Mix-or-Split: Latency-Aware Scheduling for Edge–Cloud LLM Inference

Xinghan Wang, Xiaoxiong Zhong, *Member, IEEE*, Weihong Yang, Fangming Liu, *Senior Member, IEEE*, and Weizhe Zhang, *Senior Member, IEEE*

***Abstract*—In this paper, we present a latency-aware scheduler for large-language-model (LLM) inference across mobile devices, edge servers, and a remote cloud. Our fine-grained delay model captures OFDMA uplink/downlink rates, KV-cache backhaul serialization, and profiled GPU planning-chunk resource constraints, enabling per-request routing among (i) cloud-only execution, (ii) cloud-prefill + edge-decode split execution, and (iii) edge prefill–decode mix execution. The scheduler targets high goodput and SLO attainment under hard deadlines, TTFT/TPOT targets, and guarded SM-resource constraints via a four-stage loop: windowed admission, δ-similar prompt bucketing, tile selection, and SM-feasible prefill/decode allocation with dynamic execution adjustment. We bound padding waste, prefill-latency standard deviation, and the number of buckets, and analyze how the feasible allocation frontier is shaped by active register, SMEM, and warp walls under high-bandwidth, low-load conditions. On an A800/A6000 heterogeneous testbed, repeated real-serving experiments exercise all three routing decisions. In a Qwen3-8B heterogeneous replay, the online policy lowers mean end-to-end latency by 6.4% relative to the best fixed mean baseline and attains 100% SLO, while its P95 confidence interval overlaps those of fixed Cloud and edge-local execution. In a heterogeneous Llama-3 replay, the policy selects Cloud/Mix/Split for 18/21/6 requests; it attains 100% SLO across the four primary workload regimes and 95.6--100% across the offered-load points. Real CUDA-launch analysis further shows that the dominant Prefill/Decode kernel pair fails the joint-residency resource bound for all 37 Llama shapes on both GPUs; guarded alternative pairs account for 41.4% and 26.9% of the analyzed pair weight on A800 and A6000, respectively. These measurements support profile-guided admission and fallback rather than a blanket claim that concurrent Mix is always beneficial.**

***Index Terms*—Large language models, edge–cloud inference, latency-aware scheduling, KV-cache transfer, GPU resource allocation.**

I. Introduction

Large language model (LLM) inference is increasingly shaped by the asymmetry between its two execution phases. Prefill processes the entire input prompt and is compute intensive, while decode generates tokens one by one and is dominated by KV-cache traffic, memory bandwidth, and per-token latency [1]–[8]. Existing serving systems improve this two-phase execution through batching, KV-cache memory management, SLO-aware scheduling, and prefill/decode overlap [9]–[12]. These techniques are highly effective once the serving location and resource pool are fixed. In mobile edge–cloud inference, however, the first decision is more fundamental: for each request, should prefill and decode be co-located at the edge, split across cloud and edge, or executed entirely in the cloud?

This decision is not a conventional binary offloading problem. A cloud-only route avoids edge-GPU pressure but exposes the request to cloud-side queueing and a longer access path through either direct UE–cloud connectivity or AP/edge–cloud forwarding. A split route can move the compute-heavy prefill phase to the cloud while keeping latency-sensitive decode near the user, but it introduces a KV-cache transfer whose size grows with prompt length, model depth, KV-head count, and quantization width. A mix route avoids the KV backhaul and keeps both phases close to the user, but it is feasible only when the profiled prefill/decode chunk allocation is SM-feasible under register, shared-memory, and warp constraints. Therefore, mode selection must jointly reason about wireless rate, KV-cache serialization, and SM-level GPU feasibility.

This paper studies *Network–KV–SM coupled mode selection* for mobile edge–cloud LLM inference. Mix-or-Split inserts request-level mode selection before batching and kernel dispatch, using measured network rate, KV serialization cost, deadline slack, and profiled SM-resource feasibility to choose among the three routes. Deployment is therefore a run-time decision rather than a fixed configuration.

The core challenge is that the best mode changes across requests and over time [13]. Long prompts may make edge prefill infeasible or inefficient, but they also enlarge the KV tensor that split execution must send back to the edge. Short prompts may benefit from edge co-location when edge resources are available. Wireless saturation can make every route inadmissible, whereas edge SM saturation can shift the feasible choice toward Split or Cloud. Bursty arrivals further amplify the problem because a mode that is feasible for an isolated request may violate TTFT, TPOT, or hard-deadline constraints once it is placed into a real scheduling window. We address this by combining request-level mode selection with window-level admission, δ-similar prompt bucketing, tile selection, and wave-aligned next-dispatch adjustment.

Our main contributions are summarized as follows:

- We formulate Network–KV–SM coupled mode selection for mobile edge–cloud LLM inference. The formulation elevates cloud-only, split, and edge-mix execution to per-

Xinghan Wang, Xiaoxiong Zhong, Weihong Yang, Fangming Liu and Weizhe Zhang are with Peng Cheng Laboratory, Shenzhen 518000, P. R. China. (Email: csxhwang@gmail.com; zhongxx@pcl.ac.cn; yanghw01@pcl.ac.cn; liufm@pcl.ac.cn; weizhe.zhang@pcl.ac.cn.). *Corresponding author:* Xiaoxiong Zhong. This work was supported in part by the Major Key Project of Pengcheng Laboratory under Grant PCL2025A13; in part by the Major Key Project of Pengcheng Laboratory under Grants PCL2025A10 and PCL2024A06; and in part by the Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

request competing modes rather than fixed deployment choices.
- We build a three-mode admissibility model that explicitly accounts for wireless upload/download, cloud/edge queueing, cloud prefill/decode, edge prefill/decode, KV-cache backhaul serialization, TTFT, TPOT, and hard deadlines. This model identifies when split is defeated by KV transfer and when cloud-only should be preferred.
- We develop a profiled SM-resource feasibility model for edge-mix execution. By mapping prefill/decode tiles to register, SMEM, warp, and latency footprints, Mix-or-Split determines whether the profiled prefill/decode chunk allocation is SM-feasible before admitting a request into mix mode.
- We specify a hierarchical online scheduler that combines δ-similar prompt bucketing, violation-aware token/request budgets, and wave-aligned next-dispatch adjustment. We derive bounds on padding waste, Prefill-latency dispersion under a stated profile model, and bucket count; the controller changes future kernel waves while leaving in-flight work intact.

The rest of the paper is organized as follows: Section II introduces related work. The system model and metrics are outlined in Section III. Section IV details the algorithm workflow and theoretical analysis. Performance evaluation is presented in Section V, followed by conclusions in Section VI.

II. Related Work

A. LLM Serving and GPU Scheduling

Modern LLM serving systems improve throughput and latency through memory management, batching, and SLO-aware scheduling. PagedAttention and vLLM reduce KV-cache fragmentation and improve batched serving efficiency through virtualized KV memory management [9]. State-aware and SLO-aware schedulers further decide which requests should be served under heterogeneous latency targets [11], [12]. At the GPU-kernel level, recent work also studies prefill–decode overlap and low-level kernel optimization; for example, POD-Attention unlocks prefill/decode overlap within LLM serving [10], while SpInfer exploits low-level sparsity for efficient GPU inference [14]. These systems are important baselines for datacenter LLM serving, but they primarily optimize request batching, KV-cache management, or intra-GPU overlap. They do not jointly model mobile wireless links, edge–cloud routing, KV backhaul serialization, and profiled SM-level feasibility under edge SLOs.

B. Edge Inference for LLMs

Edge LLM inference leverages nearby devices and edge servers to reduce response latency, operating cost, and privacy exposure [15]–[19]. LLM-in-a-Flash constructs an inference cost model for devices whose DRAM cannot hold the full model, reducing flash transfers and reading larger contiguous chunks to improve inference efficiency [20]. Ripple accelerates smartphone inference through correlation-aware neuron placement and runtime caching [21]. Agile-Quant uses activation-guided quantization and token pruning to trade accuracy for real inference speed on edge devices [22]–[26]. SyncIntellects predicts response length and applies QoS-friendly length control for LLM inference orchestration [27]. Other work improves throughput-oriented inference by coordinating CPU and GPU resources on general-purpose hardware [28]. These approaches mainly reduce the local execution cost of LLMs, whereas Mix-or-Split addresses when a request should remain on the edge, be split across cloud and edge, or be executed entirely in the cloud.

C. Cloud-Edge Collaborative Inference for LLMs

Cloud-edge collaborative inference exploits both local personalization and cloud-side capacity [29]. Crayon allocates demanding or non-customized queries to a server-side LLM while preserving on-device customization [30]. PerLLM schedules diverse LLM services with edge-cloud collaboration and energy-aware resource allocation [31]. SplitLLM studies model placement and collaborative inference for transformer-based architectures by jointly considering computation and communication resources [32]. Historical interaction retrieval can also enhance on-device LLM inference by selecting useful cloud-derived data within a local storage budget [33]. These studies motivate edge-cloud cooperation, but most of them treat the accelerator as a coarse compute resource and do not expose profiled SM-level register, SMEM, warp, and CTA-slot constraints. Our work fills this gap by coupling edge-cloud mode selection with wireless feasibility, KV backhaul cost, and profiled prefill/decode allocation feasibility.

D. Positioning of Mix-or-Split

Table I summarizes the distinction against representative state-of-the-art serving systems. vLLM and SGLang improve memory management, prefix/KV reuse, batching, and structured generation inside a serving runtime [9], [34]. Sarathi uses chunked prefill to construct decode-friendly batches [35]. DistServe disaggregates prefill and decode across GPUs within a serving cluster [36]. POD-Attention and NanoFlow further exploit prefill/decode overlap or intra-device operation-level scheduling [10], [37]. These systems are important and complementary, but they operate after the serving site or deployment pattern has largely been fixed. Mix-or-Split targets the preceding decision layer: before a request is served, it chooses among cloud-only, split, and edge-mix execution using wireless feasibility, KV-cache transfer cost, deadline slack, and guarded SM-level resource frontiers.

III. Model and Metrics

In this work, we consider a three-tier inference architecture that spans mobile users, edge servers, and a remote cloud server. Let $\mathcal{U}$ denote the user equipment devices (UEs), indexed by $u \in \{1, \ldots, U\}$. During each scheduling epoch, every UE is associated with exactly one of A radio access points (APs), collected in the set $\mathcal{A} = \{1, \ldots, A\}$. The association can be fixed by the access network or selected from a safe candidate set; the variable $y_u^e(t)$ introduced below represents the resulting access–edge domain for that epoch.

TABLE I
POSITIONING OF MIX-OR-SPLIT AGAINST REPRESENTATIVE LLM SERVING SYSTEMS.

System	Main decision	Wireless	KV backhaul	SM-level feasible?
vLLM/SGLang [9], [34]	Batching / KV reuse	No	No	No
Sarathi [35]	Chunked prefill	No	No	No
DistServe [36]	PD disaggregation	No	Intra-cluster	No
POD-Attention [10]	PD overlap	No	No	Partial
NanoFlow [37]	Op-level pipeline	No	No	Intra-device
Edge/split inference [30]–[32]	Coarse placement	Partial	Partial	No
Mix-or-Split	Per-request mode	Yes	Yes	Yes

Each user equipment $u \in \mathcal{U}$ issues an ordered sequence of inference requests $r_{u,i} \in \mathcal{R}_u = \{r_{u,1}, r_{u,2}, \ldots, r_{u,I}\}$, where each request is represented as

$$r_{u,i} = \left(a_{u,i}, b^{\mathrm{in}}_{u,i}, b^{\mathrm{out}}_{u,i}, L^{\mathrm{in}}_{u,i}, L^{\mathrm{out}}_{u,i}, D^{\mathrm{FT}}_{u,i}, D^{\mathrm{PT}}_{u,i}, \tau_{u,i}\right).$$

Here, $a_{u,i}$ is the request arrival time; $L^{\mathrm{in}}_{u,i}$ and $L^{\mathrm{out}}_{u,i}$ denote the input-prompt and maximum output lengths in tokens, respectively, with $L^{\mathrm{in}}_{u,i}, L^{\mathrm{out}}_{u,i} \in \mathbb{Z}_{\geq 1}$; $b^{\mathrm{in}}_{u,i}$ and $b^{\mathrm{out}}_{u,i}$ are the uplink prompt payload and downlink response payload in bits; $D^{\mathrm{FT}}_{u,i}$ and $D^{\mathrm{PT}}_{u,i}$ are the service-level objectives (SLOs) for the time-to-first-token (TTFT) and time-per-output-token (TPOT), respectively; and $\tau_{u,i}$ is the hard end-to-end latency deadline relative to the request arrival time. When the response payload is not measured directly, we estimate it as $b^{\mathrm{out}}_{u,i} = \eta_{\mathrm{tok}} L^{\mathrm{out}}_{u,i}$, where η_{tok} is the average number of returned bits per generated token after token serialization and protocol framing.

Each edge server is indexed by $e \in \mathcal{E} = \{1, \ldots, E\}$ and abstracted as a triple $\left(C_e, B^U_e, B^D_e\right)$. Here, C_e denotes the sustained compute capability of the on-board GPU. C_e can be further refined as the vector $(N_{\mathrm{sm}}, C_s, R_s, S_s, W_s, T_s)$ representing the number of SMs and the per-SM CTA-slot, register, shared-memory, warp, and thread limits, respectively. The pair $\left(B^U_e, B^D_e\right)$ captures the asymmetric wireless bandwidth between the edge and its associated users: B^U_e is the uplink bandwidth from the UE to the edge server, whereas B^D_e is the downlink bandwidth from the edge server to the UE.

The remote cloud server is modeled as a single logical node C that hosts the full-size LLM and executes the compute-intensive prefill stage for split execution (cloud-prefill + edge-decode) and both prefill and decode for cloud-only execution. The cloud node is parameterized by a GPU profile containing the number of SMs and the per-SM CTA-slot, register, SMEM, warp, and thread budgets; Section V instantiates this profile on our experimental platform. The cloud connects to all edge domains via a wired backhaul with bandwidth B^{EC}, measured in bits per second, and one-way transport delays $d^{\mathrm{EC}}_{E\to C}$ and $d^{\mathrm{EC}}_{C\to E}$. Unlike propagation delay, the serialization time of KV-cache tensors can be non-negligible for long prompts and large models, so it is modeled explicitly below.

