. This improvement results from feedback-driven admission control, which limits queue buildup during transient bursts.

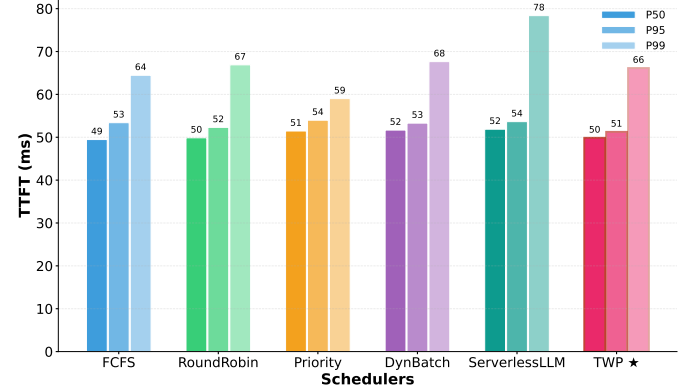


Fig. 5. Comparison of TTFT percentiles (P50, P95, and P99) across different scheduling policies.

2) *TTFT Tail Latency*: Figure 5 reports TTFT percentiles. Although all schedulers achieve comparable median latency (P50 ≈ 50 ms), larger differences emerge at the tail. TWP achieves P95 and P99 latencies of 51 ms and 66 ms, compared to 78 ms P99 for ServerlessLLM and 68 ms for DynamicBatch, indicating improved robustness under contention.

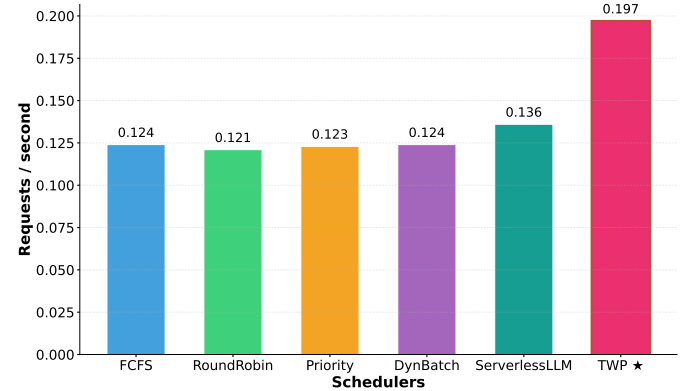


Fig. 6. Throughput comparison across different scheduling policies.

3) *Throughput*: Figure 6 compares throughput across schedulers. Traditional policies achieve 0.121–0.124 requests/second, while ServerlessLLM reaches 0.136 requests/s. TWP achieves the highest throughput at 0.197 requests/s, representing gains of approximately 45% over ServerlessLLM and more than 58% over conventional schedulers. The improvement stems from preventing excessive batch growth while maintaining high GPU utilization.

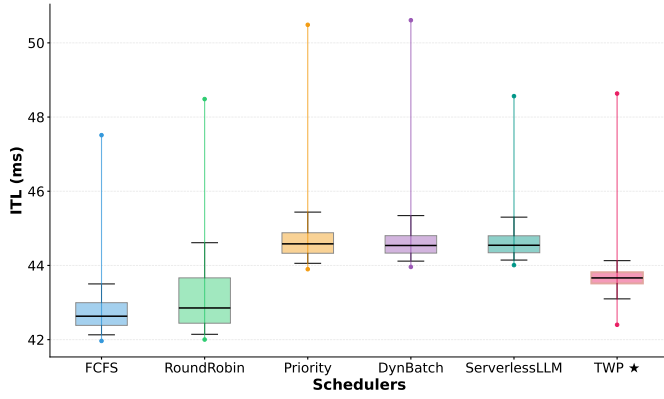


Fig. 7. ITL distribution across different scheduling policies.

4) *Inter-Token Latency (ITL)*: Figure 7 shows the ITL distribution. Baseline schedulers exhibit larger variance and more frequent high-latency outliers, with tail values reaching 61–62 ms. In contrast, TWP maintains a tighter distribution centered around 43.5–43.8 ms, indicating more stable token generation under concurrent workloads.

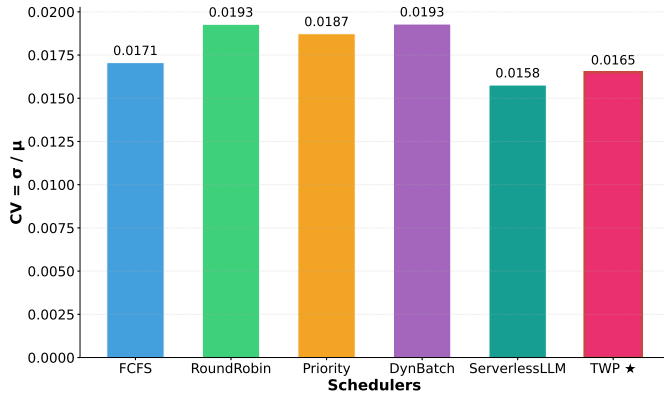


Fig. 8. Coefficient of variation (CV) of inter-token latency across scheduling policies.

