Resolving the Starvation Paradox: Efficient Disaggregated LLM Serving with Phasor

Anonymous ACL submission

Abstract

Phase-disaggregated LLM serving separates compute-bound prefill from memory-bound decode to improve efficiency and isolation. We show that physical separation alone does not guarantee either: under stochastic production bursts, schedulers driven by phase-local proxies overload compute-saturated prefill workers, throttle the ready rate into decode, and leave KV-resident decode GPUs idle—an efficiency failure rather than a tail-latency artifact.

We propose Virtual Freeness (Φ), a phase-consistent, dimensionless headroom signal that makes prefill compute saturation and decode effective KV capacity loss directly comparable, enabling principled cross-phase trade-offs while remaining orthogonal to existing execution substrates. Built on Φ, Phasor is a holistic control framework for disaggregated clusters that performs micro-scale routing, meso-scale decode maintenance, and macro-scale pool rebalancing.

Evaluated on representative NLP tasks and realistic bursty workloads, Phasor reduces end-to-end P99 latency by 25.7–38.9% compared to state-of-the-art baselines. Crucially, by suppressing fragmentation from a peak of 49.7% in baselines to a 6.2% average, it preserves effective KV capacity, thereby boosting SLO-compliant concurrency per-GPU by 1.1–2.0×, translating to a 12–50% reduction in per-request serving costs. We will make our code and data publicly available upon acceptance

1 Introduction

Autoregressive LLM serving exhibits a fundamental resource duality: prefill is compute-bound (Zhao et al., 2024; Heo et al., 2024; Agrawal et al., 2024; Jha et al., 2024), whereas decode is memory-bound (Hong et al., 2024a; Kamath et al., 2025; Shazeer, 2019). A growing line of work (Zhou et al., 2024; Zhuang et al., 2024; Hong et al., 2024b; Zhong et al., 2024; Ding and Yang, 2025; Patel et al., 2024; Qin et al., 2024) advocates *phase disaggregation* by physically disaggregating prefill and decode to improve serving efficiency and pursue performance isolation.

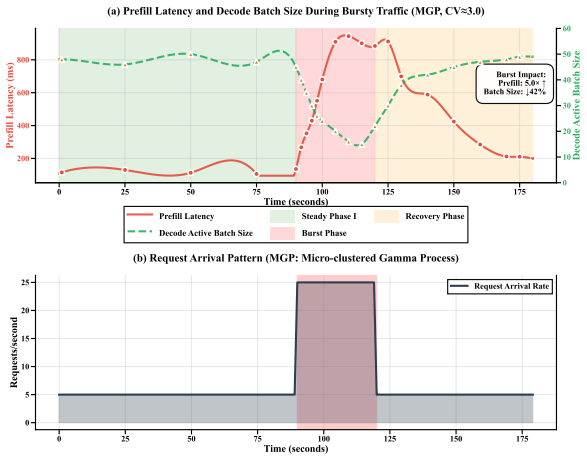

Figure 1: **Pipeline starvation under disaggregation.** Prefill-side congestion throttles the decode-ready rate, driving a high-latency, low-concurrency regime where KV-resident decode GPUs idle.

However, we show that physical separation alone guarantees neither isolation nor efficiency. As Figure 1 shows, stochastic bursts trigger upstream stalls that starve the decode phase, collapsing active batch sizes. This traps the system in a *high-latency, low-concurrency* regime where decode GPUs sit idle despite holding expensive resident KV states—an efficiency failure, not merely a latency artifact.

We attribute this to a *control-plane disconnect*: existing systems rely on phase-local, incomparable proxies (e.g., queue length (Patel et al., 2024)). This blinds schedulers to compute saturation, causing them to overload workers that appear available under memory-centric views, thereby amplifying head-of-line blocking and starving downstream resources.

To bridge this, we propose Virtual Freeness (Φ),

a unified phase-consistent scheduling interface projecting heterogeneous constraints into a dimensionless headroom signal. By rendering prefill compute saturation and decode effective capacity loss directly comparable, Φ enables principled cross-phase tradeoffs. Crucially, this interface is *orthogonal* to execution substrates (e.g., planned phase splits (Zhong et al., 2024), streaming (Feng et al., 2025), or disaggregated KV memory (Qin et al., 2024)) equipping their control plane with a common language for cross-phase decisions.

Built on Φ, we present Phasor, a holistic control framework for disaggregated clusters that integrates dimensionless load modeling with predictive resource orchestration. Phasor collects lightweight telemetry, projects each worker state into Φ, and applies actions at multiple time scales: micro-scale request routing, meso-scale decode-side maintenance, and macro-scale pool rebalancing.

Across ShareGPT-derived NLP tasks (Chiang et al., 2023) and Azure bursts workload (Wang et al., 2025b), Phasor consistently improves end-to-end tail latency and *SLO-constrained efficiency*: at 40–60 req/s, it cuts P99 latency by 38.9% and 25.7% compared to baselines. Subject to strict SLOs, it boosts compliant concurrency per-GPU by 1.1–2.0$\times$ (reducing GPU cost per request by 12–50%). It further suppresses fragmentation (49.65% peak in baselines $\rightarrow$ 6.20% average), preserving effective KV capacity and sustaining concurrency.

Our contributions are:

- We identify a control-plane failure mode in disaggregated serving where misaligned phase-local metrics induce *pipeline starvation*, yielding high latency and low effective concurrency despite phase separation.
- We propose Virtual Freeness (Φ), a unified scheduling interface that provides a phase-consistent headroom signal to directly trade off prefill compute saturation against decode-side effective capacity loss.
- We build Phasor, a holistic control framework for disaggregated clusters driven by Φ, and demonstrate improved end-to-end performance and SLO-constrained efficiency on diverse NLP and realistic bursty workloads.

2 Motivation

The transition from monolithic to disaggregated architectures is predicated on a core promise: by physically decoupling the compute-bound prefill phase from the memory-bound decode phase, systems can eliminate resource contention and achieve performance isolation (Zhong et al., 2024; Hu et al., 2024b; Jiang et al., 2025) .

We show this promise can fail even under strict physical separation, when the control plane relies on misaligned scheduling signals: upstream congestion can still propagate downstream and collapse end-to-end performance and serving efficiency.

Controlled setup. We mirror DistServe's physical phase separation but intentionally remove system-specific heuristics, creating a clean setting. This neutral setup (8x NVIDIA V100, NVLink within each phase pod, Qwen2.5-7B-Instruct FP16) ensured that any observed bottlenecks are attributable to the control signal rather than architectural noise.

2.1 The Reality: The Starvation Paradox

We inject a BurstGPT-derived shock–recovery trace (Wang et al., 2025b) using an MGP arrival model (CV$\approx$3.0) to emulate micro-clustered bursts (λ=5 $\rightarrow$ 25,req/s at t=90–120,s). Each request features an average input length of 1024 tokens and an output length of 256 tokens, mirroring RAG-like workloads.

As shown in Figure 1, a burst triggers a clear *scissor effect*: prefill latency rises sharply (from $\sim$116 ms to $>$900 ms), while the decode-side active batch size drops from $\sim$48 to $<$16.

This is *pipeline starvation*: overloaded prefill workers stall upstream, draining the decode-ready queue and collapsing the active decode batch, which lowers effective utilization and SLO-compliant concurrency per GPU. Decode GPUs still keep KV states resident, but idle behind upstream head-of-line blocking while waiting for prefill outputs. This is an efficiency failure rather than a mere tail-latency artifact, showing that physical phase separation alone does not guarantee performance isolation

2.2 The Diagnosis: The Utilization Phantom

Why does the scheduler allow prefill to become the blocking bottleneck? The root cause stems from a structural blind spot: single-dimensional metrics fail to capture the conflicting resource demands of compute-bound prefill versus memory-bound decode.

We verify this by profiling an isolated prefill pool

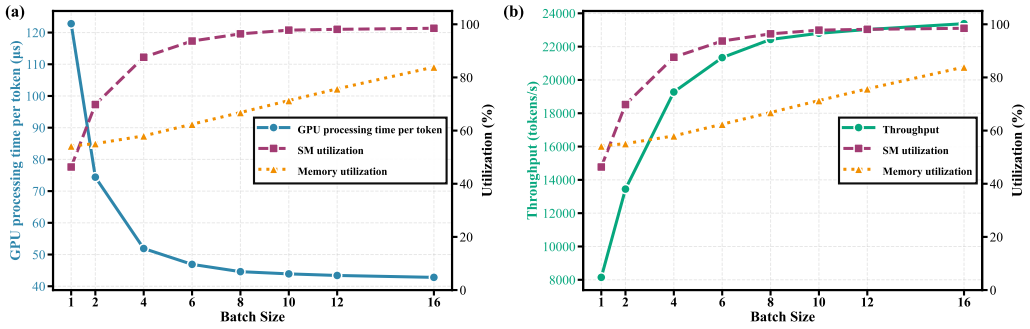

Figure 2: **The Utilization Phantom.** While memory utilization (blue) suggests the prefill node is underloaded ($\approx 60\%$), compute units (SM, orange) are already fully saturated ($> 93\%$). Schedulers relying on memory metrics are blind to this saturation.