A. Communication Model

Communication latency consists of wireless prompt upload, wireless output delivery, cloud-side prompt/output forwarding for cloud-only or split routing, and, in split mode, edge–cloud KV-cache backhaul. Let $\widehat{R}^{\mathrm{UL}}_{u,e}$ and $\widehat{R}^{\mathrm{DL}}_{u,e}$ denote the effective uplink and downlink rates used by the scheduler. In a real deployment these rates are measured online. For an OFDMA instantiation, let $\Gamma^{\mathrm{UL}}_{u,e}$ and $\Gamma^{\mathrm{DL}}_{u,e}$ be the measured post-allocation SINRs on a representative resource block during the current epoch. Equivalently, under fixed per-resource-block powers they can be written as $P_u|h^{\mathrm{UL}}_{u,e}|^2/N^{\mathrm{UL}}_0$ and $P^{\mathrm{DL}}_e|h^{\mathrm{DL}}_{u,e}|^2/N^{\mathrm{DL}}_0$, respectively. This per-resource-block convention keeps the measured spectral efficiency fixed while the scheduler changes the number of assigned resource blocks. During a scheduling epoch, the OFDMA scheduler allocates bandwidth fractions $\alpha^{\mathrm{UL}}_{u,e}$ and $\alpha^{\mathrm{DL}}_{u,e}$, subject to

$$\sum_{u\in\mathcal{U}_e} \alpha^{\mathrm{UL}}_{u,e} \leq 1, \qquad \sum_{u\in\mathcal{U}_e} \alpha^{\mathrm{DL}}_{u,e} \leq 1. \tag{1}$$

Under this measured-SINR convention, the instantaneous rates are

$$\widehat{R}^{\mathrm{UL}}_{u,e} = \alpha^{\mathrm{UL}}_{u,e} B^U_e \log_2\left(1+\Gamma^{\mathrm{UL}}_{u,e}\right), \tag{2}$$

$$\widehat{R}^{\mathrm{DL}}_{u,e} = \alpha^{\mathrm{DL}}_{u,e} B^D_e \log_2\left(1+\Gamma^{\mathrm{DL}}_{u,e}\right). \tag{3}$$

The corresponding upload and downlink transmission times are

$$T^{\mathrm{edge}}_{\mathrm{up}}(u,i,e) = \frac{b^{\mathrm{in}}_{u,i}}{\widehat{R}^{\mathrm{UL}}_{u,e}}, \qquad T^{\mathrm{edge}}_{\mathrm{down}}(u,i,e) = \frac{b^{\mathrm{out}}_{u,i}}{\widehat{R}^{\mathrm{DL}}_{u,e}}. \tag{4}$$

Let $b^{(1)}_{u,i}$ denote the serialized first-token payload. The first-token and remaining-token downlink times on the edge path are

$$\begin{aligned} T^{\mathrm{edge}}_{\mathrm{down},1}(u,i,e) &= \frac{b^{(1)}_{u,i}}{\widehat{R}^{\mathrm{DL}}_{u,e}}, \\ T^{\mathrm{edge}}_{\mathrm{down,rem}} &= \max\left\{T^{\mathrm{edge}}_{\mathrm{down}} - T^{\mathrm{edge}}_{\mathrm{down},1}, 0\right\}. \end{aligned} \tag{5}$$

For split and cloud-only routing, the UE first reaches the serving AP/edge domain and then traverses the wired edge–cloud path. We use the following conservative store-and-forward estimates; a deployment that pipelines wireless and backhaul serialization can replace their sums by measured end-to-end transfer times:

$$\begin{aligned} T^{\mathrm{cloud}}_{\mathrm{up}}(u,i,e) &= T^{\mathrm{edge}}_{\mathrm{up}}(u,i,e) + \frac{b^{\mathrm{in}}_{u,i}}{B^{\mathrm{EC}}} + d^{\mathrm{EC}}_{E\to C}, \\ T^{\mathrm{cloud}}_{\mathrm{down}}(u,i,e) &= T^{\mathrm{edge}}_{\mathrm{down}}(u,i,e) + \frac{b^{\mathrm{out}}_{u,i}}{B^{\mathrm{EC}}} + d^{\mathrm{EC}}_{C\to E}. \end{aligned} \tag{6}$$

The corresponding first-token cloud downlink and remaining-token cloud downlink are

$$\begin{aligned} T^{\mathrm{cloud}}_{\mathrm{down},1}(u,i,e) &= T^{\mathrm{edge}}_{\mathrm{down},1}(u,i,e) + \frac{b^{(1)}_{u,i}}{B^{\mathrm{EC}}} + d^{\mathrm{EC}}_{C\to E}, \\ T^{\mathrm{cloud}}_{\mathrm{down,rem}} &= \max\left\{T^{\mathrm{cloud}}_{\mathrm{down}} - T^{\mathrm{cloud}}_{\mathrm{down},1}, 0\right\}. \end{aligned} \tag{7}$$

The one-way delay terms capture propagation and transport latency on the wired edge–cloud path; if direct UE–cloud access is available, the same symbols can be replaced by

the measured direct-path upload and downlink latencies. For positive wireless-only upload and downlink budgets $D^{\mathrm{UL}}_{u,i} > 0$ and $D^{\mathrm{DL}}_{u,i} > 0$, where $D^{\mathrm{UL}}_{u,i}+D^{\mathrm{DL}}_{u,i}$ is assigned from the request-level TTFT/deadline slack, the minimum required bandwidth fractions are

$$\alpha^{\mathrm{UL}}_{u,e,\min} = \frac{b^{\mathrm{in}}_{u,i}}{D^{\mathrm{UL}}_{u,i} B^{U}_{e} \log_2\left(1+\Gamma^{\mathrm{UL}}_{u,e}\right)}, \tag{8}$$

$$\alpha^{\mathrm{DL}}_{u,e,\min} = \frac{b^{\mathrm{out}}_{u,i}}{D^{\mathrm{DL}}_{u,i} B^{D}_{e} \log_2\left(1+\Gamma^{\mathrm{DL}}_{u,e}\right)}. \tag{9}$$

When split execution is selected, the cloud sends the generated KV cache for the prompt to the edge before decode. For a model with N_ℓ transformer layers, H_{kv} KV heads, head dimension d_h, and q_{kv} bytes per KV element, the serialized KV size in bits for a prompt of length $L^{\mathrm{in}}_{u,i}$ is

$$S^{\mathrm{bit}}_{\mathrm{kv}}(L^{\mathrm{in}}_{u,i}) = 8 \cdot 2 N_\ell L^{\mathrm{in}}_{u,i} H_{\mathrm{kv}} d_h q_{\mathrm{kv}}, \tag{10}$$

where the factor 2 accounts for keys and values and the factor 8 converts bytes to bits. Let $s_{u,i} = S^{\mathrm{bit}}_{\mathrm{kv}}(L^{\mathrm{in}}_{u,i})$. The conservative end-to-end KV-migration estimate is

$$\begin{aligned}\widehat{T}^{\mathrm{e2e}}_{\mathrm{kv}}(u,i) = {} & T_{\mathrm{D2H}}(s_{u,i}) + T_{\mathrm{ser}}(s_{u,i}) + \frac{s_{u,i}}{B^{\mathrm{EC}}} + d^{\mathrm{EC}}_{C\to E} \\ & + T_{\mathrm{deser}}(s_{u,i}) + T_{\mathrm{H2D}}(s_{u,i}) + T_{\mathrm{rpc}}(s_{u,i}),\end{aligned} \tag{11}$$

where the size-dependent host/device-copy, serialization, loading, and RPC functions are measured for the selected transfer implementation. Terms that overlap in a deployment may be replaced by a directly measured migration latency; Eq. (11) is the non-overlap upper bound used for admission.

B. Mode-Selection and Routing Procedure

When a UE u issues a new request $r_{u,i} = \left(a_{u,i}, b^{\mathrm{in}}_{u,i}, b^{\mathrm{out}}_{u,i}, L^{\mathrm{in}}_{u,i}, L^{\mathrm{out}}_{u,i}, D^{\mathrm{FT}}_{u,i}, D^{\mathrm{PT}}_{u,i}, \tau_{u,i}\right)$, it first sends a one-hop control packet to its serving access point (AP). The AP immediately forwards only the metadata to the cloud scheduler; the prompt payload itself remains at the UE until the scheduler returns a routing decision. This is a control-plane decision: only metadata is sent before routing, so the scheduler does not incur prompt-payload transfer before selecting the mode; measured control-plane processing is included in the profiled queue/RPC terms. Let $\mathcal{E}_u(t)$ be the access–edge domains currently reachable by UE u (a singleton when association is fixed). The scheduler first forms the set of domains with an available wireless access path:

$$\begin{aligned}\mathcal{E}^{\mathrm{avail}}_{u}(t) = \{e \in \mathcal{E}_u(t) \mid {} & \widehat{R}^{\mathrm{UL}}_{u,e}(t) > 0, \\ & \widehat{R}^{\mathrm{DL}}_{u,e}(t) > 0\},\end{aligned} \tag{12}$$

For every available domain e, the scheduler evaluates profiled latency predictors for $m \in \mathcal{M} = \{\textsc{Mix}, \textsc{Split}, \textsc{Cloud}\}$. Let $\chi^{m}_{u,i}(e,t) \in \{0,1\}$ indicate that all non-latency requirements of mode m are available, including the required service, memory, backhaul, edge-decode capacity, and guarded SM feasibility. The per-edge feasible set is

$$\begin{aligned}\mathcal{M}^{\mathrm{feas}}_{u,i}(e,t) = \{m \in \mathcal{M} \mid {} & \chi^{m}_{u,i}(e,t) = 1, \\ & \widehat{T}^{\mathrm{FT},m}_{u,i}(e,t) \le D^{\mathrm{FT}}_{u,i}, \\ & \widehat{T}^{\mathrm{PT},m}_{u,i}(e,t) \le D^{\mathrm{PT}}_{u,i}, \\ & \widehat{T}^{m}_{u,i}(e,t) \le \tau_{u,i}\},\end{aligned} \tag{13}$$

and the scheduler jointly selects the edge and minimum-latency feasible mode:

$$(e^*_{u,i}, m^*_{u,i}) = \underset{\substack{e\in\mathcal{E}^{\mathrm{avail}}_{u}(t),\\ m\in\mathcal{M}^{\mathrm{feas}}_{u,i}(e,t)}}{\arg\min} \widehat{T}^{m}_{u,i}(e,t). \tag{14}$$

If no feasible edge–mode pair exists, the request is rejected or delayed to the next admission window. Ties are broken by larger guarded residual capacity and then by a fixed mode order. The resulting decision directs the UE either to upload its prompt directly to $e^*_{u,i}$ (mixed) or through the access domain to the cloud (split/cloud-only).

The mode predictors below use a single token-accounting convention. Full-prompt prefill produces the logits and sampled value of output token 1; Decode then generates the remaining $\max\{L^{\mathrm{out}}_{u,i} - 1, 0\}$ tokens autoregressively. Accordingly, $T^{z}_{\mathrm{decode,rem}}$ denotes the total sequential Decode time for these non-first tokens at site $z \in \{\mathrm{edge}, \mathrm{cloud}\}$. The communication-and-compute sums are conservative non-overlap upper bounds used for admission; directly measured predictors can replace them when execution and delivery are pipelined.

Mix mode. In prefill–decode mix mode, the entire inference workload is co-located on the selected edge server $e^*_{u,i}$, and a request progresses through four tightly coupled stages. (i) Prompt upload: the UE transfers the prompt ($b^{\mathrm{in}}_{u,i}$ bits) over the uplink of edge server $e^*_{u,i}$; its epoch-level transmission estimate is $T^{\mathrm{edge}}_{\mathrm{up}}$. (ii) Batch admission: upon arrival the request enters a sliding-window queue of width W; the PrefillMerge policy bundles all prompts that still satisfy the guarded SM-resource constraints, yielding a queueing delay $T^{\mathrm{edge}}_{\mathrm{batch}}$. (iii) Kernel configuration: For the resulting minibatch, the scheduler consults a profiled tile catalogue and selects a predicted prefill/decode allocation for the next dispatch wave. The configuration is selected from the guarded, SLO-admissible candidates by the profiled batch utility in Eq. (54), which trades predicted goodput against TTFT/TPOT risk. (iv) Execution and delivery: The selected profiled configuration executes in $T^{\mathrm{mix}}_{\mathrm{exe}} = T^{\mathrm{edge}}_{\mathrm{prefill}} + T^{\mathrm{edge}}_{\mathrm{decode,rem}}$; the generated token stream is then forwarded to the UE in $T^{\mathrm{edge}}_{\mathrm{down}}$. Consequently, the end-to-end, TTFT, and TPOT terms for mix mode are

$$\begin{aligned}\widehat{T}^{\mathrm{mix}}_{u,i} = {} & T^{\mathrm{edge}}_{\mathrm{up}} + T^{\mathrm{edge}}_{\mathrm{batch}} + T^{\mathrm{edge}}_{\mathrm{prefill}} \\ & + T^{\mathrm{edge}}_{\mathrm{decode,rem}} + T^{\mathrm{edge}}_{\mathrm{down}},\end{aligned} \tag{15}$$

$$\begin{aligned}\widehat{T}^{\mathrm{FT,mix}}_{u,i} = {} & T^{\mathrm{edge}}_{\mathrm{up}} + T^{\mathrm{edge}}_{\mathrm{batch}} \\ & + T^{\mathrm{edge}}_{\mathrm{prefill}} + T^{\mathrm{edge}}_{\mathrm{down},1},\end{aligned} \tag{16}$$

$$\widehat{T}^{\mathrm{PT,mix}}_{u,i} = \begin{cases} \dfrac{T^{\mathrm{edge}}_{\mathrm{decode,rem}} + T^{\mathrm{edge}}_{\mathrm{down,rem}}}{L^{\mathrm{out}}_{u,i} - 1}, & L^{\mathrm{out}}_{u,i} > 1, \\ 0, & L^{\mathrm{out}}_{u,i} = 1. \end{cases} \tag{17}$$

Concrete execution configurations and profiled allocation estimates come from the profiled catalogue described in Section IV.