5) *ITL Stability*: Figure 8 reports the coefficient of variation of ITL. TWP achieves the lowest variability ($CV = 0.0144$), compared to 0.0203 for Round Robin and 0.0159 for FCFS, demonstrating more consistent decoding performance.

6) *Evaluation Under Diverse Workload Patterns*: To evaluate robustness under varying load conditions, we consider four workload scenarios: *low_load*, *medium_load*, *high_load*, and *burst_load*, summarized in Table III. Each scheduler is evaluated over 10 independent runs per workload.

Request traces use heterogeneous prompt and output lengths. Prompt lengths are sampled from 50, 200, 800 tokens with probabilities 0.5, 0.3, 0.2, while output lengths are sampled from 20, 50, 200 tokens with the same probabilities. This setup captures both short interactive requests and long-context generation workloads.

Across all workloads, TWP consistently achieves lower latency and higher throughput than the baselines. The gains

TABLE III
PERFORMANCE ACROSS WORKLOADS

Scheduler	Throughput (req/s)	P99 Latency (ms)	TTFT (ms)	ITL (ms)	CV ITL
<i>Low Load</i>					
FCFS	363.09	2130.93	52.99	44.29	0.0141
DynamicBatch	363.09	2130.93	52.17	43.26	0.0145
Priority	433.87	2134.36	52.20	43.47	0.0142
RoundRobin	334.48	2147.98	52.17	43.30	0.0147
ServerlessLLM	451.32	2014.46	51.99	43.19	0.0142
TWP	2568.31	1152.40	52.02	43.11	0.0141
<i>Medium Load</i>					
FCFS	267.13	4571.94	52.47	44.05	0.0140
DynamicBatch	267.13	4269.44	51.47	43.02	0.0129
Priority	467.47	4527.29	52.35	44.01	0.0144
RoundRobin	254.80	4618.15	52.28	43.85	0.0138
ServerlessLLM	313.88	4263.09	52.16	43.69	0.0137
TWP	2674.39	2594.39	52.09	43.09	0.0136
<i>High Load</i>					
FCFS	254.62	8915.61	51.94	43.48	0.0141
DynamicBatch	254.62	8493.63	51.89	43.41	0.0131
Priority	448.65	8833.61	53.37	45.09	0.0200
RoundRobin	242.44	8571.43	52.08	43.56	0.0136
ServerlessLLM	299.29	8005.79	53.42	45.14	0.0173
TWP	2709.57	5103.69	52.86	44.10	0.0146
<i>Burst Load</i>					
FCFS	226.30	17869.95	53.25	44.95	0.0169
DynamicBatch	226.30	16894.07	53.13	44.83	0.0172
Priority	452.60	17488.25	53.34	45.22	0.0177
RoundRobin	219.97	18052.02	53.38	45.03	0.0172
ServerlessLLM	258.57	16915.02	53.35	45.04	0.0179
TWP	2532.53	10272.56	52.49	44.68	0.0169

become more pronounced under high-load and burst-load conditions, where uncontrolled admission leads to queue buildup and increased decode interference. By dynamically regulating token admission, TWP maintains stable TTFT and ITL even under sustained bursts.

VII. CONCLUSION

This work presented (TWP), a feedback-driven scheduling layer for serverless LLM inference that regulates request admission at decoding-step granularity using only runtime signals available in existing serving systems. TWP operates without modifying model architectures, attention kernels, KV-cache management, or GPU execution primitives, making it broadly deployable.

Experiments on Poisson-driven workloads in a shared serverless environment show that TWP consistently outperforms existing scheduling policies and ServerlessLLM. It achieves up to 58% higher throughput, reduces tail TTFT by 15–20%, and provides the lowest inter-token latency variability among all evaluated methods. These results demonstrate that step-level, feedback-driven token admission is an effective mechanism for stabilizing LLM inference under dynamic concurrency.