(Figure 2). As batch size scales, computational efficiency improves (per token processing time drops $122.8\mu s \rightarrow 42.8\mu s$) but quickly hits a hard wall: SM utilization saturates (>93.7% at batch size 6).

Crucially, however, memory utilization hovers at $\sim$60%. Figure 2(b) confirms that throughput gains diminish sharply beyond batch size 8 (only 21% improvement), despite 38% of memory capacity sitting idle.

This discrepancy creates a *Utilization Phantom*: the worker appears *available* in memory space while being *saturated* in compute time. Schedulers relying solely on memory headroom are thus blind to this compute saturation, erroneously routing burst traffic to overloaded nodes and triggering the downstream starvation observed in Section 2.1.

2.3 Consequence: From Skew to Stall

The utilization phantom manifests as queue divergence in the prefill pool: a memory-centric policy repeatedly routes bursts to memory-available yet compute-saturated workers, creating stragglers. In our burst trace, the busiest prefill worker builds a 40+ queue while others stay around $\sim$15, whereas a compute-aware policy keeps queues near-uniform (details in Appendix A). This skew amplifies upstream head-of-line blocking, reduces the decode-ready rate, and pushes the system into a high-latency, low-utilization regime, hurting both tail performance and cost efficiency despite phase disaggregation.

2.4 The Control Plane Disconnect

The observed queue divergence exposes a systemic disconnect: while the *mechanisms* of disaggregation have matured, the *metrics* governing them remain dangerously fragmented.

Prior research has rigorously established the physical foundations for efficiency: Phase Disaggregation (Zhong et al., 2024) decouples interference, State Disaggregation (Qin et al., 2024) pools global capacity, and Stream Scheduling (Feng et al., 2025) introduces pipeline fluidity. These works brilliantly solve *where* to place computations and *how* to move data.

The Missing Link: Metric Incomparability. However, the *control signals* driving these decisions operate in silos. Consequently, each system's control plane optimizes a different local proxy metric, e.g., *token velocity* (Feng et al., 2025) or *queue length* (Patel et al., 2024). Crucially, they lack a common denominator to weigh the quadratic compute cost of a prefill burst against the linear memory risk of a decode stream.

Our Solution: The Phasor Framework. To bridge this semantic gap, we identify the need for a unified scheduling interface. We first propose Virtual Freeness (Φ), a dimensionless projection designed to translate distinct physical constraints (Time vs. Space) into a unified language. Crucially, This interface is orthogonal to existing execution paradigms, equipping the control plane with a bilingual capability that transforms chaotic interference into a tractable optimization objective for cross-phase control.

Built upon this interface, we propose Phasor, a holistic control framework that integrates dimensionless load modeling with predictive resource orchestration.

3 Methodology

We now formalize the unified interface, and show how Phasor uses it to drive closed-loop routing and maintenance in disaggregated clusters.

3.1 Virtual Freeness: The Unified Mathematical Form

To optimize across heterogeneous phases, we formalize the system state using a unified mathematical framework. For any worker w, Virtual Freeness (Φ) is a *SLO-normalized residual headroom*:

$$\Phi(w) = 1 - \frac{\mathcal{U}_{virt}(w)}{\mathcal{C}_{phys}(w)}. \tag{1}$$

Here, $\mathcal{C}_{phys}$ is the hard physical budget and $\mathcal{U}_{virt}$ is the predicted virtual load under the corresponding phase constraint.

Crucially, while the *form* of Φ is unified, its *internal definition* adapts dynamically to the phase-specific physics of the worker. We formulate this duality using the following system of equations:

$$\Phi(w) = \begin{cases} w \in \mathcal{P}\textbf{(Time-Bound):} \\ 1 - \dfrac{\sum_{r \in \mathcal{Q}_w} \left(\alpha\, l_{in}(r)^2 + \beta \cdot P_m \cdot l_{in}(r)\right)}{\mathbb{P}_{peak}\, T_{SLO}^{\text{TTFT}}} \\ w \in \mathcal{D}\ \textbf{(Space-Bound):} \\ 1 - \dfrac{N_{used}(w) + \mathcal{H}_{frag}(w)}{\mathbb{M}_{total}} \end{cases} \tag{2}$$

For the Prefill Phase, $\mathcal{U}$ represents the Virtual Computational Load, defined as the estimated compute cost for all requests in the current instance queue $\mathcal{Q}$, where l_{in} is the input length of request r. The quadratic term $\alpha \cdot l_{in}^2$ captures the dominant self-attention scaling, while the linear term $\beta \cdot P_m \cdot l_{in}$ captures FFN-related compute, where P_m is the (active) parameter count obtained from model metadata. $\mathcal{C}_{\mathcal{T}}$ denotes the compute budget over the TTFT window, determined by the hardware peak throughput at the deployed precision ($\mathbb{P}_{peak}$) and the TTFT budget T_{SLO}^{TTFT}. We discuss the generalization and calibration of this time-bound surrogate in Appendix B.3.

For the Decode Phase, the capacity $\mathcal{C}$ is defined by the hard memory limit $\mathbb{M}_{total}$. Here, the load includes the count of physically allocated KV blocks N_{used} and a fragmentation penalty $\mathcal{H}_{frag}$, defined as stranded capacity on worker w—free blocks that are physically available but practically unusable for large bursts without consolidation.

All hyperparameters and system constants (e.g., $\alpha, \beta, \mathbb{P}_{peak}, T_{SLO}^{\text{TTFT}}, \mathbb{M}_{total}, \mathcal{H}_{frag}$), along with calibration and some parameters' sensitivity analyses, are provided in Appendix B and F.

Algorithm 1 Phasor Holistic Control Loop

Require: Request stream R, pools $\mathcal{P}, \mathcal{D}, \mathcal{M}$
Require: Calibrated params $\alpha, \beta, \theta_{frag}, \theta_{flow}$
while system running **do**
Micro: Gradient Routing ($t \sim \mu s$)
  if new request r arrives **then**
    $w^* \leftarrow \arg\max_{w \in \mathcal{P}} \Phi_{\mathcal{T}}(w)$
    Dispatch r to w^*
  end if
Meso: Proactive De-fragmentation ($t \sim ms$)
  for all decode worker $d \in \mathcal{D}$ **do**
    compute $\Phi_{\mathcal{S}}(d)$ ▷ space-bound freeness, Eq. (2)
    if $\Phi_{\mathcal{S}}(d) < \theta_{frag}$ **and** $\mathcal{H}_{frag}(d)$ is high **then**
      Trigger AsyncDefrag(d)
    end if
  end for
Macro: Elastic Flow Control ($t \sim s$)
  $\bar{\Phi}_{\mathcal{P}} \leftarrow$ AvgFreeness($\mathcal{P}$);
  $\bar{\Phi}_{\mathcal{D}} \leftarrow$ AvgFreeness($\mathcal{D}$)
  if $\bar{\Phi}_{\mathcal{P}} < \theta_{flow} \cdot \bar{\Phi}_{\mathcal{D}}$ **and** $\mathcal{M} \neq \emptyset$ **then**
    $m \leftarrow \mathcal{M}$.pop(); **Promote** m to $\mathcal{P}$
  else if $\bar{\Phi}_{\mathcal{D}} < \theta_{flow} \cdot \bar{\Phi}_{\mathcal{P}}$ **and** $\mathcal{M} \neq \emptyset$ **then**
    $m \leftarrow \mathcal{M}$.pop(); **Promote** m to $\mathcal{D}$
  end if
  Sleep for control interval Δt
end while

3.2 Phasor Architecture

Phasor is a centralized control plane over three pools: Prefill ($\mathcal{P}$), Decode ($\mathcal{D}$), and an elastic Mixed pool ($\mathcal{M}$) (Figure 3). It collects lightweight telemetry (per-worker queue metadata and decode-side KV allocation/fragmentation maps), projects each worker state into the unified Φ space, and enforces decisions at multiple time scales: micro request routing, meso decode maintenance, and macro pool rebalancing.

3.3 Actuation: Multi-Scale Control Mechanisms

Equipped with the unified Φ signal, Phasor operationalizes the control loop through three synergistic mechanisms spanning different time scales. Visualization of scheduling mechanisms is illustrated in Appendix C.

Δt denotes the sampling interval. We conducted a sensitivity analysis on Δt in the Appendix F.

Zero-Downtime KV Migration. Phasor pipelines KV migration by overlapping asynchronous KV replication with ongoing inference and using a brief final handoff for consistency; detailed mechanisms and overhead in Appendix D.