Split mode. In the prefill–decode split mode, the prompt is processed in two geographically separated stages: prefill on the cloud server GPU and token-by-token decode on the selected edge server $e^*_{u,i}$. A request advances through: (i) Prompt upload: First, the prompt is uploaded from the UE to the cloud over the wireless uplink of the edge cell; its epoch-level transmission estimate is $T^{\text{cloud}}_{\text{up}}$. (ii) Batch admission and kernel configuration: The cloud server aggregates the incoming prompts into prefill mini-batches so that the large-tile kernel can saturate the tensor cores, incurring a queueing delay $T^{\text{cloud}}_{\text{batch}}$. Symmetrically, the edge server continues to queue the KV-enabled decode requests and applies the same sliding-window merge policy to build decode batches that exploit small-tile edge parallelism; the corresponding queueing delay is $T^{\text{edge}}_{\text{batch}}$. (iii) Execution and delivery: The cloud then executes the prefill kernel, and attaches sampled token 1 to the KV-migration metadata. The full prompt KV is sent to the edge through the backhaul. After loading the KV and admitting the request to an edge Decode batch, the edge forwards token 1 and launches Decode for the remaining tokens. This conservative release rule prevents edge queueing from appearing as an unmodeled first inter-token gap. The corresponding predicted execution-path time is $T^{\text{split}}_{\text{exe}} = T^{\text{cloud}}_{\text{prefill}} + \widehat{T}^{\text{e2e}}_{\text{kv}} + T^{\text{edge}}_{\text{decode,rem}}$. Finally, the generated token stream is delivered to the UE over the downlink. Collecting all terms yields a conservative non-overlap upper bound for the end-to-end communication-and-compute latency, along with the TTFT and TPOT terms:

$$\begin{aligned}\widehat{T}^{\text{split}}_{u,i} = {} & T^{\text{cloud}}_{\text{up}} + T^{\text{cloud}}_{\text{batch}} + T^{\text{cloud}}_{\text{prefill}} + \widehat{T}^{\text{e2e}}_{\text{kv}} \\ & + T^{\text{edge}}_{\text{batch}} + T^{\text{edge}}_{\text{decode,rem}} + T^{\text{edge}}_{\text{down}},\end{aligned} \tag{18}$$

$$\begin{aligned}\widehat{T}^{\text{FT,split}}_{u,i} = {} & T^{\text{cloud}}_{\text{up}} + T^{\text{cloud}}_{\text{batch}} + T^{\text{cloud}}_{\text{prefill}} \\ & + \widehat{T}^{\text{e2e}}_{\text{kv}} + T^{\text{edge}}_{\text{batch}} + T^{\text{edge}}_{\text{down,1}},\end{aligned} \tag{19}$$

$$\widehat{T}^{\text{PT,split}}_{u,i} = \begin{cases} \dfrac{T^{\text{edge}}_{\text{decode,rem}} + T^{\text{edge}}_{\text{down,rem}}}{L^{\text{out}}_{u,i} - 1}, & L^{\text{out}}_{u,i} > 1, \\ 0, & L^{\text{out}}_{u,i} = 1. \end{cases} \tag{20}$$

The split TTFT expression is a conservative upper bound: if the edge decode queue can reserve capacity while cloud prefill and KV transfer are still in progress, $T^{\text{edge}}_{\text{batch}}$ can partially overlap with the preceding terms.

Cloud-only mode. In cloud-only mode, both prefill and decode run in the cloud. This mode is useful when the edge GPU lacks memory or decode capacity, or when KV backhaul serialization makes split execution unattractive. Its latency components are

$$\begin{aligned}\widehat{T}^{\text{cloud}}_{u,i} = {} & T^{\text{cloud}}_{\text{up}} + T^{\text{cloud}}_{\text{batch}} + T^{\text{cloud}}_{\text{prefill}} \\ & + T^{\text{cloud}}_{\text{decode,rem}} + T^{\text{cloud}}_{\text{down}},\end{aligned} \tag{21}$$

$$\begin{aligned}\widehat{T}^{\text{FT,cloud}}_{u,i} = {} & T^{\text{cloud}}_{\text{up}} + T^{\text{cloud}}_{\text{batch}} \\ & + T^{\text{cloud}}_{\text{prefill}} + T^{\text{cloud}}_{\text{down,1}},\end{aligned} \tag{22}$$

$$\widehat{T}^{\text{PT,cloud}}_{u,i} = \begin{cases} \dfrac{T^{\text{cloud}}_{\text{decode,rem}} + T^{\text{cloud}}_{\text{down,rem}}}{L^{\text{out}}_{u,i} - 1}, & L^{\text{out}}_{u,i} > 1, \\ 0, & L^{\text{out}}_{u,i} = 1. \end{cases} \tag{23}$$

C. QoS Metrics

For each request, let $x^m_{u,i} \in \{0,1\}$ indicate whether mode $m \in \mathcal{M} = \{\text{MIX}, \text{SPLIT}, \text{CLOUD}\}$ is selected. Exactly one mode is selected:

$$\sum_{m\in\mathcal{M}} x^m_{u,i} = 1, \qquad x^m_{u,i} \in \{0,1\}. \tag{24}$$

Each request is associated with one serving edge/AP domain for wireless access, and bandwidth fractions remain within the OFDMA simplex:

$$\sum_{e\in\mathcal{E}} y^e_u(t) = 1, \quad y^e_u(t) \in \{0,1\}, \quad 0 \le \alpha^{\text{UL}}_{u,e}, \alpha^{\text{DL}}_{u,e} \le y^e_u(t). \tag{25}$$

Unlike the hatted admission predictors, the unhatted mode latencies below are observed at run time. The realized end-to-end latency, TTFT, and TPOT are

$$\begin{aligned} T_{u,i} &= \sum_{e\in\mathcal{E}}\sum_{m\in\mathcal{M}} y^e_u(t) x^m_{u,i} T^m_{u,i}(e), \\ T^{\text{FT}}_{u,i} &= \sum_{e\in\mathcal{E}}\sum_{m\in\mathcal{M}} y^e_u(t) x^m_{u,i} T^{\text{FT},m}_{u,i}(e), \\ T^{\text{PT}}_{u,i} &= \sum_{e\in\mathcal{E}}\sum_{m\in\mathcal{M}} y^e_u(t) x^m_{u,i} T^{\text{PT},m}_{u,i}(e). \end{aligned} \tag{26}$$

The request is SLO-conformant if it satisfies all three constraints:

$$T_{u,i} \le \tau_{u,i}, \qquad T^{\text{FT}}_{u,i} \le D^{\text{FT}}_{u,i}, \qquad T^{\text{PT}}_{u,i} \le D^{\text{PT}}_{u,i}. \tag{27}$$

IV. Problem Formulation and Solutions

A. Design Overview

The solution follows the objective in P1 but avoids solving the full mixed-integer, time-varying problem online. Instead, Mix-or-Split uses a staged runtime procedure. First, the mode selector evaluates cloud-only, split, and mix execution for each incoming request using the measured wireless rates, KV-transfer cost, deadline slack, and guarded SM feasibility. Second, the admission controller closes a short waiting window, forms δ-similar prompt buckets, and adjusts the request/token budgets according to recent TTFT, TPOT, and deadline violations. Third, the GPU planner maps the admitted batches to profiled prefill/decode tiles and selects a predicted allocation plan that satisfies the guarded SM-resource constraints. Finally, during execution, the rescue loop updates only the next kernel-wave worklist when request-level TPOT/deadline slack or batch-level TTFT slack becomes unsafe. This staged design keeps the online decision lightweight while preserving the coupling among routing, batching, and SM-level execution.

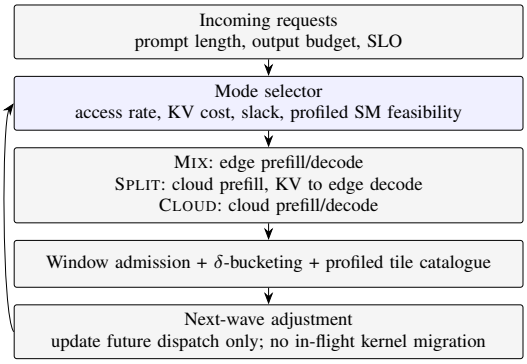

Fig. 1. Design overview. Mix-or-Split selects one of three modes, then applies window admission, prompt bucketing, profiled tile planning, and next-wave adjustment using observed SLO slack.

B. Problem Formulation

For each request $r_{u,i}$ the scheduler selects a mode indicator $x^m_{u,i} \in \{0,1\}$ for each $m \in \mathcal{M}$, a hosting edge server/access domain $y^e_u(t) \in \{0,1\}$, the uplink/downlink bandwidth fractions $\alpha^{\mathrm{UL}}_{u,e}$ and $\alpha^{\mathrm{DL}}_{u,e}$, and the planned Prefill/Decode chunk counts $(n_{p,e,s}, n_{d,e,s})$ for the target $\mathrm{SM}(e,s)$. In addition, it controls window-level admission and GPU tiling: the token/request caps $(S^{\mathrm{cap}}_k, B^{\mathrm{cap}}_k)$, the δ-similar buckets, the prefill/decode tile configurations $(\mathcal{T}_p, \mathcal{T}_d)$, and the boundary update $\Delta\mathbf{n}$ to the next-wave allocation plan. The scheduling policy

$$\pi = \{x^m_{u,i}, y^e_u(t), (\alpha^{\mathrm{UL}}_{u,e}, \alpha^{\mathrm{DL}}_{u,e}), (n_{p,e,s}, n_{d,e,s}), (S^{\mathrm{cap}}_k, B^{\mathrm{cap}}_k), \delta, \Delta\mathbf{n}, (\mathcal{T}_p, \mathcal{T}_d)\}$$

must obey the routing, edge-capacity, SLO requirements, deadline constraints and the micro-level resource split. Consider a measurement interval of duration T.

Let $\mathcal{R}_T$ be the set of requests that the scheduler admits during this interval and let $\mathcal{C}_T \subseteq \mathcal{R}_T$ denote the subset that complete no later than their individual deadline and meet the prescribed TTFT and TPOT thresholds. We quantify system performance by [12]:

$$\begin{aligned} G(\pi,T) &= \frac{|\mathcal{C}_T|}{T}, \\ A(\pi,T) &= \frac{|\mathcal{C}_T|}{\max\{|\mathcal{R}_T|, 1\}}, \end{aligned} \tag{28}$$

where G is the goodput and $A \in [0,1]$ is the attainment ratio. Goodput gauges the rate at which quality-of-service (QoS) conformant requests finish, whereas SLO attainment measures the fraction of admitted requests that satisfy all three latency requirements. The maximum in the denominator sets $A(\pi,T) = 0$ when $|\mathcal{R}_T| = 0$.

Given an SLO-attainment target $\bar{A} \in (0,1]$, the scheduler uses a GPU-wide guarded Mix set $\mathcal{A}^{\mathrm{mix}}_g(e,t)$. A plan belongs to this set when every per-SM pair $(n_{p,e,s}, n_{d,e,s})$ lies in $\mathcal{X}_g(t)$ and at least one SM pair lies in the positive joint set $\mathcal{X}^{\mathrm{mix}}_g(t)$ defined below. Let $\mathbf{n}_e(t)$ denote that GPU-wide plan. The long-run scheduling objective is

$$\begin{aligned} \mathbf{P1}: \quad \max_{\pi} \ & \liminf_{T\to\infty} \mathbb{E}\left[G(\pi,T)\right] \\ \text{s.t.} \ & \liminf_{T\to\infty} \mathbb{E}\left[A(\pi,T)\right] \geq \bar{A}, \\ & (1),\ (24),\ (25), \\ & x^m_{u,i} y^e_u(t) = 1 \Rightarrow m \in \mathcal{M}^{\mathrm{feas}}_{u,i}(e,t), \\ & x^{\mathrm{mix}}_{u,i} y^e_u(t) = 1 \Rightarrow \mathbf{n}_e(t) \in \mathcal{A}^{\mathrm{mix}}_g(e,t), \\ & \forall (u,i) \in \mathcal{R}_T,\ m \in \mathcal{M},\ e \in \mathcal{E}. \end{aligned} \tag{29}$$

Here $x^{\mathrm{mix}}_{u,i}$ denotes the indicator for $m = \mathrm{MIX}$. The feasible-set implication applies the predicted SLO and availability checks in Eq. (13) at admission, whereas Eqs. (28) and (27) use realized latencies. This distinction keeps the long-run attainment constraint nontrivial under prediction error and run-time interference. For an edge-mix request, every per-SM plan must remain in the guarded nonnegative region and at least one SM must have a positive joint Prefill/Decode allocation; this does not require every SM to host both phases. Cloud-only requests and split-mode edge decode are handled by their corresponding capacity constraints. Due to the high dimensionality and non-convexity of P1, solving it exactly within every online decision epoch is impractical. Instead, we use P1 as a guiding objective, and design a hierarchical heuristic scheduler that approximates its behavior through window-level admission, δ-similar bucketing, and GPU tile-aware allocation.

C. SM-Level Feasibility for Edge Mix Execution

Why not only chunked prefill? Chunked prefill reduces head-of-line blocking by slicing long prompts and interleaving prefill chunks with decode requests. It is effective after the system has already decided to serve a request within a given GPU pool. However, chunked prefill does not answer whether the request should be served by cloud-only, split, or edge-mix execution, nor does it model cloud-edge KV serialization or wireless access delay. Moreover, edge mix execution requires a stricter admission test: the profiled prefill/decode chunk allocation must fit under the same guarded SM-resource budgets. Mix-or-Split therefore treats chunked prefill as complementary. It can be used within a selected mode, while our scheduler decides whether mix is admissible and which prefill/decode allocation is SM-feasible for the next wave.

All three modes traverse the same UE–AP wireless access in the stated architecture, so radio feasibility is a shared admission condition rather than a Mix-specific advantage. After that condition is satisfied, the scheduler admits prefill+decode mixed mode only when the edge GPU has adequate registers, threads, warps, shared memory, and CTA slots for the profiled prefill/decode allocation and the current batch can be processed without hitting a hardware wall. If mix-mode SM feasibility fails, the scheduler evaluates Split and Cloud as competing alternatives through Eq. (14); if neither is feasible, it delays or rejects the request.