REFERENCES

- [1] M. Shahradd, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '20. USA: USENIX Association, 2020.
- [2] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Low-Latency serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 135–153. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/fu>
- [3] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/you>
- [4] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: <https://doi.org/10.1145/3600006.3613165>
- [5] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," 2024. [Online]. Available: <https://arxiv.org/abs/2403.02310>
- [6] H. Jin, S. Li, and M. A. Gregory, "Gensched: Phase-aware generative scheduling for llm inference in heterogeneous edge networks," in *IEEE INFOCOM 2026 - IEEE Conference on Computer Communications*, 2026, pp. 1–6.
- [7] W. Cui, Z. Xu, H. Zhao, Q. Chen, Z. Li, B. He, and M. Guo, "Efficient function-as-a-service for large language models with tidal," 2025. [Online]. Available: <https://arxiv.org/abs/2503.06421>
- [8] M. Yu, R. Yang, C. Jia, Z. Su, S. Yao, T. Lan, Y. Yang, Z. Wang, Y. Cheng, W. Wang, A. Wang, and R. Chen, "λscale: Enabling fast scaling for serverless large language model inference," 2026. [Online]. Available: <https://arxiv.org/abs/2502.09922>
- [9] W. Zhu, Z. Shen, Z. Shao, H. Dai, and F. Chen, "Tangram: Accelerating serverless llm loading through gpu memory reuse and affinity," 2025. [Online]. Available: <https://arxiv.org/abs/2512.01357>
- [10] C. Lou, S. Qi, C. Jin, D. Nie, H. Yang, Y. Ding, X. Liu, and X. Jin, "Hydraserve: Minimizing cold start latency for serverless llm serving in public clouds," 2025. [Online]. Available: <https://arxiv.org/abs/2502.15524>
- [11] Y. Sui, H. Yu, Y. Hu, J. Li, and H. Wang, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 178–195. [Online]. Available: <https://doi.org/10.1145/3698038.3698509>
- [12] C. Lou, S. Qi, R. Kang, Y. Zhang, C. Sun, P. Wang, B. Liu, X. Liu, and X. Jin, "Warmserve: Enabling one-for-many gpu prewarming for multi-llm serving," 2025. [Online]. Available: <https://arxiv.org/abs/2512.09472>
- [13] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," 2023. [Online]. Available: <https://arxiv.org/abs/2308.16369>
- [14] P. Patel, E. Choukse, C. Zhang, A. Shah, I. n. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ser. ISCA '24. IEEE Press, 2025, p. 118–132. [Online]. Available: <https://doi.org/10.1109/ISCA59077.2024.00019>
- [15] A. Bari, P. Hegde, and G. de Veciana, "Optimal scheduling algorithms for llm inference: Theory and practice," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 9, no. 3, Dec. 2025. [Online]. Available: <https://doi.org/10.1145/3771574>
- [16] R. Lai, H. Liu, C. Lu, Z. Liu, S. Cao, S. Shao, Y. Zhang, L. Mai, and D. Ustiugov, "Tokenscale: Timely and accurate autoscaling for disaggregated llm serving with token velocity," 2025. [Online]. Available: <https://arxiv.org/abs/2512.03416>
- [17] R. Li, F. Chen, and P. Li, "Semi-clairvoyant scheduling of speculative decoding requests to minimize llm inference latency," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, ser. IJCAI '25, 2025. [Online]. Available: <https://doi.org/10.24963/ijcai.2025/951>
- [18] T. Han, Z. Wang, C. Fang, S. Zhao, S. Ma, and Z. Chen, "Token-budget-aware llm reasoning," *arXiv preprint arXiv:2412.18547*, 2024.
- [19] G. Thompson, M. Shahradd, and S. Gopalakrishnan, "Job scheduling in serverless systems: Cost and response time tradeoffs," in *Proceedings of the 27th International Conference on Distributed Computing and Networking*, ser. ICDNC '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 62–70. [Online]. Available: <https://doi.org/10.1145/3772290.3772308>
- [20] Y. Ni, X. Yang, Y. Tang, Z. Qiu, C. Wang, and T. Yuan, *Predictive-LoRA: A Proactive and Fragmentation-Aware Serverless Inference System for LLMs*. New York, NY, USA: Association for Computing Machinery, 2026, p. 1267–1273. [Online]. Available: <https://doi.org/10.1145/3795154.3795359>
- [21] Y. Sui, H. Wang, H. Yu, Y. Hu, J. Li, and H. Wang, "Serverlesslora: Minimizing latency and cost in serverless inference for lora-based llms," 2025. [Online]. Available: <https://arxiv.org/abs/2505.14468>
- [22] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, J. Meng, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, and Y. Shan, "DEEPSERVE: Serverless large language model serving at scale," in *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 57–72. [Online]. Available: <https://www.usenix.org/conference/atc25/presentation/hu-junhao>
- [23] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. D. Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," 2024. [Online]. Available: <https://arxiv.org/abs/2312.11514>
- [24] A. Ali, R. Pincioli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.
- [25] M. AI, "Llama 3.2: Open foundation and fine-tuned language models," <https://huggingface.co/meta-llama/Llama-3.2-3B>, 2024.
- [26] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.
- [27] W. Zhao and J. A. Stankovic, "Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems," in *[1989] Proceedings. Real-Time Systems Symposium*. IEEE, Jan. 1989, p. 16.
- [28] E. Hahne, "Round-robin scheduling for max-min fairness in data networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 7, pp. 1024–1039, 1991.
- [29] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda, *Fixed Priority Scheduling Theory for Hard Real-Time Systems*. Boston, MA: Springer US, 1991, pp. 1–30. [Online]. Available: https://doi.org/10.1007/978-1-4615-3956-8_1
- [30] J. Liu and J. Hu, "Dynamic batch processing in workflows: Model and implementation," *Future Generation Computer Systems*, vol. 23, no. 3, pp. 338–347, 2007. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X06001336>