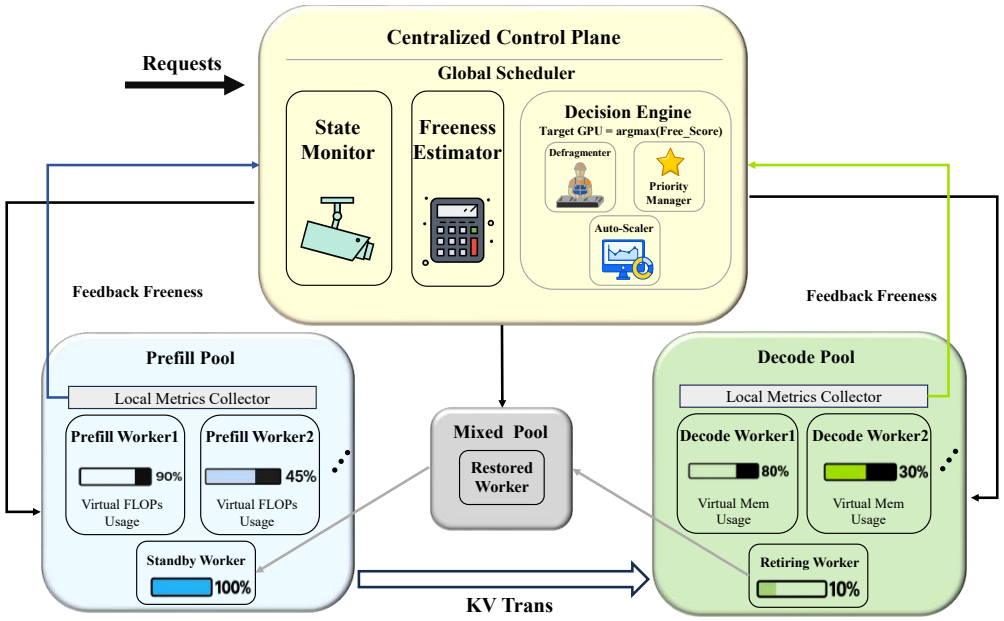

Figure 3: **Overview of the Phasor Architecture.** The Global Scheduler orchestrates resources via a closed loop: collecting raw telemetry, projecting it into the unified Φ space, and executing actuation decisions.

Micro (Routing). Phasor dispatches each arrival to the prefill worker with the highest time-bound freeness $\Phi_{\mathcal{T}}$. This greedy move-to-the-best-Φ rule steers bursty, compute-heavy requests away from workers already dominated by quadratic prefill costs, mitigating the queue divergence described in Section 2.

Meso (De-fragmentation). Phasor treats fragmentation as stranded capacity and triggers ASYNCDEFRAG when $\Phi_{\mathcal{S}}(d) < \theta_{frag}$. Rather than performing physical compaction (obviated by PagedAttention), it asynchronously migrates a subset of KV states to other workers. This effectively performs online bin-packing, consolidating scattered capacity on the source node to accommodate new heavy requests. (details in Appendix D).

Macro (Flow Control). Phasor compares the mean freeness of $\mathcal{P}$ and $\mathcal{D}$ and promotes a standby node from $\mathcal{M}$ when one pool's mean freeness drops below the other by a factor of the *flow-control ratio* θ_{flow}. The choice and sensitivity of θ_{flow} are detailed in Appendix B.5 and F.

4 Experiments

Our evaluation aims to answer three fundamental questions:

RQ1 (End-to-End Performance): Does Phasor suppress cross-phase interference and deliver consistently lower end-to-end tail latency across NLP-critical workloads and real-world production bursts?

RQ2 (Efficiency): Can Phasor substantially amplify the serving efficiency of fixed hardware budgets under strict SLO constraints?

RQ3 (Validity): Are gains attributable to Phasor's key mechanisms, and are results stable to threshold/sampling variations?

4.1 Experimental Setup

Testbed and Models. We conduct all experiments on a server equipped with 8 $\times$ NVIDIA V100-32GB GPUs interconnected via NVLink, representative of a standard high-performance inference pod. This setting leaves limited KV-cache headroom, making capacity sensitive to fragmentation. We deploy Qwen2.5-7B-Instructt (FP16) as the primary model. We also employ Llama-3-8B for ablation studies to verify cross-model generalization.

Workloads: Semantic Task Archetypes. To bridge the gap between system metrics and NLP semantics, we do not rely on random synthetic traces. Instead, we construct three distinct workload scenarios by sampling directly from the ShareGPT dataset (Chiang et al., 2023): First, Context-Heavy Tasks feature long inputs and short outputs. This pattern dominates prefill compute time, stressing the $O(L^2)$ attention bottleneck. Second, Generation-Heavy Tasks utilize short inputs but long outputs. This pattern stresses memory capacity and is highly susceptible to KV fragmenta-

ID	In (L_{in})	Out (L_{out})	NLP Task
Sharegpt-L-S	∼2048	∼128	**Summarization**
Sharegpt-S-L	∼128	∼2048	**Code Generation**
BurstGPT	1024 (avg)	256 (avg)	**Realistic Mix**
Sharegpt-S-S*	*128*	*128*	*Short QA*
Sharegpt-L-L*	*2048*	*512*	*Translation*
Sharegpt-M-M*	*1024*	*1024*	*Dialog*

Table 1: **Evaluated Workloads.**

tion. Finally, For production bursts, we use BurstGPT (Wang et al., 2025b) derived from Azure traces to reflect real-world arrival variability. Table 1 summarizes these characteristics.

SLO settings and metrics. We report TTFT, TPOT, and end-to-end latency, and evaluate serving efficiency via SLO-constrained capacity under a fixed 8-GPU budget. SLO thresholds (TTFT/TPOT) are configured per workload scenario; the exact settings are summarized in the Appendix B.7.

Baselines. We compare Phasor against two state-of-the-art systems under an identical 8-GPU budget. DistServe (Zhong et al., 2024) represents the static disaggregation paradigm. Splitwise (Patel et al., 2024) represents standard disaggregation with a reactive Mixed Pool, routing requests based on queue length. These baselines cover the dominant static vs. reactive control paradigms in phase-disaggregated serving.

4.2 End-to-End Performance (RQ1)

To answer RQ1, we evaluate Phasor's latency performance under heterogeneous bottlenecks. Figure 4 presents the latency panorama across three critical NLP task archetypes. Comprehensive results for all workloads are provided in Appendix E.

Scenario 1: Realistic Production (BurstGPT). Across all burst phases, Phasor reduces the average total requests P99 latency by 34.8% compared to Splitwise and 22.4% compared to DistServe. Under high load (40–60 req/s), reactive signals mask the phase-specific bottleneck, causing cascading stragglers. Virtual Freeness exposes this risk at dispatch, routing requests to workers with residual headroom to flatten load variance. Phasor therefore bounds total requests P99 latency at 206.8 s, versus 338.6 s (Splitwise) and 278.4 s (DistServe), a 38.9% and 25.7% reduction.

Scenario 2: Summarization – Prefill Saturation. Across all arrival rates, Phasor reduces average TTFT P99 by 44.2% versus Splitwise and 34.1% versus DistServe. Phasor's gain comes from making the *superlinear* long-context prefill cost a first-class routing signal: by pricing an l_{in}^2-scaled cost in $\Phi_{\mathcal{T}}$, it avoids straggler-dominated queues where a few long requests dominate prefill service time and inflate TTFT, preventing head-of-line blocking from cascading into decode starvation.

Scenario 3: Generation – Memory Fragmentation. Across all rates, Phasor reduces average TPOT by 15.6%/7.6% vs. Splitwise/DistServe. Specifically at λ=40, it bounds TPOT P99 at 112.8,ms (vs. 136.8/121.2,ms) and total requests P99 latency at 174.6,s (vs. 216.4/192.4,s). These gains stem from preemptive memory hygiene guided by Virtual Freeness. While baselines lose effective capacity to fragmentation, causing false-OOMs that throttle concurrency, Phasor proactively defragment memory. This preserves contiguous space, ensuring stable decode concurrency and low TPOT.

4.3 Resource Efficiency & Mechanism (RQ2)

We quantify serving efficiency by *SLO-constrained capacity*: the largest closed-loop *in-flight concurrency* that a fixed 8-GPU cluster can sustain while keeping both TTFT P99 and TPOT P99 within their workload-specific SLO budgets. We report the exact $(T_{SLO}^{\text{TTFT}}, T_{SLO}^{\text{TPOT}})$ settings per workload in Table 3 (Appendix B.7).