Let $\mathcal{U}_e(t)$ be the set of UEs served by edge server e during scheduling epoch t. The shared wireless condition is that the minimum UL/DL bandwidth fractions required by all active users fit within the OFDMA budget:

$$\sum_{u\in\mathcal{U}_e(t)} \alpha^{\mathrm{UL}}_{u,e,\min} \le 1, \qquad \sum_{u\in\mathcal{U}_e(t)} \alpha^{\mathrm{DL}}_{u,e,\min} \le 1. \tag{30}$$

Mix admission additionally requires enough free register, thread, shared-memory, warp, and CTA-slot budget for at least one profiled Prefill planning chunk and one profiled Decode planning chunk. A planning chunk is a fixed scheduler work unit with a compiler/profile-derived aggregate resident-resource vector. It can reserve C_ϕ physical CTA slots for phase $\phi \in \{p, d\}$; $C_\phi = 1$ when the unit is one CUDA thread block. This definition avoids equating an arbitrary request slice with one physical CTA. The implementation checks all five resource walls, while the reduced analysis below retains only walls shown by profiling to be active for the selected GPU and tile pair.

Let (e, s) denote the s-th streaming multiprocessor (SM) of edge GPU e. We aggregate all five resource dimensions, including CTA slots, registers, warps, shared memory, and threads into a single vector. Let C_s and T_s denote the per-SM CTA-slot and thread limits, respectively:

$$\Delta_{e,s} = \left(\underbrace{\Delta^{\mathrm{CTA}}_{e,s}}_{\text{CTA slots}}, \underbrace{\Delta^{\mathrm{REG}}_{e,s}}_{\text{registers}}, \underbrace{\Delta^{\mathrm{WARP}}_{e,s}}_{\text{warps}}, \underbrace{\Delta^{\mathrm{SMEM}}_{e,s}}_{\text{shared memory}}, \underbrace{\Delta^{\mathrm{THR}}_{e,s}}_{\text{threads}} \right) \tag{31}$$

with the components defined once for all as

$$\begin{cases} \Delta^{\mathrm{CTA}}_{e,s} = C_s - (n_{p,e,s}C_p + n_{d,e,s}C_d)\,, \\ \Delta^{\mathrm{REG}}_{e,s} = R_s - (n_{p,e,s}R_p + n_{d,e,s}R_d)\,, \\ \Delta^{\mathrm{WARP}}_{e,s} = W_s - (n_{p,e,s}W_p + n_{d,e,s}W_d)\,, \\ \Delta^{\mathrm{SMEM}}_{e,s} = S_s - (n_{p,e,s}S_p + n_{d,e,s}S_d)\,, \\ \Delta^{\mathrm{THR}}_{e,s} = T_s - (n_{p,e,s}T_p + n_{d,e,s}T_d)\,. \end{cases} \tag{32}$$

A vector component $\Delta^{(\cdot)}_{e,s} < 0$ indicates that the corresponding wall is violated, i.e. the predicted allocation oversubscribes that resource dimension. Consequently, the admission controller blocks additional next-wave work until the resource recovers.

For either phase, the pure-phase resident planning-chunk count is the minimum of the CTA-slot, register, SMEM, warp, and thread ceilings. For example, Decode has $\min\{\lfloor C_s/C_d\rfloor, \lfloor R_s/R_d\rfloor, \lfloor S_s/S_d\rfloor, \lfloor W_s/W_d\rfloor, \lfloor T_s/T_d\rfloor\}$. NVIDIA hardware schedules threads in warps of 32, but CTA-slot, warp, and thread limits are read from the actual device profile rather than inferred from a different GPU model.

Reduced state vector. If offline profiling on the selected GPU shows that register, shared-memory, and warp constraints become active before CTA-slot and thread-count limits, the closed-form analysis can retain the following three dominant constraints while the implementation continues to check all five:

$$\Delta^{\mathrm{red}}_{e,s} = \left(\Delta^{\mathrm{REG}}_{e,s}, \Delta^{\mathrm{WARP}}_{e,s}, \Delta^{\mathrm{SMEM}}_{e,s}\right). \tag{33}$$

Note that shared memory is allocated per CTA and shared across its warps. Thus, while both warp count and shared memory usage depend on the tile configuration, at the resource-accounting level they are treated as orthogonal dimensions: the warp budget limits scheduler concurrency, whereas the SMEM budget constrains tile size and pipeline depth.

We adopt a three-level compute tiling hierarchy: (1) TB-tile (thread-block tile): the output panel produced by one CTA (thread block), of size $(M_{\mathrm{tb}}, N_{\mathrm{tb}})$. (2) Warp-tile: the sub-panel inside a TB-tile computed by a single warp, of size $(M_{\mathrm{warp}}, N_{\mathrm{warp}})$. (3) TC-tile / MMA micro-tile: the minimal output block produced by one Tensor Core instruction. For the FP16 Ampere catalogue, we parameterize the micro-tile as $\mathrm{mma.m16n8k16}$, so $(M_{\mathrm{mma}}, N_{\mathrm{mma}}, K_{\mathrm{mma}}) = (16, 8, 16)$.

A kernel configuration further specifies the K-advance per main-loop step K_{step} (aligned with K_{mma}), the pipeline depth stages (2 for double buffering, 3 for triple buffering), register-level staging factors $U_A, U_B \in \{1, 2\}$, element sizes (b_A, b_B) in bytes, and a per-thread temporary budget r_{misc}.

Given kernel configuration, the CTA-level parallel partitioning is fixed as follows. The number of warps per CTA is $W_{\mathrm{CTA}} = (M_{\mathrm{tb}}/M_{\mathrm{warp}}) \times (N_{\mathrm{tb}}/N_{\mathrm{warp}})$ with both ratios integral, and the threads per CTA are $T_{\mathrm{block}} = 32W_{\mathrm{CTA}}$. A warp holds $P = (M_{\mathrm{warp}}/M_{\mathrm{mma}}) \times (N_{\mathrm{warp}}/N_{\mathrm{mma}})$ MMA micro-tiles concurrently (on Ampere FP16, $P = (M_{\mathrm{warp}}/16) \times (N_{\mathrm{warp}}/8)$), which directly sets the accumulator footprint. Intuitively, W_{CTA}governs the resource upper bound on CTA-level parallelism, whereas P controls per-thread accumulator count and therefore register pressure.

Shared memory is not allocated per warp, but per CTA; increasing the number of warps (at fixed tile size) does not directly increase SMEM usage. For the dense kernels used in our evaluation, the shared-memory footprint per CTA is approximated by the staged A and B panels plus auxiliary buffers [14]:

$$S = \eta_{\mathrm{stage}}\left(b_A M_{\mathrm{tb}} K_{\mathrm{step}} + b_B N_{\mathrm{tb}} K_{\mathrm{step}}\right) + S_{\mathrm{aux}}, \tag{34}$$

where η_{stage} is the pipeline depth and S_{aux} covers alignment, metadata, and epilogue buffers. These buffers must satisfy $S \le S_s$, and K_{step} and η_{stage} are selected to balance reuse against SMEM pressure. In the implementation, the final S value is taken from compiler/profiler measurements; Eq. (34) only explains the dominant scaling terms.

The register footprint is estimated per thread and scaled to the CTA. For Ampere FP16 with $\mathrm{mma.m16n8k16}$ and FP32 accumulation, a practical approximation is

$$r_{\mathrm{thr}} = 4U_A + 2U_B + 4P + r_{\mathrm{misc}}, \quad R = 32W_{\mathrm{CTA}}r_{\mathrm{thr}}, \tag{35}$$

where $4U_A$ and $2U_B$ account for A/B fragments staged in registers and $4P$ counts FP32 accumulators per concurrently held 16×8 micro-tile (the dominant term in practice). As with SMEM, the implementation uses compiler-reported register counts; Eq. (35) is used to explain how tile geometry drives register pressure. When scaled by the planned chunk counts per SM, $(n_{p,e,s}, n_{d,e,s})$, the measured register usage must respect the per-SM register budget R_s.

We connect kernel micro-design (tile geometry) with runtime macro-scheduling (planning-chunk allocation per SM). For each phase $\phi \in \{p, d\}$, candidate tile $\mathcal{T}_\phi$, and profile

context ξ_ϕ (GPU, model/kernel, sequence band, and batch band), offline compilation and profiling yield

$$(C_\phi, R_\phi, W_\phi, S_\phi, T_\phi, L_\phi) = f_\phi\left(\mathcal{T}_\phi; \xi_\phi\right), \phi \in \{p, d\}, \quad (36)$$

where C_ϕ, $\{R, W, S, T\}_\phi$, and L_ϕ denote the reserved CTA slots, aggregate resident resources, and measured wall time of one Prefill/Decode planning chunk, respectively.

A profiled prefill planning chunk covers $w_p(\mathcal{T}_p; \xi_p)$ effective prompt-token rows. For a bucket B with padded prompt length $L_B^{\text{in}} = \max_{(u,i)\in B} L_{u,i}^{\text{in}}$ the dominant work-unit count is

$$\text{Tiles}_B^{\text{prefill}} = \left\lceil \frac{|B| L_B^{\text{in}}}{w_p(\mathcal{T}_p; \xi_p)} \right\rceil. \quad (37)$$

The scheduler uses a profiled prefill planning-chunk function

$$N_{B,p}^{\text{chunk}} = g_p^{\text{chunk}}\left(|B|, L_B^{\text{in}}; \mathcal{T}_p, \xi_p\right). \quad (38)$$

The explanatory tile count in Eq. (37) captures the implemented flattened-batch scaling; hidden-size, layer, head, split-K, and multi-kernel factors are absorbed into g_p^{chunk} and verified by offline profiling. Define $c_p(t) = \sum_{s=1}^{N_{\text{sm}}} n_{p,e,s}(t)$. Prefill requires

$$b_p = \begin{cases} \left\lceil N_{B,p}^{\text{chunk}} / c_p(t) \right\rceil, & c_p(t) > 0, \\ +\infty, & c_p(t) = 0, \end{cases} \quad (39)$$

waves. Similarly, if $N_{B,d}^{\text{chunk}}$ is the Decode planning-chunk count for one output-token step and $c_d(t) = \sum_{s=1}^{N_{\text{sm}}} n_{d,e,s}(t)$, then

$$b_d = \begin{cases} \left\lceil N_{B,d}^{\text{chunk}} / c_d(t) \right\rceil, & c_d(t) > 0, \\ +\infty, & c_d(t) = 0. \end{cases} \quad (40)$$

For homogeneous SM allocations, these denominators reduce to $N_{\text{sm}} n_{p,e,s}$ and $N_{\text{sm}} n_{d,e,s}$, respectively. For a mix-mode edge batch, the estimated batch-level first-token latency and per-token decode latency are

$$\widehat{T}_B^{\text{FT}} = T_{\text{up},B} + T_{\text{batch},B} + b_p L_p + T_{\text{down},1,B}, \quad (41)$$

$$\widehat{T}_B^{\text{PT}} = b_d L_d(\mathcal{T}_p, \mathcal{T}_d, \mathbf{n}; \xi_d) + T_{\text{down,step},B}, \quad (42)$$

where $T_{\text{down},1,B}$ and $T_{\text{down,step},B}$ are the batch-level worst-case first-token and non-first-token delivery estimates, and $L_d(\mathcal{T}_p, \mathcal{T}_d, \mathbf{n}; \xi_d)$ is the measured per-wave Decode latency for the selected joint tile/allocation profile. Thus any observed Prefill–Decode contention is included in the profile rather than introduced through an uncalibrated analytic factor. For split/cloud-only batches, the corresponding cloud queueing, KV, and cloud decode terms from the mode-level formulas are added. The mix-mode batch makespan upper bound is

$$\Theta_B = \widehat{T}_B^{\text{FT}} + \max\left\{ \max_{(u,i)\in B} L_{u,i}^{\text{out}} - 1, 0 \right\} \widehat{T}_B^{\text{PT}}. \quad (43)$$

During each scheduling cycle, the runtime executes four ordered control actions: i) *Prompt bucketing, token and request admission.* The scheduler closes the current arrival window, groups prompts into δ-similar buckets, and admits a bounded number of requests and tokens. This bounds the window-induced waiting time and narrows the profiled Prefill-runtime range used in the batch-wide time-to-first-token (TTFT) estimate. (ii) *Tile selection.* With the request batch finalized, the system evaluates available prefill tile configurations (e.g., 128×64, 64×64) against the active profiled resource walls; all five resource dimensions are checked at admission. The tile with the lowest predicted TTFT among valid candidates is selected. (iii) *SM-feasible chunk allocation.* After tile selection, the runtime plans profiled prefill/decode chunks for the next dispatch wave. This allocation must satisfy the guarded hardware constraints and maximize the profiled batch utility. (iv) *Dynamic execution adjustment.* During the decode phase, the system evaluates each request's timing slack. If impending deadline violations are detected, it shifts next-wave decode allocation from requests with ample slack to those in danger of missing SLOs. Conversely, TTFT delays trigger next-wave allocation shifts from decode back to prefill operations.

1) *Token and request admission control.* Let a_i represent the arrival time of request i. A sliding waiting window $W(t_0, \Delta t) = \{i \mid t_0 \le a_i < t_0 + \Delta t\}$ is opened at time t_0 and closed when either (i) the window length reaches Δt ms, (ii) the number of collected requests reaches B_k^{cap}, or (iii) admitting the next request would make its accumulated input-token count exceed S_k^{cap}. Hence $|B_k| \le B_k^{\text{cap}}$, $S_k := \sum_{(u,i)\in B_k} L_{u,i}^{\text{in}} \le S_k^{\text{cap}}$, and the waiting time of every request in the window is at most Δt. For a fixed-length window and a stationary arrival process of intensity λ, the uncapped expected arrival count is $\lambda \Delta t$; the request and token caps can only reduce the retained batch size. Let V_k^{FT} denote the number of requests whose time-to-first-token exceeds the TTFT SLO, V_k^{PT} denote the number of requests whose per-token decode latency exceeds the TPOT SLO, and V_k^{DL} denote the number of requests that miss their absolute deadline. The corresponding violation ratios are defined as

$$\begin{aligned} \phi_k^{\text{FT}} &= \frac{V_k^{\text{FT}}}{\max\{|B_k|, 1\}}, \\ \phi_k^{\text{PT}} &= \frac{V_k^{\text{PT}}}{\max\{|B_k|, 1\}}, \\ \phi_k^{\text{DL}} &= \frac{V_k^{\text{DL}}}{\max\{|B_k|, 1\}}. \end{aligned} \quad (44)$$

The token and request caps S_k^{cap} and B_k^{cap} for the next window W_{k+1}, are dynamically adjusted based on observed violation ratios $\left(\phi_k^{\text{FT}}, \phi_k^{\text{PT}}, \phi_k^{\text{DL}}\right)$ using a violation-aware update rule: when deadline violations exceed tolerance, both budgets are reduced; when both TTFT and TPOT violations exceed tolerance ε (C_1), both budgets are scaled down by factors γ_s, and γ_b respectively; for TTFT-only violations (C_2), only the token budget is reduced; for TPOT-only violations (C_3), only the request budget is decreased; otherwise, both budgets receive additive increments Δ_s, Δ_b, providing a bounded adaptive response (after clipping) to quality-of-service violations while promoting throughput. Define the priority-ordered branch conditions for batch k: $C_0 : \phi_k^{\text{DL}} \ge \varepsilon_{DL}$; $C_1 : \phi_k^{\text{FT}} \ge \varepsilon \wedge \phi_k^{\text{PT}} \ge \varepsilon$; $C_2 : \phi_k^{\text{FT}} \ge \varepsilon \wedge \phi_k^{\text{PT}} < \varepsilon$; $C_3 : \phi_k^{\text{FT}} < \varepsilon \wedge \phi_k^{\text{PT}} \ge \varepsilon$, where $\varepsilon, \varepsilon_{DL} \in (0, 1]$. The controller checks C_0 first; only if C_0 is false does it select

one of C_1, C_2, C_3, or the otherwise branch. Then the cap update keeps its branch structure:

$$\left(S_{k+1}^{\mathrm{cap}}, B_{k+1}^{\mathrm{cap}}\right) = \begin{cases} \left(\lfloor \gamma_s S_k^{\mathrm{cap}} \rfloor, \lfloor \gamma_b B_k^{\mathrm{cap}} \rfloor\right), & C_0 \vee C_1, \\ \left(\lfloor \gamma_s S_k^{\mathrm{cap}} \rfloor, B_k^{\mathrm{cap}}\right), & C_2, \\ \left(S_k^{\mathrm{cap}}, \lfloor \gamma_b B_k^{\mathrm{cap}} \rfloor\right), & C_3, \\ \left(S_k^{\mathrm{cap}} + \Delta_s, B_k^{\mathrm{cap}} + \Delta_b\right), & \text{otherwise.} \end{cases} \tag{45}$$

with $0 < \gamma_s, \gamma_b < 1$ and integer increments $\Delta_s, \Delta_b > 0$. This bounded AIMD-like rule is a run-time heuristic rather than a claim of closed-loop asymptotic stability. The raw update is clipped to hardware-safe ranges:

$$\begin{aligned} S_{k+1}^{\mathrm{cap}} &\leftarrow \mathrm{clip}\left(S_{k+1}^{\mathrm{cap}}, S_{\min}, S_{\max}\right), \\ B_{k+1}^{\mathrm{cap}} &\leftarrow \mathrm{clip}\left(B_{k+1}^{\mathrm{cap}}, B_{\min}, B_{\max}\right). \end{aligned} \tag{46}$$

Prompt Bucketing: This subsection formalizes the prompt-bucketing stage that precedes GPU-level tiling. The construction bounds padding overhead and the number of buckets per window. Under the linear Prefill profile assumed in Proposition 2, it also bounds within-bucket latency dispersion.

δ-similar buckets. Let L_i denote the input-token length of request i, with $L_i > 0$. For a threshold $\delta \in (0, 1)$, a subset $B \subset W$ is called a δ-similar bucket if

$$\begin{aligned} &\frac{L_{\max}(B)}{L_{\min}(B)} \leq 1 + \delta, \\ &L_{\max}(B) = \max_{i \in B} L_i, \quad L_{\min}(B) = \min_{i \in B} L_i. \end{aligned} \tag{47}$$

Remark. Eq. (47) means that the shortest sequence in the bucket is padded to at most $(1 + \delta)$ times its original length. It directly bounds padding waste and, under the linear model in Proposition 2, bounds Prefill-latency dispersion.

Proposition 1. ***(Padding upper bound)*** *For any δ-similar bucket B, the padding waste inside any bucket is bounded by*

$$\rho\left(B\right) = 1 - \frac{\sum_{i \in B} L_i}{|B|\, L_{\max}(B)} \leq \frac{\delta}{1 + \delta}. \tag{48}$$

Proof. In a δ-similar bucket, every sequence is padded to $L_{\max}(B)$. Since $L_i \geq L_{\min}(B)$ for all $i \in B$ and $L_{\max}(B) \leq (1 + \delta) L_{\min}(B)$,

$$\rho(B) \leq 1 - \frac{|B| L_{\min}(B)}{|B| L_{\max}(B)} \leq 1 - \frac{1}{1 + \delta} = \frac{\delta}{1 + \delta}.$$

Proposition 2. ***(Prefill-latency standard-deviation bound)*** *For any δ-similar bucket B, assume that the profiled Prefill planning-chunk latency is well approximated over the bucket by $L_p\left(i\right) = c_p L_i + d_p$, where $c_p > 0$. Then*

$$\begin{aligned} \sigma_{L_p} &= \sqrt{\mathrm{Var}_{i \in B} L_p\left(i\right)} \\ &\leq c_p \left(L_{\max}(B) - L_{\min}(B)\right) \\ &\leq c_p \frac{\delta}{1 + \delta} L_{\max}(B). \end{aligned} \tag{49}$$

Proof. The standard deviation, not the variance, scales linearly under an affine transformation, hence $\sigma_{L_p} = c_p \sigma_L$. For any finite sample, the standard deviation is upper-bounded by the sample range, so $\sigma_L \leq L_{\max}(B) - L_{\min}(B)$. Using Eq. (47), $L_{\max}(B) - L_{\min}(B) \leq \frac{\delta}{1+\delta} L_{\max}(B)$, which proves Eq. (49).

Proposition 3. ***(Upper bound on the number of buckets per window)*** *Under the logarithmic bucketing rule used by the scheduler, let the token lengths that fall into a window $W\left(t_{r_0}, \Delta t\right)$ lie in the closed interval $[L_{\min}, L_{\max}]$ with $L_{\max}/L_{\min} > 1$. When bucketing is executed with similarity tolerance δ, the number K of non-empty δ-similar buckets it emits satisfies*

$$K \leq \left\lceil \log_{1+\delta} \frac{L_{\max}}{L_{\min}} \right\rceil + 1. \tag{50}$$

Proof. The logarithmic bucketing rule partitions the length axis so that every returned bucket is δ-similar; hence the shortest and longest sequence lengths that reside in the same bucket differ by at most a factor $1 + \delta$. Arrange the whole interval $[L_{\min}, L_{\max}]$ onto the logarithmic axis with base $1 + \delta$; more precisely, partition the interval by the grid $L_{\min}, (1 + \delta) L_{\min}, (1 + \delta)^2 L_{\min}, \ldots, (1 + \delta)^q L_{\min}$, where $q := \left\lceil \log_{1+\delta} \frac{L_{\max}}{L_{\min}} \right\rceil$. Consequently, the algorithm never needs more than $q + 1$ buckets.

2) *Tile-design methodology:* The CUDA tile catalogue is constructed in two steps: (i) micro-design under the profiled resource walls; and (ii) run-time mapping driven by prompt length, augmented with a guarded tile-downshift rollback path on the prefill side.

Tile micro-design under profiled resource walls. Prefill phase (throughput-bound). We enumerate tensor-core tiles and retain candidates that satisfy the register, shared-memory, warp, CTA-slot, and thread limits reported by the target GPU. Decode phase (latency-bound) uses the same test for lighter candidates. The catalogue stores the complete vector in Eq. (36); no resource limit from a different GPU model is substituted for the edge profile.

Run-time mapping driven by prompt length. For a given batch size, the implementation maps a request to the smallest profiled sequence length that is no shorter than the request. Thus, prompts shorter than 128 tokens use the 128-token profile, and the gaps $(1024, 1536]$ and $(2048, 3072]$ use the 1536- and 3072-token profiles, respectively. If no same-batch ceiling profile exists, the request is marked *unprofiled* for Mix and the scheduler evaluates the remaining modes; it is not called SM-infeasible. Only a request with a matched profile that subsequently fails one of the guarded resource inequalities is marked *SM-infeasible*. Within these profiled length bands, short-to-moderate prompts use a small prefill tile to reduce the guarded resource footprint; medium prompts use a larger tile that preserves high tensor-core efficiency while remaining within the guarded register budget; long prompts use the largest profiled tile when the guarded constraints permit. In our catalogue, Decode uses the fixed 64×32 tile. At admission the scheduler (i) classifies the prompt into a profiled length/batch region; (ii) looks up the corresponding prefill-tile resource vector; and (iii) checks the five resource inequalities. If the guarded test passes, the request is admitted to the SM-feasible next-wave allocation; otherwise the scheduler evaluates the remaining feasible modes through Eq. (14).

3) *SM-feasible chunk allocation:* A local scheduler decides the planned Prefill and Decode chunk counts, denoted by $n_{p,e,s}$

and $n_{d,e,s}$, for a given SM at the next dispatch boundary. The resource constraints are:

$$\begin{aligned}\mathcal{X}_0(t) = \{(n_p, n_d) \in \mathbb{Z}^2_{\geq 0} : & \, n_p C_p + n_d C_d \leq C_s, \\ & n_p R_p + n_d R_d \leq R_s, \\ & n_p W_p + n_d W_d \leq W_s, \\ & n_p S_p + n_d S_d \leq S_s, \\ & n_p T_p + n_d T_d \leq T_s\}.\end{aligned} \tag{51}$$

To avoid near-wall oscillations, all planned updates (including rollbacks) must satisfy a guarded feasibility region with nonnegative safety margins $\mathbf{g} = (g_C, g_R, g_W, g_S, g_T)$, each smaller than its corresponding device budget:

$$\begin{aligned}\mathcal{X}_g(t) = \{(n_p, n_d) \in \mathbb{Z}^2_{\geq 0} : & \, n_p C_p + n_d C_d \leq C_s - g_C, \\ & n_p R_p + n_d R_d \leq R_s - g_R, \\ & n_p W_p + n_d W_d \leq W_s - g_W, \\ & n_p S_p + n_d S_d \leq S_s - g_S, \\ & n_p T_p + n_d T_d \leq T_s - g_T\}.\end{aligned} \tag{52}$$

Initial concurrent-Mix admission uses $\mathcal{X}_g^{\mathrm{mix}}(t) = \mathcal{X}_g(t) \cap \mathbb{Z}^2_{\geq 1}$; $\mathcal{X}_g(t)$ retains zero-component plans only for temporary single-phase next-wave dispatch. The GPU-wide set $\mathcal{A}_g^{\mathrm{mix}}(e,t)$ used in P1 contains plans $\mathbf{n}_e = ((n_{p,e,s}, n_{d,e,s}))_{s=1}^{N_{\mathrm{sm}}}$ whose per-SM pairs all lie in $\mathcal{X}_g(t)$ and for which at least one SM pair lies in $\mathcal{X}_g^{\mathrm{mix}}(t)$.

Given the profiled per-chunk vectors in Eq. (36), the runtime selects the initial GPU-wide concurrent-Mix plan by maximizing a batch utility over this guarded feasible set:

$$\mathbf{n}_e^* = \underset{\mathbf{n}_e \in \mathcal{A}_g^{\mathrm{mix}}(e,t)}{\arg\max} \; \mathcal{U}_B\left(\mathbf{n}_e\right), \tag{53}$$

where $\mathcal{U}_B$ is a profiled batch-level utility:

$$\begin{aligned}\mathcal{U}_B(\mathbf{n}_e) = & -\omega_F \left[\frac{\widehat{T}_B^{\mathrm{FT}}(\mathbf{n}_e) - d_B^{\mathrm{FT}}}{d_B^{\mathrm{FT}}}\right]_+ \\ & -\omega_P \left[\frac{\widehat{T}_B^{\mathrm{PT}}(\mathbf{n}_e) - d_B^{\mathrm{PT}}}{d_B^{\mathrm{PT}}}\right]_+ \\ & +\omega_G \frac{\widehat{G}_B(\mathbf{n}_e)}{G_B^{\mathrm{ref}}},\end{aligned} \tag{54}$$

with $[x]_+ = \max\{x, 0\}$, $\omega_F, \omega_P, \omega_G \geq 0$ and $\omega_F + \omega_P + \omega_G > 0$, $d_B^{\mathrm{FT}} = \min_{(u,i)\in B} D_{u,i}^{\mathrm{FT}}$, and $d_B^{\mathrm{PT}} = \min_{(u,i)\in B} D_{u,i}^{\mathrm{PT}}$. The estimate $\widehat{G}_B$ is the number of requests predicted to satisfy TTFT, TPOT, and deadline divided by the predicted batch makespan in Eq. (43). The reference rate $G_B^{\mathrm{ref}} > 0$ normalizes the goodput term, so all three terms in the utility are dimensionless. This form avoids always saturating Prefill first; the selected point balances Prefill throughput against Decode TPOT under the guarded resource constraints.

4) **Dynamic execution adjustment.** We model a profile-guided next-wave controller that treats the fixed work units in Eq. (36) as planning chunks and never changes in-flight work. Sampling is aligned with dispatch waves rather than a free-running timer. A wave contains a Prefill launch, a Decode launch, or both. At its boundary, the controller observes remaining Prefill chunks, the per-request chunks required for the *next* Decode token, elapsed queueing time, and profiled per-wave times L_p and L_d. Because generation is autoregressive, Decode chunks belonging to future token steps are never placed in the same ready set. The controller only reorders ready work and changes the planned Prefill/Decode chunk counts for the next wave. This abstraction can drive a host-side worklist in a CUDA runtime. The current real-serving evaluation does not claim an isolated speedup for this controller; any standalone evaluation of it must be labeled as profile-driven wave replay.

a) Host-side rescue loop.: During Decode, a run-time rescue loop operates at wave-aligned sampling boundaries. It never migrates or relabels in-flight kernel work. Instead, it (i) reorders ready queues and (ii) adjusts the next dispatch boundary (kernel wave) to reduce two predicted SLO risks:

- **Request-level deadlines:** it prioritizes requests whose projected completion time approaches the absolute deadline or per-token (TPOT) SLO.
- **Batch-level TTFT:** it increases planned Prefill service when the Prefill queue threatens the batch-wide time-to-first-token budget.