Capacity Gains Figure 5 demonstrates that Phasor achieves superior SLO-compliant concurrency compared to baselines across diverse NLP workloads on an 8-GPU budget. Specifically, it delivers 1.85× capacity over DistServe (1.99× vs. Splitwise) on the realistic *Production Mix*. These gains persist across distinctive regimes: 1.66× on memory-bound *Code Generation*, 1.41× on dual-bottleneck *Translation*, and 1.13–1.89× across Dialog, Summarization, and Q&A." Overall, Phasor improves SLO-compliant concurrency per GPU by 1.1–2.0×. Under a standard closed-loop assumption where the completion rate scales approximately with sustainable concurrency at fixed client think-time and workload mix, this corresponds to cutting the GPU cost per SLO-satisfying request by roughly 12–50% on the same budget.

To understand why Phasor unlocks more SLO-compliant concurrency, we examine the decode-side memory state and quantify fragmentation over time.

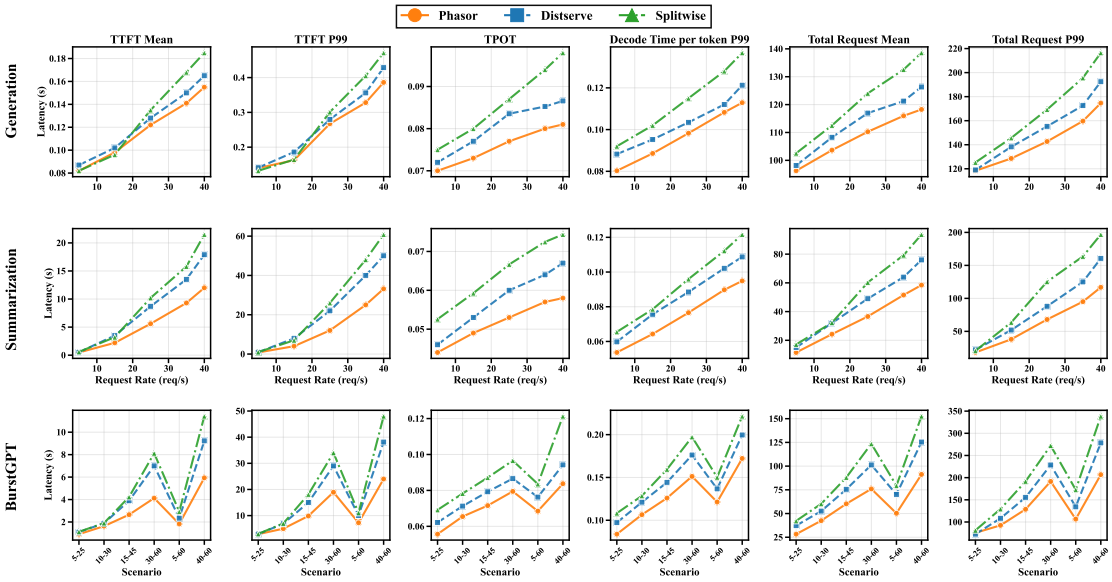

Figure 4: **End-to-end Latency Performance.** Phasor establishes a superior Pareto frontier compared to baseline.

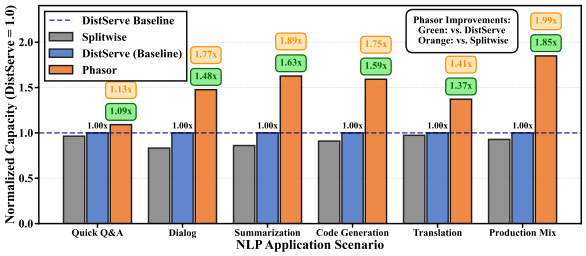

Figure 5: **SLO-constrained capacity.** Phasor yields the largest gains in generation-heavy and bursty workloads under the same 8-GPU budget.

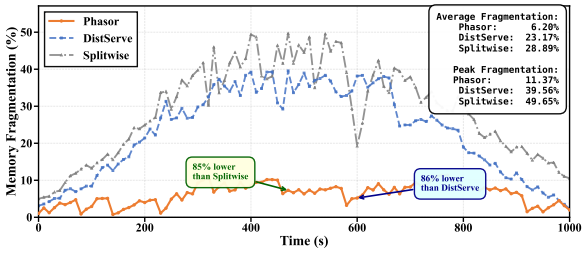

Figure 6: **Mechanism Verification: Fragmentation Suppression.** Baselines accumulate fragmentation and peak at 49.65%, while Phasor bounds fragmentation at a 6.20% average

Attribution: Proactive De-fragmentation. Figure 6 shows that baselines accumulate severe fragmentation (49.7%) under bursts. While PagedAttention handles physical non-contiguity, it cannot consolidate free blocks scattered across workers. Guided by Eq. (2), Phasor proactively triggers AsyncDefrag to asynchronously migrate fragmented memory between workers, effectively reclaiming contiguous capacity. Consequently, Phasor maintains a low 6.2% average fragmentation under the same load.

By reducing peak fragmentation at t=470 s by 85.3% (7.29% vs 49.65%), Phasor converts wasted holes into usable slots, confirming proactive hygiene as the decisive lever for the capacity gains in Figure 5.

Energy Efficiency. Finally, we verify that these capacity gains translate to sustainability. By amortizing static power over higher throughput, Phasor reduces the energy cost per token by 28.0% compared to the Splitwise baseline (0.175 J/T vs. 0.243 J/T); see Appendix G for the detailed analysis.

4.4 Architectural Validity (RQ3)

To attribute Phasor's gains to its design (rather than incidental heuristics), we conduct subtractive ablations by disabling (i) Virtual Freeness (Φ), (ii) proactive de-fragmentation, and (iii) the Mixed Pool. Table 2 reports P99 latency and *SLO-constrained capacity* normalized to DistServe.

Virtual Freeness is the primary driver of tail latency. Removing Φ causes the largest and most consistent P99 regression across workloads. On Qwen2.5-7B-Instruct, Summarization TTFT P99 degrades from 0.66 to 0.92 (+39%), and Production-Mix total P99 degrades from 0.74 to 0.90 (+22%), showing that the unified time-bound signal is essential to avoid straggler-dominated prefill queues and the resulting HOL blocking.

Table 2: **Ablation of Phasor components.** Normalized to DistServe (Baseline=1.00). Removing Φ, Mixed Pool, or De-frag degrades performance significantly.

System Variant	Norm. Latency P99 (↓)			Norm. Capacity (↑)		
	Summ. (TTFT)	Code (Total)	Prod. Mix (Total)	Code Gen.	Summ.	Prod. Mix
Data Format: Qwen2.5-7B-Instruct (Left) / Llama-3-8B (Right)						
Splitwise (Static)	1.22 / 1.24	1.13 / 1.15	1.34 / 1.36	0.91 / 0.90	0.86 / 0.85	0.93 / 0.92
DistServe (Baseline)	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Phasor (Full System)	**0.66 / 0.69**	**0.91 / 0.93**	**0.74 / 0.76**	**1.59 / 1.52**	**1.63 / 1.58**	**1.85 / 1.78**
Ablations (Impact of removing components)						
– w/o Virtual Freeness	0.92 / 0.94	0.95 / 0.97	0.90 / 0.92	1.42 / 1.36	1.55 / 1.50	1.52 / 1.46
– w/o Mixed Pool	0.72 / 0.75	0.93 / 0.95	0.82 / 0.84	1.50 / 1.44	1.58 / 1.53	1.48 / 1.42
– w/o De-frag	0.68 / 0.71	0.96 / 0.98	0.82 / 0.84	1.35 / 1.30	1.60 / 1.55	1.68 / 1.62

De-fragmentation is decisive for memory-bound capacity. Disabling de-fragmentation yields the largest capacity drop in generation-heavy workloads: Code-Generation capacity falls from 1.59 to 1.35 (-15%). This suggests that Φ-based routing is necessary to avoid placing load on risky workers, but de-fragmentation is needed to reclaim fragmentation-induced *effective* KV-cache capacity and keep decode concurrency stable.

The Mixed Pool specializes in burst elasticity. Removing the Mixed Pool primarily hurts burst capacity: Production-Mix capacity drops from 1.85 to 1.48 (-20%), while its impact on steady synthetic regimes is smaller, confirming that the Mixed Pool mainly provides macro-scale elasticity rather than per-request routing benefits.

Robustness to parameter drift. We sweep θ_{frag}, θ_{flow}, and the control epoch Δt in Appendix F and observe stable performance over a broad range, indicating that Phasor does not rely on fragile tuning.

Cross-model generalization. The same *specialization pattern* holds on Llama-3-8B (Table 2): Φ remains the dominant driver of tail-latency improvements across workloads, while de-fragmentation is the key enabler for capacity in memory-bound generation, and the Mixed Pool primarily affects Production-Mix capacity.