Per-request slack (Decode phase). For a request i that arrived at time a_i and is still in Decode at time t, its projected deadline slack must preserve token-to-token dependence. Let $\mathcal{J}_i(t)$ be the ordered set of remaining output-token steps, $N_{i,j}^{\mathrm{chunk}}(t)$ the profiled chunks that step j will require when it becomes ready, and $n_i^{\mathrm{plan}}(j,t) > 0$ its planned within-step parallelism. Then

$$\begin{aligned}\widehat{T}_{i,\mathrm{rem}}^{\mathrm{dec}}(t) &= \sum_{j \in \mathcal{J}_i(t)} \left\lceil \frac{N_{i,j}^{\mathrm{chunk}}(t)}{n_i^{\mathrm{plan}}(j,t)} \right\rceil L_{d,i,j}, \\ \Lambda_i^{\mathrm{DL}}(t) &= a_i + \tau_i - \big[t + \Delta_i^{\mathrm{wait}}(t) \\ &\qquad + \widehat{T}_{i,\mathrm{rem}}^{\mathrm{dec}}(t) + T_{\mathrm{down},i}^{\mathrm{rem}}(t)\big].\end{aligned} \tag{55}$$

Here $\Delta_i^{\mathrm{wait}}(t)$ is the time until the next dispatch boundary and $T_{\mathrm{down},i}^{\mathrm{rem}}(t)$ is the remaining delivery time. The sum serializes token steps while permitting parallel chunks within one step. If any remaining step has no positive plan, its projected completion time is $+\infty$. The TPOT slack is tracked separately as

$$\begin{aligned}\Lambda_i^{\mathrm{PT}}(t) &= D_{u,i}^{\mathrm{PT}} - \widehat{T}_i^{\mathrm{PT}}(t), \\ \widehat{T}_i^{\mathrm{PT}}(t) &= \Delta_i^{\mathrm{wait}}(t) + \left\lceil \frac{N_{i,\mathrm{next}}^{\mathrm{chunk}}(t)}{n_i^{\mathrm{plan}}(t)} \right\rceil L_{d,i,\mathrm{next}} \\ &\qquad + T_{\mathrm{down,step},i}(t).\end{aligned} \tag{56}$$

Here $N_{i,\mathrm{next}}^{\mathrm{chunk}}(t)$ denotes the ready work for the next output token only and $n_i^{\mathrm{plan}}(t) > 0$; a zero allocation gives an infinite next-token projection.

Batch-level slack (Prefill queue). Let $Q_p^{\mathrm{chunk}}(t)$ be the total number of remaining Prefill chunks in the current window, $c_p^{\mathrm{plan}}(t) = \sum_{s=1}^{N_{\mathrm{sm}}} n_{p,e,s}^{\mathrm{plan}}(t) > 0$, and let $\bar{t}_B^{\mathrm{FT}} = \min_{(u,i)\in B}(a_{u,i} + D_{u,i}^{\mathrm{FT}})$ be the earliest absolute first-token

deadline among requests in the batch. The batch-level Prefill slack is

$$\Lambda_B^{\mathrm{FT}}(t) = \bar{t}_B^{\mathrm{FT}} - \left[t + \Delta_p^{\mathrm{wait}}(t) + \left\lceil \frac{Q_p^{\mathrm{chunk}}(t)}{c_p^{\mathrm{plan}}(t)} \right\rceil L_p + T_{\mathrm{down},1,B}(t) \right]. \tag{57}$$

Next-dispatch adjustment policy. At each sampling instant t, the scheduler operates only at the next dispatch boundary t^+ and never migrates in-flight kernel work. It constructs a small adjustment $\Delta\mathbf{n}(t)$ that either increases Decode parallelism for the most urgent request or increases Prefill parallelism when the batch-level TTFT slack is negative. Let $\mathcal{X}_g^{\mathrm{plan}}(t)$ denote the vectorized guarded feasible region obtained by aggregating the planned Prefill and per-request Decode chunks into the five resource constraints in $\mathcal{X}_g(t)$. Since planning-chunk counts are integer-valued, the next-wave plan is obtained by enumerating the small set of feasible integer plans and selecting the closest plan to the requested update:

$$\mathbf{n}^{\mathrm{plan}}(t^+) = \underset{\mathbf{z} \in \mathcal{X}_g^{\mathrm{plan}}(t) \cap \mathbb{Z}^d_{\geq 0}}{\arg\min} \left\| \mathbf{z} - \left(\mathbf{n}^{\mathrm{plan}}(t^-) + \Delta\mathbf{n}(t) \right) \right\|_1. \tag{58}$$

Ties are broken in favor of the plan with larger utility in Eq. (54) and then by a fixed ordering. This operation updates only the next-wave plan; it does not relabel or migrate already-running work.

D. Feasible Split-Mode Scheduling under Bandwidth, Compute Resource, or Load Constraints

Edge-side prefill chunks are both register and SMEM intensive. Because all modes share the same UE–AP access path, weak wireless access does not by itself favor Split. Split becomes competitive with Mix when (i) edge Prefill is expensive or fails guarded SM/memory feasibility, (ii) cloud Prefill is faster, (iii) the end-to-end KV-migration path is sufficiently fast, and (iv) edge Decode remains feasible; it is selected only when its complete predicted path is the minimum-latency feasible mode. Window-level burstiness can create this regime when the per-window token total S_k or request total $|B_k|$ has reached its current cap, or the previous-window SLO-violation ratio has exceeded its tolerance. In these cases, keeping prefill at the edge inflates TTFT or degrades TPOT. Split mode (cloud-prefill + edge-decode) therefore places prefill on the cloud side while retaining the lighter decode phase on the edge side.

For a cloud-side prefill planning chunk with profiled vector $(C_{p,C}, R_{p,C}, W_{p,C}, S_{p,C}, T_{p,C}, L_{p,C})$, the maximum number of profiled Prefill chunks per cloud SM is

$$n_{p,C} = \min \left\{ \left\lfloor \frac{C_{s,C}}{C_{p,C}} \right\rfloor, \left\lfloor \frac{R_{s,C}}{R_{p,C}} \right\rfloor, \left\lfloor \frac{S_{s,C}}{S_{p,C}} \right\rfloor, \left\lfloor \frac{W_{s,C}}{W_{p,C}} \right\rfloor, \left\lfloor \frac{T_{s,C}}{T_{p,C}} \right\rfloor \right\}. \tag{59}$$

Given a batch of $N_{p,C}^{\mathrm{chunk}}$ Prefill chunks and $n_{p,C} > 0$, the cloud prefill latency is

$$\widehat{T}_{\mathrm{prefill}}^{\mathrm{cloud}} = \left\lceil \frac{N_{p,C}^{\mathrm{chunk}}}{N_{\mathrm{sm},C} n_{p,C}} \right\rceil L_{p,C}. \tag{60}$$

If $n_{p,C} = 0$, cloud prefill is marked infeasible rather than evaluated by Eq. (60).

Using the edge-side Decode profile, remaining token steps are summed sequentially. For $n_{d,E}(j) > 0$, the estimator is

$$\widehat{T}_{\mathrm{decode,rem}}^{\mathrm{edge}}(u,i) = \sum_{j=2}^{L_{u,i}^{\mathrm{out}}} \left\lceil \frac{N_{u,i,j}^{\mathrm{chunk}}}{N_{\mathrm{sm},E} n_{d,E}(j)} \right\rceil L_{d,E}(j). \tag{61}$$

Any required step with $n_{d,E}(j) = 0$ marks the edge-decode component infeasible. This form permits parallel work within a token step but never parallelizes future autoregressive token steps.

V. Performance

This section evaluates three questions. First, does the online policy execute and select all three intended paths—Cloud, Split, and edge-local Mix—under the conditions predicted by the model? Second, do the network, KV-transfer, queueing, and GPU-resource terms explain the measured crossovers? Third, does guarded Mix admission remain safe when the dominant Prefill and Decode kernels fail the profiled joint-resource bound? We report real serving experiments separately from profile-driven sensitivity studies, and label every profile-driven result explicitly.

A. Experimental Setup and Evidence Protocol

Testbed. The cloud worker uses an NVIDIA A800 80-GB PCIe GPU and the edge worker uses an NVIDIA RTX A6000 48-GB GPU. The integrated experiments run a local Llama-3 checkpoint and Qwen3-8B. The cloud process exposes Prefill and Decode endpoints, while the edge process exposes serialized edge execution, Split KV loading and Decode, and concurrent Mix execution. Unless a sweep changes it, wireless access uses 20/50 Mbps uplink/downlink with 10 ms access delay in each direction; the edge–cloud path uses 1 ms one-way propagation. The controlled backhaul points are 1, 10, and 40 Gbps. The equivalent Linux configuration is `tc qdisc replace dev IFACE root netem delay 1ms rate BWgbit`; jitter and loss are zero in the controlled runs.

The cloud-return sweep folds a non-overlapped per-token RPC/pacing term into the modeled remaining-token Cloud downlink term; it is distinct from fixed edge–cloud propagation.

Workloads and SLOs. Each heterogeneous trace contains equal numbers of Mix-favorable, Split-favorable, and Cloud-favorable requests. Separate traces isolate each favorable regime. The integrated policy, offered-load, and fallback traces use TTFT $\leq$ 1000 ms, TPOT $\leq$ 75 ms, and end-to-end latency $\leq$ 2500 ms. The longer-output network keypoint trace uses TTFT $\leq$ 2000 ms, TPOT $\leq$ 50 ms, and end-to-end latency $\leq$ 4000 ms. All policies compared within one point use the

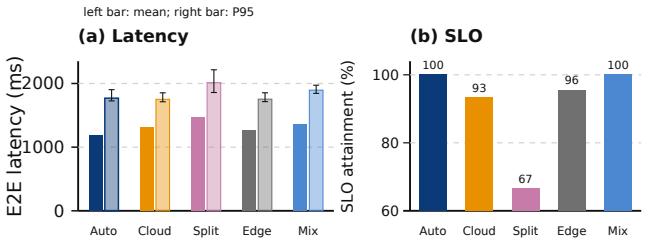

Fig. 2. Real Qwen3-8B heterogeneous replay. Each policy contains three independent repetitions and 45 formal requests. Error bars are bootstrap 95% CIs. Auto lowers mean E2E by 6.4% versus the best fixed mean baseline and reaches 100% SLO; its P95 confidence interval overlaps those of Cloud and edge-local.

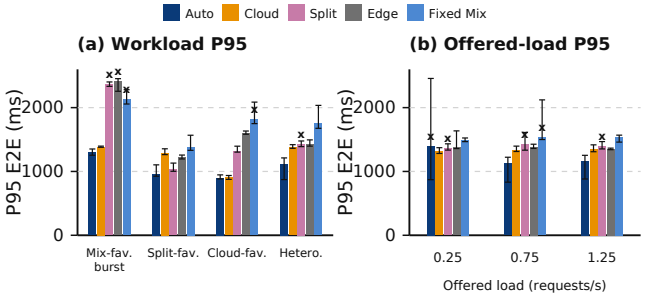

Fig. 3. Real Llama-3 policy matrix and offered-load validation. Panel (a) compares Auto, Cloud, Split, serialized Edge, and fixed Mix over four workloads; each cell has three repetitions and 45 formal requests. Panel (b) is a separately seeded five-policy matrix at three offered loads, again with three repetitions and 45 formal requests per policy/load point. Bars report P95 E2E with bootstrap 95% CIs; an "x" marks SLO attainment below 100%.

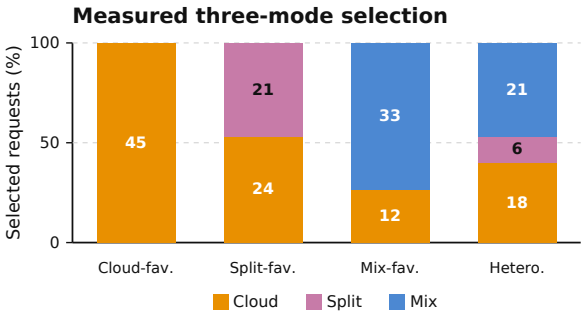

Fig. 4. Measured Auto mode share for the four Llama-3 workloads in Fig. 3(a)–(b), with three repetitions and 45 formal requests per workload. Numbers inside bars are request counts; no modeled mode-region points are used.

same trace and SLO tuple; results from the two trace families are not pooled. The short gate-reject boundary uses measured 47-, 55-, and 62-token prompts; it is a boundary test rather than a representative production mix. Offered load is swept over 0.25, 0.75, and 1.25 requests/s.

Repetition and audit. Every integrated Qwen/Llama policy point has three independent repetitions. Each run starts with four warm-up requests and discards the first post-warm-up request, leaving 15 formal requests. Network keypoints retain 41 formal requests per configuration after their own warm-up/discard protocol. We retain all measurements, including outliers, from the uncontaminated completed runs. P95 error bars are request-bootstrap 95% confidence intervals over the three repetitions.

Baselines. The real end-to-end comparison uses fixed CLOUD, fixed SPLIT, serialized edge-local execution, fixed MIX, and Auto. Auto evaluates only the paper's three primary modes. An ablation named Auto+Edge-fallback decouples edge placement from concurrent Mix admission: when the strict Mix gate rejects concurrency, it may choose a costed serialized edge path. This is a safe-degradation ablation, not a fourth primary execution mode. Profile-driven comparisons additionally include fixed PD-disaggregation and fixed chunked-Prefill proxies inspired by DistServe, vLLM, SGLang, and Sarathi [9], [34]–[36]; their limitations are stated in Sec. V-G.

B. Real End-to-End Mode Selection

Fig. 2 reports the audited Qwen3-8B heterogeneous replay. Auto has the lowest mean end-to-end latency, 1179.2 ms, which is 6.4% below the best fixed mean baseline (serialized edge-local at 1259.5 ms). Auto and fixed Mix both attain 100% SLO, whereas Cloud, Split, and serialized edge-local attain 93.3%, 66.7%, and 95.6%, respectively. Auto does not improve P95: its 1769.4 ms is 1.0% above Cloud and 0.9% above edge-local, and their intervals overlap. Thus this run supports lower mean latency and higher SLO robustness, not a Qwen tail-latency superiority claim.

The strict Qwen Mix gate rejects concurrent Mix for all 45 Auto requests: 30 fail the profiled SLO/resource test and 15 have no predicted benefit. Because fixed Mix nevertheless attains 100% SLO in this trace, the first count also exposes conservatism in the strict profile gate: it can reject measured-feasible concurrency rather than risk an unsafe admission. Consequently, Qwen validates Cloud/Split routing but not Auto Mix selection; the Llama-3 five-policy matrix below supplies complete three-mode coverage. Fig. 3(a)–(b) shows both the favorable and unfavorable results. In the bursty Mix-favorable workload, Auto reduces mean/P95 E2E by 18.1/6.7% relative to the best fixed policy, Cloud, and by 40.6/39.1% relative to fixed Mix; Auto attains 100% SLO, compared with 66.7% for fixed Mix. Under the Split-favorable workload, Auto improves mean/P95 by 4.5/7.0% over fixed Split. In the heterogeneous replay, it improves mean/P95 by 33.0/19.2% over the best fixed policy, Cloud. In the Cloud-favorable workload, Auto correctly selects Cloud for every request; its mean is 0.7% higher and its P95 is 0.6% lower than fixed Cloud.

Fig. 4 reports the Auto decisions from the same primary matrix. Auto selects Cloud 45 times in the Cloud-favorable workload, Cloud/Split 24/21 times in the Split-favorable workload, Cloud/Mix 12/33 times in the bursty Mix-favorable workload, and Cloud/Mix/Split 18/21/6 times in the heterogeneous trace. The last trace therefore exercises all three primary modes through real serving.

Table III(a) summarizes the comparison with the best fixed policy in each workload; Fig. 3 retains all five policies, includ-

TABLE II
COMPLETE MEASURED QWEN3-8B FIXED-POLICY MATRIX. EVERY ROW CONTAINS THREE REPETITIONS AND 45 FORMAL REQUESTS. C/S/E/M ARE SELECTED REQUEST COUNTS FOR CLOUD/SPLIT/SERIALIZED EDGE/MIX; LATENCY VALUES ARE IN MS.

Policy	Mean	P95	95% CI	TTFT P95	TPOT P95	SLO	C/S/E/M
Auto	1179.2	1769.4	[1726.4,1903.1]	824.3	68.8	100.0%	33/12/0/0
Cloud	1316.4	1751.3	[1712.9,1853.4]	855.1	69.7	93.3%	45/0/0/0
Split	1467.8	2013.2	[1858.2,2215.3]	1415.1	61.1	66.7%	0/45/0/0
Edge-local	1259.5	1753.4	[1715.2,1853.8]	990.4	55.1	95.6%	0/0/45/0
Fixed Mix	1362.1	1895.4	[1845.0,1973.1]	716.6	67.3	100.0%	0/0/0/45

TABLE III
PRIMARY MEASURED LLAMA-3 WORKLOAD SUMMARY AND OFFERED-LOAD VALIDATION. EVERY ROW CONTAINS THREE REPETITIONS AND 45 FORMAL REQUESTS; LATENCY VALUES AND BOOTSTRAP CONFIDENCE INTERVALS ARE IN MS. THE TWO PANELS COME FROM SEPARATELY SEEDED CAMPAIGNS AND ARE NOT MERGED REQUEST BY REQUEST.

(a) Primary five-policy workload matrix

Regime	Auto mean	Best fixed mean	Auto P95	Best fixed P95	SLO	Auto modes
Cloud-fav.	691.3	Cloud 686.3	897.6	Cloud 903.1	100%	C45
Split-fav.	758.8	Split 794.8	957.9	Split 1030.3	100%	C24/S21
Mix-fav. burst	948.6	Cloud 1157.9	1293.7	Cloud 1386.3	100%	C12/M33
Heterogeneous	666.7	Cloud 995.6	1114.1	Cloud 1378.3	100%	C18/M21/S6

(b) Heterogeneous five-policy offered-load sweep

Load	Auto mean	Best fixed mean	Auto P95	Best fixed P95	SLO	Modes
0.25	713.7	Cloud 973.2	1402.0	Cloud 1313.0	95.6%	C18/M21/S6
0.75	666.9	Edge 976.9	1124.8	Cloud 1333.2	100%	C18/M21/S6
1.25	676.8	Edge 949.3	1163.3	Edge 1350.3	100%	C18/M21/S6

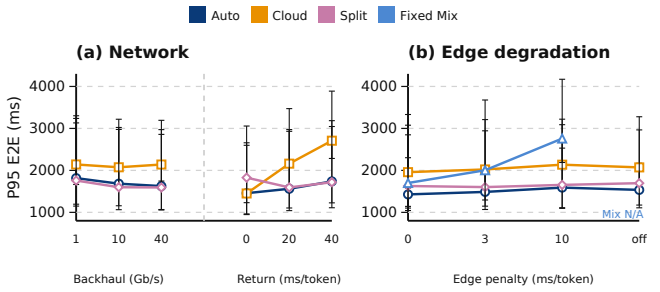

Fig. 5. Single-column real-serving sensitivity: (a) backhaul bandwidth and cloud-return cost, separated by the dashed divider, and (b) Edge-Prefill degradation. Each point has three repetitions and 41 retained formal requests; error bars show bootstrap 95% CIs. Cloud SLO falls to 0% at 40 ms/token, whereas Auto retains 97.6%; fixed Mix is N/A when Edge Prefill is disabled.

ing the unfavorable fixed-Mix and Split outcomes. Panel (b) of the table and panel (b) of the figure use a separately seeded five-policy campaign to evaluate offered load. We do not merge either campaign request by request. The optional serialized-edge path remains a separate safe-degradation ablation in Sec. V-D rather than a fourth primary mode.

C. Network and Compute Sensitivity

Fig. 5 combines the three network and compute sweeps. Every point uses three repetitions; the synchronized summary retains 41 formal requests per configuration. Backhaul improvement from 1 to 40 Gbps lowers Split P95 from 1758.7 to 1593.0 ms and Auto P95 from 1816.0 to 1626.2 ms. Queueing makes the fixed Cloud curve non-monotonic, which is why the plot includes confidence intervals instead of a fitted trend. Because several latency intervals overlap, these P95 point estimates are interpreted as measured sensitivity trends rather than significance claims.

Increasing cloud-return cost from 0 to 40 ms/token raises Cloud P95 by 86.7% and lowers its SLO from 100% to 0%. Auto increases by only 19.3% and retains 97.6% SLO by leaving the remote-cloud path when appropriate.

Conversely, adding 10 ms/token to edge Prefill increases fixed-Mix P95 from 1699.8 to 2758.5 ms, while Auto increases from 1427.7 to 1589.7 ms and retains 100% SLO. When edge Prefill is disabled, fixed Mix is marked N/A; Auto remains valid through Cloud/Split. The two panels therefore expose measured Cloud, Split, and Mix crossover conditions.

D. Load Robustness and Safe Edge Fallback

Panel (b) of Fig. 3 sweeps offered load. Auto's mean E2E is 713.7, 666.9, and 676.8 ms at 0.25, 0.75, and 1.25 requests/s, respectively, improving on the best fixed mean by 26.7–31.7%. Its P95 is 1402.0, 1124.8, and 1163.3 ms. We retain the unfavorable 0.25 requests/s point: one repeat contains a large tail outlier, so Auto's P95 is 6.8% above fixed Cloud and SLO is 95.6%, although its mean remains 26.7% lower. At 0.75 and 1.25 requests/s, Auto reduces P95 by 15.6% and 13.9% relative to the best fixed policy and attains 100% SLO. Its measured queue wait rises from 0.136 to 6.17 and 7.17 ms. Because each load uses the same 15/15/15 regime mix, the Cloud/Mix/Split counts remain 18/21/6; this experiment demonstrates latency robustness, not load-driven mode-share adaptation.

Fig. 6 isolates where the serialized-edge fallback is useful. With short prompts and a strict concurrent-Mix rejection, original Auto chooses Cloud/Split 24/21. The ablation preserves the same 24 Cloud decisions but replaces the 21 Split decisions with serialized edge-local execution. Mean latency falls 7.2%, P95 falls 8.1%, and measured queue wait falls from 22.43 to 0.007 ms. The P95 intervals partially overlap, so this is reported as a mechanism boundary, not a general speedup claim.

The separately seeded Qwen pair in Table V(b) shows a larger boundary benefit, but it remains secondary evidence because it does not share the seed set used by the five-policy headline matrix.

E. Actual KV Transport and Component Accounting

Fig. 7 quantifies the route tradeoff: Cloud and Split pay 238.9 and 276.8 ms of measured queue wait, Cloud pays 267.7 ms of return cost, and Split pays both KV handling and backhaul. Fixed Mix avoids those communication terms in this run but keeps both compute phases at the edge.

TABLE IV

COMPLETE MEASURED NETWORK AND EDGE-PREFILL KEYPOINTS UNDERLYING FIG. 5. EVERY ROW CONTAINS THREE REPETITIONS AND 41 RETAINED FORMAL REQUESTS; P95, CONFIDENCE INTERVALS, AND QUEUE WAIT ARE IN MS. FIXED MIX IS N/A WHEN EDGE PREFILL IS DISABLED.

(a) Backhaul bandwidth and cloud-return cost

Sweep point	Policy	P95	95% CI	SLO	Queue
1 Gbps	Auto	1816.0	[1194.5,3237.9]	97.6%	225.2
1 Gbps	Cloud	2141.2	[1663.8,3301.8]	80.5%	192.6
1 Gbps	Split	1758.7	[1149.4,3131.4]	97.6%	214.5
10 Gbps	Auto	1683.5	[1157.4,2975.3]	97.6%	206.7
10 Gbps	Cloud	2074.2	[1600.6,3218.7]	95.1%	177.4
10 Gbps	Split	1599.0	[1063.6,3022.7]	97.6%	207.3
40 Gbps	Auto	1626.2	[1056.2,2971.5]	97.6%	207.0
40 Gbps	Cloud	2139.8	[1746.0,3193.2]	90.2%	173.1
40 Gbps	Split	1593.0	[1067.7,2857.9]	100.0%	195.1
Return 0	Auto	1456.9	[946.9,2657.8]	100.0%	182.3
Return 0	Cloud	1450.1	[963.4,2597.9]	100.0%	180.8
Return 0	Split	1827.1	[1231.0,3056.9]	97.6%	222.6
Return 20	Auto	1561.6	[1041.2,2972.7]	97.6%	207.0
Return 20	Cloud	2160.6	[1615.5,3471.9]	87.8%	191.5
Return 20	Split	1594.1	[1104.1,2931.0]	97.6%	205.0
Return 40	Auto	1738.4	[1226.5,3045.3]	97.6%	212.1
Return 40	Cloud	2707.5	[2284.5,3888.6]	0.0%	189.1
Return 40	Split	1718.5	[1109.9,3184.8]	97.6%	212.2

(b) Edge-Prefill degradation

Penalty	Policy	P95	95% CI	SLO	Queue
0	Auto	1427.7	[1102.2,2302.2]	100.0%	5.1
0	Cloud	1958.1	[1577.6,3076.6]	100.0%	161.5
0	Mix	1699.8	[1043.7,3331.6]	100.0%	86.8
0	Split	1628.0	[1142.4,2847.7]	100.0%	194.4
3	Auto	1487.1	[1147.6,2056.8]	100.0%	0.0
3	Cloud	2022.6	[1595.4,3210.6]	100.0%	175.1
3	Mix	2003.2	[1294.7,3677.2]	100.0%	88.9
3	Split	1602.1	[1068.3,2943.1]	97.6%	200.9
10	Auto	1589.7	[1111.6,2530.7]	100.0%	12.0
10	Cloud	2135.9	[1689.0,3220.0]	100.0%	182.4
10	Mix	2758.5	[2188.4,4171.3]	95.1%	95.7
10	Split	1654.2	[1094.7,3088.8]	97.6%	209.3
Disabled	Auto	1535.5	[1108.2,2170.6]	100.0%	42.8
Disabled	Cloud	2071.4	[1683.9,3278.1]	100.0%	180.5
Disabled	Mix	N/A	N/A	N/A	N/A
Disabled	Split	1695.8	[1173.8,2964.8]	97.6%	209.7

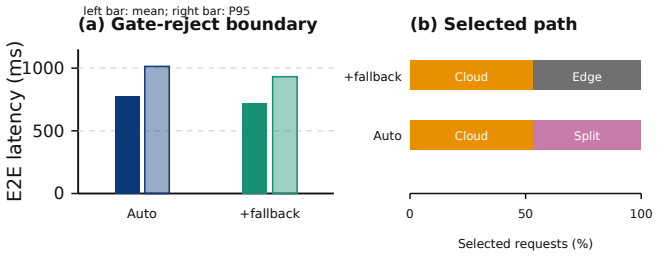

Fig. 6. Short-prompt gate-reject boundary, three repetitions and 45 formal requests per policy. The serialized-edge ablation preserves Cloud decisions, replaces 21 Split decisions, lowers mean/P95 by 7.2/8.1%, and nearly removes the measured edge-queue buildup.

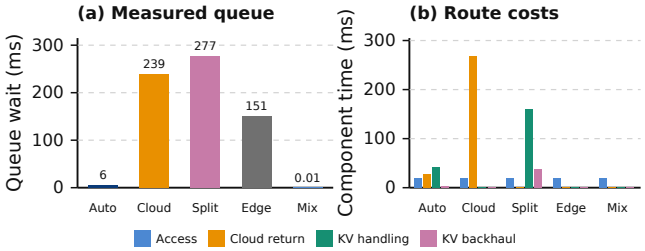

Fig. 7. Measured queue and route-specific costs at 0.75 requests/s. Access, cloud-return, KV-handling, and KV-backhaul bars are shown side by side rather than stacked because separately timestamped components may overlap. Auto is a path-weighted aggregate, not one physical route.