5 Related Work

Phase Disaggregation. Splitwise (Patel et al., 2024) and DistServe (Zhong et al., 2024) formalized separating prefill/decode phases, employing specialized pools with queue-based routing (queue-/token-backlog proxies) or offline profiling for planning-based split under TTFT/TPOT constraints. P/D-Serve (Jin et al., 2024) extends this with dynamic mapping for large-scale deployments.

Dynamic & Stream-Based Scheduling. Recent work enhances execution fluidity. WindServe (Feng et al., 2025) introduces stream-based scheduling to minimize bubbles. TaiChi (Wang et al., 2025a) dynamically toggles aggregation levels based on SLO regimes, while DéjàVu (Strati et al., 2024) leverages KV-cache streaming as a substrate for prompt-token disaggregation and fast recovery under failures.

State-Centric Optimization. Addressing the memory wall, Mooncake (Qin et al., 2024) and MemServe (Hu et al., 2024a) propose disaggregated architectures pooling global resources (GPU HBM and CPU DRAM). semi-PD (Hong et al., 2025) advocates storage aggregation, while Nexus (Shi et al., 2025) explores fine-grained intra-engine isolation.

6 Conclusion

This work demonstrates that physical separation alone fails to guarantee performance isolation. We identify a control-plane gap: phase-local, misaligned metrics can overload compute-saturated prefill workers, throttling the ready-task rate into decode so decode GPUs sit idle while still keeping KV states resident. To bridge this, we propose Phasor, a holistic control framework built upon the Virtual Freeness interface. By translating conflicting Time (latency) and Space (fragmentation) constraints into a unified coordinate system, Phasor transforms the control plane from a set of disjoint heuristics into a tractable gradient optimization problem, effectively unlocking the true efficiency potential of disaggregated serving.

Limitations

The limitations below primarily affect the *actuation cost* and *scalability* of the framework, rather than the phase-consistent scheduling interface itself.

State-transfer bandwidth and topology. Phasor's elastic rebalancing relies on amortizing KV/state transfers (Patel et al., 2024; Zhong et al., 2024; Qin et al., 2024); when the interconnect is low-bandwidth (e.g., Ethernet-only or PCIe-constrained fabrics), migration overhead can contend with serving traffic and dilute the net capacity gains (Patel et al., 2024; Zhong et al., 2024; Qin et al., 2024). In such regimes, Phasor naturally degrades to a purely routing-based controller (using Φ for dispatch), as the cost of physical rebalancing would outweigh the fragmentation benefits.

Control-plane scalability and fault tolerance. We prioritize global scheduling optimality, which fits pod-scale deployments typical of high-performance inference. While sufficient for our evaluation scope, a single global scheduler may become a bottleneck at data-center scales (Schwarzkopf et al., 2013; Ousterhout et al., 2013). Future work involves extending Phasor to a hierarchical architecture, where local agents handle micro-second actuation while a global coordinator manages coarse-grained pool sizing.

Extremely long contexts. For workloads with contexts orders of magnitude larger than our test set (e.g., >100K tokens), the sheer cost of moving KV states dominates, regardless of bandwidth (Patel et al., 2024; Kwon et al., 2023). In these scenarios, Phasor's interface should be paired with copy-avoidant mechanisms (e.g., remote memory disaggregation or logical caching) (Gu et al., 2017; Huang et al., 2022; Kwon et al., 2023; Qin et al., 2024) to achieve rebalancing via *access redirection* rather than physical state movement (Gu et al., 2017; Huang et al., 2022).

References

Amey Agrawal and 1 others. 2024. Taming (throughput-latency) tradeoff in llm inference with sarathi-serve. *18th USENIX OSDI*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. https://lmsys.org/blog/2023-03-30-vicuna/. Accessed: 2025-02-20.

Zhiqiang Ding and Tongkai Yang. 2025. Dynamicattention: Dynamic kv cache for disaggregate llm inference. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE.

Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. Windserve: Efficient phase-disaggregated llm serving with stream-based dynamic scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1283–1295.

Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA. USENIX Association.

Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 722–737.

Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, and 1 others. 2025. semi-pd: Towards efficient llm serving via phase-wise disaggregated computation and unified storage. *arXiv preprint arXiv:2504.19867*.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024a. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161.

Ke Hong and 1 others. 2024b. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems (MLSys)*, 6:148–161.

Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and 1 others. 2024a. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*.

Junda Hu, Yinmin Zhong, Shengyu Liu, Jianbo Chen, Xuanzhe Liu, and Hao Zhang. 2024b. TetriInfer: Disaggregated locality-aware inference for LLMs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Jianyu Huang, Hailong Yang, Anthony Cheung, Shan Lu, Michael Leach, Aastha Mehta, Ayan Sinha, Prabhakar Misra, Karthikeyan Shanmugam, and Pradeep

Dubey. 2022. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '22)*.

Siddharth Jha, Coleman Hooper, Xiaoxuan Liu, Sehoon Kim, and Kurt Keutzer. 2024. Learned best-effort llm serving. *arXiv preprint arXiv:2401.07886*.

Youhe Jiang, Ran Yan, and Binhang Yuan. 2025. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. *Preprint*, arXiv:2502.07903.

Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, and 1 others. 2024. P/d-serve: Serving disaggregated large language model at scale. *arXiv preprint arXiv:2408.08147*.

Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 897–912.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*.

Kay Ousterhout, Aurojit Panda, Joshua Rosen, Ion Morgenthaler, Simon andMatin, Scott Shenker, and Sylvia Ratnasamy. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 69–84.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE.

Yuxin Qin, Weqning Li, , and 1 others. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00000*.

Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

Xiaoxiang Shi, Colin Cai, Junjia Du, Zhanda Zhu, and Zhihao Jia. 2025. Nexus: Taming throughput-latency tradeoff in llm serving via efficient gpu sharing. *arXiv e-prints*, pages arXiv–2507.

Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving. *Preprint*, arXiv:2403.01876.

Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. 2025a. Prefill-decode aggregation or disaggregation? unifying both for goodput-optimized llm serving. *arXiv preprint arXiv:2508.01989*.

Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, and 1 others. 2025b. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 5831–5841.

Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Llm-pq: Serving llm on heterogeneous clusters with phase-aware partition and adaptive quantization. *arXiv preprint arXiv:2403.01136*.

Yinmin Zhong, Shengyu Liu, Jianbo Chen, Junda Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating pre-fill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, and 1 others. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*.

Jidong Zhuang and 1 others. 2024. Vserve: A context-aware serving system for large language models. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC 24)*.

A Additional Evidence for Prefill Queue Divergence

As shown in Figure 7, under memory-aware scheduling, *prefill* requests accumulate catastrophically on "Phantom" nodes—Node0's queue reaches 40+ requests (peaking at 46), while other prefill workers maintain queues of only ∼19. In contrast, compute-aware scheduling distributes prefill load evenly across all prefill workers (∼16 requests each). This represents a 2.4× load skew, with Node0's queue growing 4.6× from baseline while queues on other prefill workers merely double.

Figure 7(b) quantifies this prefill-phase disparity: during the burst plateau (300–420s), Node0 accumulates an average queue length of 36.2 requests under memory-aware scheduling, compared to 15.1 for other prefill workers and a uniform ∼15.8 under compute-aware scheduling.

B Parameter Calibration and Configuration

To ensure reproducibility, this appendix details how Phasor calibrates the time-bound cost model parameters and configures control thresholds in our evaluation. Unless otherwise specified, all values refer to our V100 (FP16) serving stack.

Notation. For a prefill worker $w \in \mathcal{P}$, the time-bound virtual load aggregates per-request costs in its queue $\mathcal{Q}_w$: $\mathcal{U}_\mathcal{T}(w) = \sum_{r\in\mathcal{Q}_w}\big(\alpha\, l_{in}(r)^2 + \beta\, P_m\, l_{in}(r)\big)$. We interpret $\mathcal{U}_\mathcal{T}$ as a *FLOP-equivalent* proxy, and normalize it by the compute budget $\mathcal{C}_\mathcal{T} = \mathbb{P}_{peak}\, T_{SLO}$ so that $\mathcal{U}_\mathcal{T}/\mathcal{C}_\mathcal{T}$ approximates the fraction of the latency budget consumed by the queued work.

B.1 Peak Throughput Measurement ($\mathbb{P}_{peak}$)

Theoretical peak numbers can deviate from the effective throughput of an inference stack. We therefore use $\mathbb{P}_{peak}$ as an *effective peak throughput* measured on the target GPU under the same precision and kernel configuration as serving. Concretely, we run a GEMM-heavy microbenchmark representative of the prefill kernels and take the maximum sustained FP16 throughput as $\mathbb{P}_{peak}$. In our V100 setup, we observe $\mathbb{P}_{peak} \approx 121$ TFLOP (FP16, tensor-core enabled). We report the measured effective peak rather than vendor-only specifications to keep $\mathcal{U}_\mathcal{T}/\mathbb{P}_{peak}$ consistent with observed latency.

B.2 Coefficient Profiling (α, β)

The coefficients map request lengths to FLOP-equivalent costs under our serving stack.

Reference model and normalization. We use Qwen2.5-7B-Instruct (FP16) as the reference model M_{ref} and normalize the model-scale factor to it.

Measurement. We measure prefill latency $T_{prefill}(l_{in})$ for synthetic requests with $l_{in} \in [128, 8192]$ under a fixed batching policy to remove scheduler-induced variance.

FLOP-equivalent conversion and fitting. While α and β could be derived from analytical FLOPs, such derivations are highly sensitive to implementation details (e.g., fused kernels, attention variants, and batching) and may deviate from the latency behavior observed in an end-to-end serving stack. Since scheduling decisions ultimately aim to control user-visible latency (e.g., TTFT) and prevent congestion under bursts, we calibrate the time-bound model using the most directly relevant observable: prefill latency. We therefore convert measured prefill latency into a FLOP-equivalent target: $U(l_{in}) \triangleq T_{prefill}(l_{in}) \cdot \mathbb{P}_{peak}$. We then fit $U(l_{in}) \approx \alpha\, l_{in}^2 + \beta\, P_m(M_{ref})\, l_{in}$ with least-squares regression ($R^2 > 0.99$) to obtain stack-specific (α, β).

The fitted values used in our experiments are listed in Table 4.

B.3 Cross-Model Generalization of the Time-Bound Cost Model

To improve cross-model generalization without per-model profiling of the linear component, we introduce an explicit model-scale factor P_m extracted from static metadata.

Definition. For dense Transformer LLMs, we define

$$P_m(M) \triangleq \frac{N_{\text{param}}(M)}{N_{\text{param}}(M_{ref})}, \tag{3}$$

where $N_{\text{param}}(M)$ denotes the number of trainable parameters of model M obtained from model metadata. Thus, for our two evaluated models, P_m differs only by the parameter-scale ratio (e.g., 7B vs. 8B).

Scope and boundary. The above generalization targets *model changes under a fixed serving stack*. If the execution stack changes materially (e.g., precision, attention kernel, compiler/runtime), $\mathbb{P}_{peak}$ and/or (α, β) should be re-measured following §B.1–§B.2.

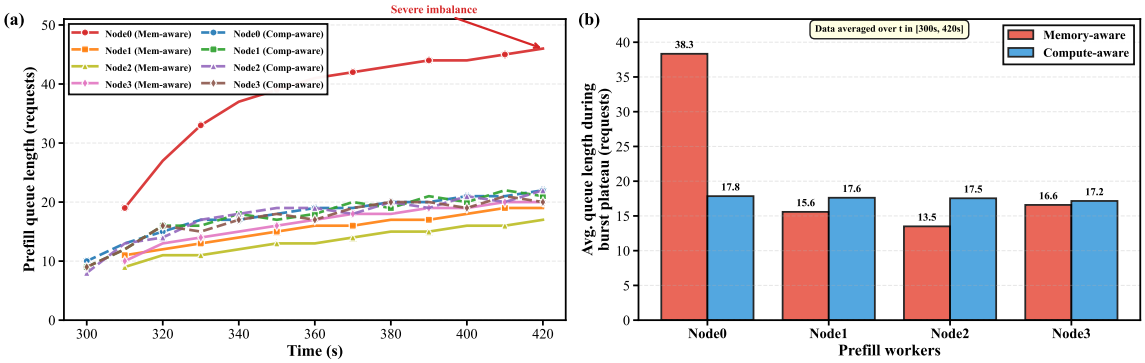

Figure 7: **Queue Divergence under Metric Misalignment.** Memory-aware routing piles requests on compute-saturated prefill workers, creating stragglers and amplifying head-of-line blocking; compute-aware routing maintains near-uniform queues.

B.4 Handling Negative Freeness

It is important to note that $\Phi(w)$ can become negative if the estimated virtual load exceeds the physical budget (i.e., $\mathcal{U}_{virt} > \mathcal{C}_{phys}$). Physically, a negative $\Phi(w) < 0$ signifies that the worker is currently overloaded relative to the strict SLO target.

Graceful Degradation: In such saturation regimes, the argmax operator in our Micro-scale Routing (Algorithm 1) remains valid and robust. By selecting the worker with the maximum algebraic value (i.e., the least negative Φ), Phasor naturally dispatches requests to the worker with the smallest overload margin.

Interpretation: This ensures that under extreme bursts where all workers violate SLOs, the system does not reject requests or deadlock; instead, it optimally distributes the excess load to minimize the aggregate magnitude of SLO violations, achieving graceful degradation.

B.5 Control Threshold Derivation ($\theta_{frag}, \theta_{flow}$)

Thresholds are chosen via sensitivity analysis to balance benefit and actuation overhead.

De-fragmentation trigger threshold (θ_{frag}). We sweep $\theta_{frag} \in [0.05, 0.30]$ on a decode-heavy workload from sharegpt and evaluate (i) OOM/allocation-failure incidence and (ii) background maintenance overhead (extra time/bandwidth consumed by AsyncDefrag). We select $\theta_{frag} = 0.168$ as a Pareto point that eliminates OOM events in our testbed while keeping defragmentation overhead below a small budget (under 2% in our measurements).

Flow control ratio (θ_{flow}). θ_{flow} provides hysteresis for macro-scale pool reconfiguration. We sweep θ_{flow} and measure (i) reconfiguration frequency (to detect flapping) and (ii) end-to-end capacity/latency improvement under drifting workloads. We choose $\theta_{flow} = 0.62$ as the smallest ratio that suppresses oscillatory reconfigurations while preserving most of the elasticity gains in our evaluation. In addition to this spatial hysteresis, we enforce a temporal Cool-down Period (T_{cool}) to strictly prevent rapid role flipping. A worker promoted from the Mixed Pool is locked in its new role for a minimum duration (set to $20 \times \Delta t$, i.e., 1 second) before it can be recalled. This dual-mechanism ensures that reconfiguration only responds to sustained load shifts rather than transient noise.

B.6 Control Epoch (Δt)

Δt sets the cadence of meso-/macro-scale control (e.g., de-fragmentation checks and pool rebalancing). It trades off responsiveness and control overhead: smaller Δt reacts faster but increases telemetry and background actuation frequency, while larger Δt reduces overhead but delays adaptation under bursts and workload drifts. We set $\Delta t = 50\,\text{ms}$ in our testbed, matching the natural meso-scale timescale of decode-side memory dynamics while remaining orders of magnitude larger than micro-scale request routing. We report a sensitivity sweep of Δt in Appendix F.

B.7 Latency Budget (T_{SLO})

We adopt the standard two-metric SLO formulation in LLM serving, where a request is considered *SLO-satisfied* only if it meets *both* a TTFT budget and a TPOT budget. In contrast, our time-bound freeness for *prefill* normalizes only by T_{SLO}^{TTFT}, since TTFT is determined by the prefill path, while TPOT is governed by the decode path.

Table 3: **Per-workload SLO budgets used in evaluation.** A request is counted as SLO-satisfied only if it meets *both* TTFT and TPOT budgets.

Workload ID	T_{SLO}^{TTFT} (s)	T_{SLO}^{TPOT} (s)
ShareGPT-S-S (Short QA)	0.30	0.15
ShareGPT-M-M (Dialog)	0.40	0.20
BurstGPT	0.40	0.20
ShareGPT-S-L (Code Gen.)	0.125	0.20
ShareGPT-L-S (Summarization)	2.25	0.13
ShareGPT-L-L (Translation)	3.00	0.18

SLO budgets are operator-configurable and workload-/tier-dependent in practice. Rather than claiming universal budgets, we align our per-scenario settings with widely-used tiers reported by prior serving systems (Zhong et al., 2024; Hong et al., 2025) and evaluations. Specifically, we map our synthetic archetypes to the closest prior workload categories (interactive chat, long-context reading, code completion) and use their published TTFT/TPOT targets as defaults. Table 3 summarizes the budgets used in our evaluation.

C Visualization of Control Mechanisms

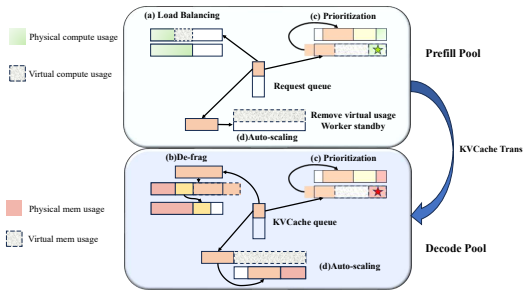

Figure 8: **Phasor Control Mechanisms.** Driven by Virtual Freeness, the system executes (a) Gradient-based Routing to balance compute, (b) Proactive Defragmentation to reclaim memory, and (c/d) Prioritization and Auto-scaling to manage capacity.

D Zero-Downtime KV Migration

Phasor introduces a multi-stage pipelining mechanism for KV migration. Instead of pausing execution, the system asynchronously replicates the generated KV Cache from the source to the target worker while the source continues inference. This approach effectively masks the data transfer latency, requiring only a negligible interruption for the final consistency check.

Specifically, to mitigate rescheduling overhead, Phasor employs an asynchronous, iterative migration strategy that pipelines state transfer with active decoding. As illustrated in Figure 9, the process exploits the append-only nature of the KV cache to minimize the critical section (downtime).

Initial Copy Phase (Stage 0): Upon triggering migration, the Source Worker continues decoding new tokens while asynchronously transferring the existing snapshot of KV blocks to the Destination Worker. This ensures the request makes progress during the bulk transfer.

Page Iteration (Stage 1...N): Since decoding continues during the initial phase, new "dirty" KV blocks are generated. Phasor enters an iterative convergence loop where it transfers only the incremental KV blocks generated in the previous stage. Because transfer speed typically exceeds generation speed, the volume of dirty data shrinks with each iteration.

Stop-and-Copy Phase: Once the remaining delta is small enough (Stage N), Phasor suspends execution for a final, atomic synchronization. This transfers the last few blocks and hands over control to the destination worker.

This design ensures that the Downtime is constant and independent of the total sequence length, limited only to the transfer time of the final iteration's output.

E Comprehensive Performance Panorama

While the main text focuses on extreme bottleneck regimes, Figure 10 illustrates our holistic evaluation across the full spectrum of workload archetypes, including Quick Q&A (Short-Short), Translation (Long-Long), and Dialog (Medium-Medium).

Analysis of Auxiliary Workloads.

1. Quick Q&A (Sharegpt-S-S): Low Overhead Validation.

Data Overview: This lightweight scenario ($L_{\text{in}}, L_{\text{out}} < 512$) serves as a litmus test for system overhead. Across all load levels ($\lambda = 5$ to 40 req/s), Phasor reduces the average Total Request P99 by 11.5% compared to Splitwise and 8.9% compared to DistServe.

Highlight: At peak load ($\lambda = 40$), Phasor achieves a Total Request P99 of 28.9 s, outperforming Splitwise (33.5 s) by 13.8% and DistServe (32.4 s) by 10.9%. The TTFT P99 is reduced

Table 4: Phasor Parameters in Our V100 (FP16) Testbed

Symbol	Definition	Value
	Physical / Stack Constants	
$\mathbb{P}_{peak}$	Measured effective peak throughput (FP16, serving stack)	≈ 121 TFLOPS
$\mathbb{M}_{total}$	Physical memory limit per decode worker	32 GB
α	Quadratic cost coefficient (attention proxy, FLOP-equivalent)	4.25×10^{-5}
β	Linear cost coefficient (FFN proxy, FLOP-equivalent; scaled by P_m)	6.80×10^{-3}
P_m	Model-scale factor	model-dependent
	Control Parameters	
Δt	Control epoch / sampling interval	50 ms
θ_{frag}	De-fragmentation trigger threshold	0.168
θ_{flow}	Flow control ratio (hysteresis for pool reconfiguration)	0.62

to 202 ms vs Splitwise's 215 ms and DistServe's 208 ms, confirming minimal scheduling overhead even for short tasks.

Attribution: These results contradict the intuition that sophisticated schedulers penalize lightweight requests. Phasor's asynchronous control plane incurs negligible overhead (<3% TTFT increase vs optimal) while effectively preventing micro-burst queuing through proactive load redistribution.

2. Translation (Sharegpt-L-L): Dual-Bottleneck Efficiency.

Data Overview: In this heavy regime where both prefill and decode phases saturate ($L_{\text{in}}, L_{\text{out}} \approx 2048$), Phasor demonstrates an average Decode Time Per Token P99 reduction of 11.8% versus Splitwise and 4.1% versus DistServe across all arrival rates. For end-to-end performance, the average Total Request P99 improves by 14.9% over Splitwise and 8.7% over DistServe.

Highlight: At saturation load ($\lambda = 40$), static partitioning collapses. DistServe degrades to a Total Request P99 of 175.9 s and Splitwise to 190.2 s, while Phasor suppresses this to 151.4 s—achieving a 20.4% reduction versus Splitwise and 14.0% versus DistServe. The Decode Time Per Token P99 is maintained at 118.7 ms, significantly outperforming Splitwise (138.9 ms, +17.0% slower) and DistServe (125.5 ms, +5.7% slower).

Attribution: When both stages are bottlenecked, rigid resource boundaries become pathological. Phasor's Mixed Pool enables elastic capacity reallocation, dynamically borrowing idle decode GPUs for prefill bursts. This fluidity mitigates the "hard edges" of static splits, preventing cascading queue buildup when the entire cluster is under stress.

3. Dialog (Sharegpt-M-M): Balanced Workload Robustness.

Data Overview: For standard balanced workloads ($L \approx 1024$), Phasor demonstrates the most pronounced superiority. Across all load levels, Phasor achieves an average Total Request P99 reduction of 25.6% compared to Splitwise and 19.6% compared to DistServe.

Highlight: At peak load ($\lambda = 40$), Phasor reduces Total Request P99 by 30.7% compared to Splitwise (102.5 s vs 148.0 s) and 19.9% compared to DistServe (102.5 s vs 128.0 s). This advantage emerges without extreme workload skew—neither stage is exclusively bottlenecked, yet static schedulers still suffer from suboptimal partitioning under load variance.

Attribution: This confirms that Phasor acts as a robust generalist. Even when workloads lack extreme input/output length skew, production traffic exhibits inherent burstiness and heterogeneity. Phasor's dimensionless interface unifies heterogeneous resource demands (compute-bound prefill vs memory-bound decode) into a common optimization space, enabling gradient-based balancing that static heuristics cannot achieve.

F Sensitivity Analysis

To evaluate the robustness of Phasor under varying hyper-parameter settings, we conduct a comprehensive sensitivity analysis on the BurstGPT workload. We examine four critical dimensions: the Cost Weight (α), the De-fragmentation Threshold (θ_{frag}), the Control Epoch (Δt), and the Flow Control Ratio (θ_{flow}). Table 5 summarizes the impact on P99 Latency, Capacity, and TTFT.

Robustness Observations. Cost Weight (α): The system exhibits asymmetric sensitivity. Underestimating α ($0.5\times$) is detrimental, causing con-

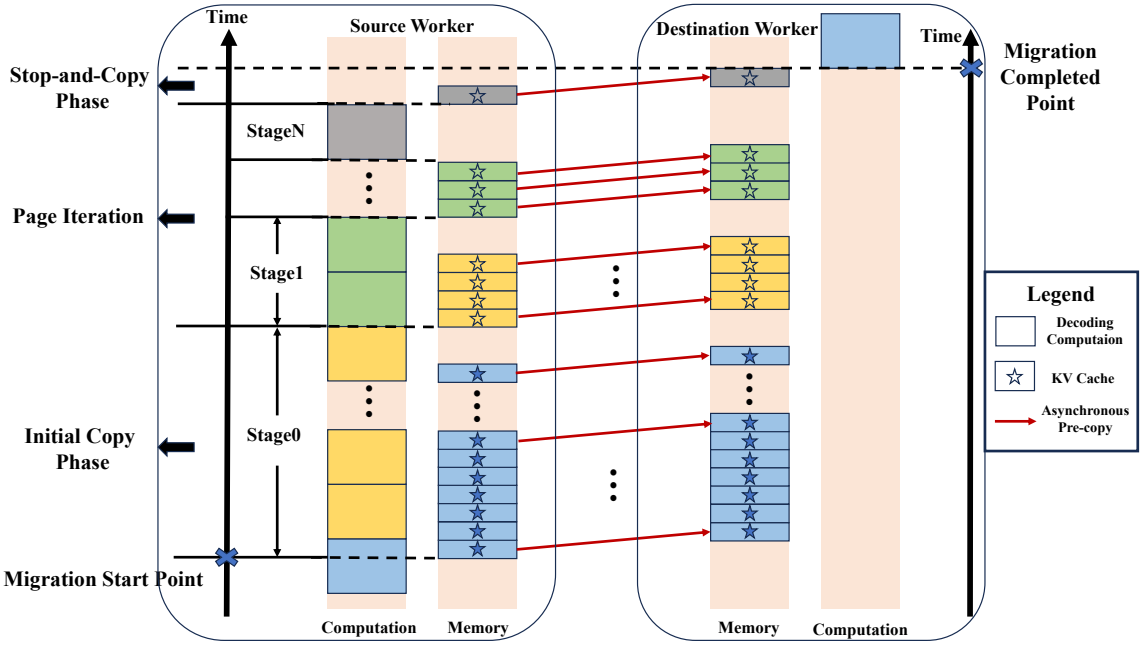

Figure 9: **Phasor's Asynchronous Pipelined Migration Workflow.** The migration is divided into the *Initial Copy Phase* (bulk background transfer), *Page Iteration* (incremental state convergence), and a minimal *Stop-and-Copy Phase* (atomic handover). By pipelining KV transfer with decoding computation on the Source Worker, Phasor achieves near-zero downtime regardless of the context length.

gestion and a 14% latency spike. Conversely, overestimating α (2.0×) makes the scheduler conservative: it maintains strict latency compliance (P99 $\approx$ 0.99) by actively rejecting risky batches. However, this safety comes at a cost, effectively "wasting" usable headroom and leading to a 6% drop in serviceable capacity.

De-fragmentation Threshold (θ_{frag}): We observe a saturation effect. While a low threshold (0.10) allows fragmentation to degrade latency (+9%), raising the threshold beyond the default (0.35) yields diminishing returns. It does not further improve latency (as blocking is already eliminated) but incurs slightly higher migration overhead, reducing capacity by 2%.

Control Epoch (Δt): Phasor demonstrates remarkable robustness to the telemetry interval. Increasing Δt from 50ms to 200ms only degrades performance by 6–8%, designated as "Low" impact. This validates that the predictive nature of Virtual Freeness reduces the reliance on high-frequency hardware feedback.

Flow Control Ratio (θ_{flow}): The system remains stable across a wide range of flow ratios. Because θ_{flow} works in tandem with the temporal Cooldown Period (Appendix B.4), the Mixed Pool resists oscillation even when the ratio is loose (0.50) or strict (0.80), showing only marginal impact on end-to-end metrics.

G Energy Efficiency Analysis

In the context of LLM efficiency, energy consumption is the physical denominator of utilization. We therefore test whether Phasor's throughput gains translate into better *work per joule*, rather than merely higher power draw.

Methodology. We monitor *aggregate GPU power* (sum over 8 GPUs) via `nvidia-smi`/NVML at 100 ms intervals and evaluate the *Production Mix* (BurstGPT) at the peak-load phase (40–60 req/s), excluding warm-up. We define **Energy Efficiency** as *Joules per generated output token* (J/token), computed as $\overline{P_{\text{GPU}}}/\overline{\text{tok/s}}$, which is equivalent to $\int P(t)\,dt \,/\, \#\text{generated tokens}$ over the same window. We use *output-token goodput* (successfully generated output tokens/s) to reflect end-to-end generation progress, which is most sensitive to decode-side stalls and fragmentation-induced effective-capacity loss.

Sanity Check: Utilization vs. Power. As shown in Table 6, Phasor draws higher average GPU power (∼1720 W) than Splitwise (∼1315 W), a 31% increase. This is consistent with Phasor mitigating pipeline starvation: by sustaining higher

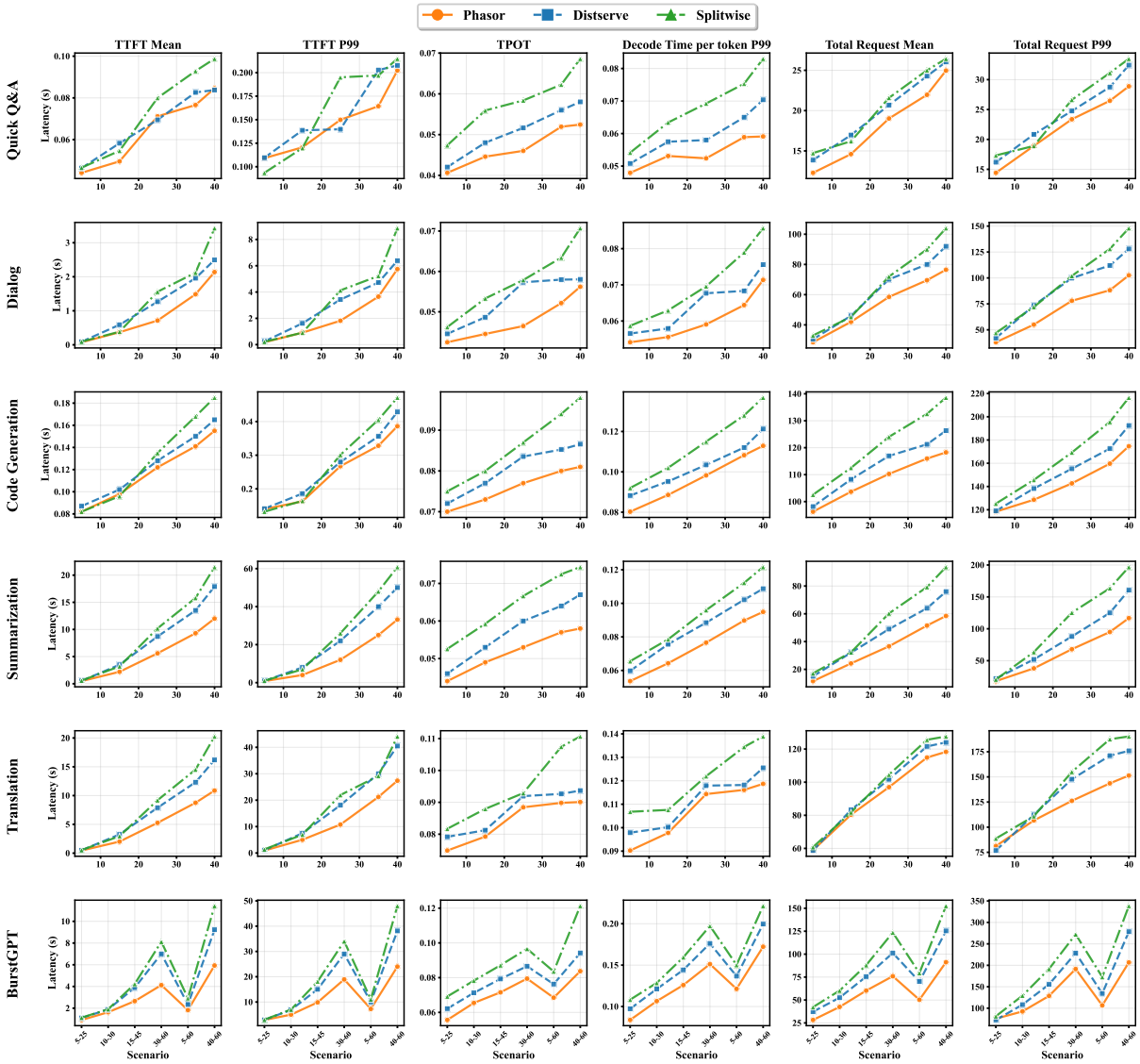

Figure 10: **Comprehensive Performance Panorama.** End-to-end latency metrics across six workload archetypes, covering Quick Q&A (S-S), Dialog (M-M), Code Generation (S-L), Summarization (L-S), Translation (L-L), and Production Mix (BurstGPT). Phasor demonstrates universal advantages in TTFT, TPOT, and total request latency across all load levels.

effective decode concurrency, it converts idle GPU power into active generation work.

Conclusion. Phasor improves energy proportionality by translating higher utilization into lower GPU energy per generated token, rather than merely increasing power draw.

Dimension	Parameter Variation	P99 Latency (↓)	Capacity (↑)	TTFT P99 (↓)	Sensitivity
Cost Weight (α)	$0.5 \times \alpha_{def}$	1.14	0.92	1.18	Moderate
	$2.0 \times \alpha_{def}$	**0.99**	0.94	1.00	Low
De-frag Threshold (θ_{frag})	Low (0.10)	1.09	0.96	1.12	Moderate
	High (0.35)	1.00	0.98	1.00	Low
Control Epoch (Δt)	Fast (20 ms)	0.99	1.01	0.99	Negligible
	Slow (200 ms)	1.06	0.97	1.08	Low
Flow Ratio (θ_{flow})	Loose (0.50)	1.02	0.99	1.01	Low
	Strict (0.80)	1.04	0.95	1.03	Low

Table 5: Sensitivity Analysis on BurstGPT. Metrics are normalized to the default configuration (α_{def}, $\theta_{frag} = 0.168$, $\Delta t = 50$ms, $\theta_{flow} = 0.62$). "Sensitivity" denotes the system's responsiveness to parameter drift.

System	Avg. Power (W)	Gen. Throughput (tok/s)	Efficiency (J/Token)
Splitwise	∼1,315	∼5,422	0.243
DistServe	∼1,410	∼7,009	0.201
Phasor	**∼1,720**	**∼9,820**	**0.175**

Table 6: Energy Efficiency on BurstGPT (8×V100). Phasor converts "idle power" into "active work". Despite a higher raw power draw, the substantial gain in throughput results in a **28.0% reduction in energy cost per token**.