TABLE V

COMPLETE MEASURED FALLBACK-BOUNDARY RESULTS. EACH ROW CONTAINS THREE REPETITIONS AND 45 FORMAL REQUESTS. THE QWEN PAIR USES A SEPARATELY SEEDED RERUN AND IS NOT MERGED WITH TABLE II.

(a) Short-prompt concurrent-Mix rejection

Policy	Mean	P95	95% CI	Queue	SLO	Modes
Auto	774.6	1013.9	[985.9,1135.0]	22.430	100%	C24/S21
Auto+fallback	718.7	931.6	[897.2,1037.6]	0.007	100%	C24/E21

(b) Separately seeded Qwen fallback rerun

Policy	Mean	P95	95% CI	Queue	SLO	Modes
Auto	1120.0	1676.0	[1648.5,1854.2]	206.32	100%	C33/S12
Auto+fallback	828.9	1078.0	[1029.7,1216.7]	8.23	100%	C24/E15/S6

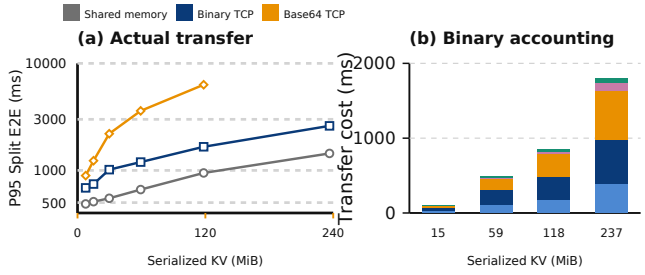

Fig. 8. Actual process-boundary KV transfer, five trials per point. Panel (a) uses a logarithmic y-axis. Binary TCP is consistently faster than Base64 but 1.4–1.9× slower than same-host shared memory in P95 E2E. Component bars exclude edge Decode and are therefore transfer-side costs rather than an E2E stack.

The Split path serializes the actual KV tensors and transfers them across a real HTTP/TCP process boundary. This measurement is loopback on the same physical host; it captures serialization, socket, body-read, deserialization, GPU-load, and Decode costs, but it is not described as an inter-host network measurement. Linux rate/delay emulation remains responsible for controlled backhaul sensitivity.

At 15.0, 59.3, 118.4, and 236.5 MiB, binary-TCP P95 is 745.4, 1195.8, 1662.2, and 2606.1 ms, compared with 510.3, 661.9, 946.6, and 1440.4 ms for shared memory. For 118.4 MiB, the measured means are 180.5 ms serialization, 313.0 ms TCP-response overhead, 295.1 ms body read, 31.9 ms deserialization, 30.3 ms GPU load, and 536.0 ms edge Decode. Base64 reaches 6313.0 ms P95 at 118.4 MiB and is not run at 236.5 MiB. Split is therefore viable only when fast cloud Prefill and near-user Decode amortize a KV path whose cost grows with prompt length.

Table VI reports server enqueue/start/end queue timestamps

TABLE VI

MEASURED MEAN COMPONENT TIMES AT 0.75 REQUESTS/S (MS). COMPONENTS ARE SEPARATELY TIMESTAMPED AND MAY OVERLAP; ROWS ARE NOT ADDITIVE STACKS. "KV PATH" IS SERIALIZE/WRITE/BACKHAUL/READ/DESERIALIZE/GPU-LOAD.

Policy	Access	Queue	Cloud P/D	KV path	Edge P/D	Cloud return	TTFT	TPOT	E2E
Auto	20.1	6.2	33.5/407.4	53.1	140.9/535.0	27.1	169.2	30.9	666.9
Cloud	20.1	238.9	53.0/393.3	0	0/0	267.7	337.7	42.9	981.5
Split	20.1	276.8	54.9/–	197.7	–/432.0	0	574.5	33.8	1082.2
Edge-local	20.1	151.4	0/0	0	80.3/427.0	0	474.5	33.4	976.9
Fixed Mix	20.1	0.01	0/0	0	108.0/633.2	0	369.0	39.0	1077.6

TABLE VII

COMPLETE MEASURED KV-TRANSPORT KEYPOINTS, FIVE TRIALS PER POINT. VALUES ARE P95 END-TO-END LATENCY IN MS; BASE64 AT 236.5 MIB WAS NOT RUN.

Prompt	KV MiB	Shared memory	Binary TCP	Base64
120	15.0	510.3	745.4	1231.2
474	59.3	661.9	1195.8	3599.2
947	118.4	946.6	1662.2	6313.0
1892	236.5	1440.4	2606.1	N/A

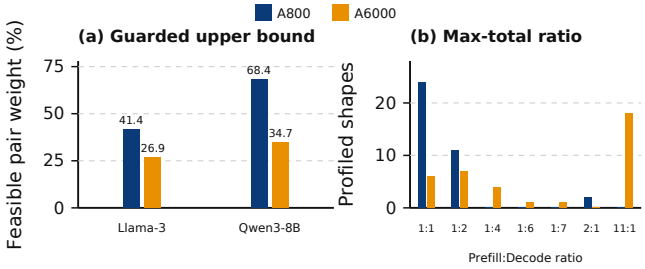

Fig. 9. Real-launch resource analysis. Guarded alternatives provide mean feasible-pair weight of 41.4/26.9% for Llama and 68.4/34.7% for Qwen on A800/A6000. The right panel reports the Llama-3 max-total profile-selected upper-bound ratios, not enforced resident CTA counts.

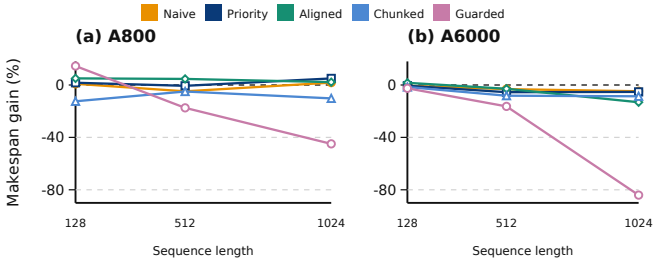

Fig. 10. Steady-state real Qwen3-8B Mix outcomes after discarding repeat 0, with six retained records per policy/point. Lines show all five concurrent-Mix policies. Positive makespan gain alone is insufficient: admission also requires a TPOT-SLO rate of at least 95%.

rather than cost-model queue estimates. Auto's row is an average over different selected paths and is not one physical route.

F. CUDA Launch-Resource Feasibility and Mix Admission

We parse actual CUDA launches from Llama Prefill/Decode ranges on both GPUs: 37 sequence/batch shapes per GPU, 781,392 A800 launches, and 735,504 A6000 launches. For each shape, the resource test uses measured launch threads, registers, and static/dynamic shared memory. It computes a per-SM resource-feasible upper bound; it does not claim that the application pins a specific number of resident CTAs or that achieved occupancy was measured. CUDA retains control of physical residency. For a candidate kernel pair, its time weight is the product of the Prefill and Decode kernels' fractions of profiled GPU time. "Feasible-pair weight" is the sum of those weights over pairs that pass the bound, normalized by the sum over all analyzed pairs for that shape.

The dominant kernel pair fails the derived joint-residency resource bound for every Llama shape on both GPUs. Guarded alternative pairs nevertheless account for a mean 41.4% of analyzed top-pair weight on A800 and 26.9% on A6000. The conservative balanced plan is 1:1 for all 37 A800 shapes and effectively 1:1 for all 37 A6000 shapes. The max-total profile is architecture dependent: A800 selects 1:1 for 24 shapes, 1:2 for 11, and 2:1 for 2; A6000 often selects 11:1 (18 shapes), but also 1:1, 1:2, 1:4, 1:6, and 1:7. Qwen3-8B shows the same architecture dependence: its mean feasible-pair weight is 68.4% on A800 and 34.7% on A6000. We therefore use 1:1 only as a conservative reference; the operational allocation is profile-selected.

Table IX reports the maximum-gain policy at each point. Among those six maximum-gain policies, only A800 at sequence 1024 passes the combined makespan/TPOT rule (Priority Mix: 5.0% makespan reduction and 96.8% TPOT SLO). A800 at 128 also has a lower-gain Naive policy that passes the gate (0.9%, 98.4%), whereas its maximum-gain policy violates the TPOT guard. A6000 at 128 improves makespan by 1.7% but reaches only 94.1% TPOT SLO; at 512 and 1024 even the best measured Mix is 2.9% and 4.8% slower than sequential execution. Thus the scheduler must fall back when concurrent Mix is not both profitable and SLO-safe.

G. Profile-Driven Baselines and Claim Boundary

The multi-model sensitivity matrix covers Qwen3-1.7B, Qwen2.5-3B, Qwen3-8B, and Qwen2.5-14B on GSM8K, SVAMP, ASDiv, Alpaca, XSum, and MBPP. It uses real prompt distributions, model configurations for KV sizing, and measured A800/A6000 service profiles. The DistServe-style PD proxy fixes every request to cloud-Prefill/edge-Decode Split and retains the measured wireless, cloud Prefill, KV serialization/transfer/load, and edge Decode components; it omits DistServe's scheduler, replica placement, batching, cache manager, kernels, and transport runtime. Each chunked-Prefill proxy fixes placement to Cloud, Split, or edge Mix and multiplies only the corresponding profiled Prefill-latency slope by a gain factor g. Decode latency, KV size and handling,

TABLE VIII
COMPLETE ARCHITECTURE-LEVEL LAUNCH-RESOURCE SUMMARY. "FEASIBLE WEIGHT" IS THE FRACTION OF ANALYZED KERNEL-PAIR TIME WEIGHT PASSING THE RESOURCE UPPER BOUND, NOT MEASURED ACHIEVED OCCUPANCY.

Model	GPU	Shapes	Dominant feasible	Mean weight	Range	Balanced	Max-total profile
Llama-3	A800	37	0/37	41.43%	35.79–51.73%	1:1 (37)	1:1(24), 1:2(11), 2:1(2)
Llama-3	A6000	37	0/37	26.87%	21.45–32.34%	1:1-equiv. (37)	11:1(18), 1:1(6), 1:2(7), other(6)
Qwen3-8B	A800	3	2/3	68.39%	66.59–69.67%	1:1 (3)	1:1(3)
Qwen3-8B	A6000	3	0/3	34.66%	29.51–37.90%	1:1 (3)	1:4(2), 1:6(1)

TABLE IX
MAXIMUM-GAIN MEASURED CONCURRENT-MIX POLICY PER GPU/SEQUENCE POINT AFTER DISCARDING REPEAT 0. GAIN IS RELATIVE TO SEQUENTIAL EDGE-LOCAL EXECUTION; "GATE" APPLIES TO THE DISPLAYED MAXIMUM-GAIN POLICY.

GPU	Seq.	Mix	Serial	Gain	TPOT SLO	Gate
A800	128	2076	2428	14.48%	93.01%	Reject
A800	512	2277	2388	4.64%	93.55%	Reject
A800	1024	2203	2319	5.01%	96.77%	Admit
A6000	128	2327	2367	1.67%	94.09%	Reject
A6000	512	2317	2253	−2.88%	82.80%	Reject
A6000	1024	2210	2109	−4.77%	85.48%	Reject

links, queue model, and serving location are unchanged. These proxies omit the original vLLM, SGLang, and Sarathi schedulers, kernels, cache managers, batching, preemption, and admission policies. The fixed modes and static-threshold heuristic are internal comparators rather than reproductions of external systems. Because none of these proxies is functionally identical to the named system, we retain the "-style proxy" label throughout. The numerical profile/proxy tables and the chunk-gain sensitivity figure are provided in the separately submitted supplementary appendix; they are not direct end-to-end SOTA reproductions.

In a separate strict-gate profile replay over the same 24 model–dataset combinations, sweeping the assumed chunk-Prefill gain over 0.50, 0.75, 0.90, and 1.00 changes mean Auto P95 reduction only from 46.85% to 47.22%; the minimum across 24 settings ranges from 34.87% to 37.71%, and no setting makes Auto slower than the best static proxy. This strict-gate campaign is not pooled with the primary proxy matrix in the separately submitted supplementary appendix; its absolute latency and mode counts are therefore not compared with those tables.

VI. CONCLUSIONS

In this work, we introduced a latency-aware scheduler that couples wireless access, KV-cache movement, queueing, and guarded GPU resource feasibility to choose among cloud-only, cloud-Prefill/edge-Decode Split, and edge-local Mix execution. Repeated A800/A6000 serving experiments execute all three paths. In the Qwen3-8B replay, Auto lowers mean end-to-end latency by 6.4% relative to the best fixed mean baseline and attains 100% SLO, while its P95 is not statistically better than fixed Cloud or edge-local execution. The Llama-3 replay exercises Cloud/Mix/Split decisions, attaining 100% SLO across the four primary workload regimes and 95.6--100% across the offered loads. Actual process-boundary KV transfer quantifies the serialization and transport cost that limits Split, and real CUDA-launch traces show that the dominant Prefill/Decode pair fails the joint-residency resource bound for all 37 Llama shapes on both GPUs, even though lower-weight alternatives satisfy that bound. The resulting evidence supports the central design principle: concurrent Mix must be admitted only when it is both profitable and SLO-safe, with Split and Cloud serving as measured fallbacks. Future work will evaluate inter-host high-speed KV fabrics, achieved occupancy counters, and multi-model, multi-tenant deployments.

REFERENCES

[1] J. Bai, S. Bai, and Y. C. et al., "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[2] OpenAI, J. Achiam, S. Adler, and S. A. et al., "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2024.

[3] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," *arXiv preprint arXiv:2401.02385*, 2024.

[4] S. Zhang, S. Roller, and N. G. et al., "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[5] R. Anil, A. M. Dai, and O. F. et al., "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[6] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[7] A. Grattafiori, J. Ainslie, S. Bhosale *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[8] A. Yang, J. Bai, S. Gu *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[9] 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 Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[10] A. K. Kamath, R. Prabhu, J. Mohan, S. Peter, R. Ramjee, and A. Panwar, "Pod-attention: Unlocking full prefill-decode overlap for faster llm inference," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025, pp. 897–912.

[11] K. Hong, X. Li, L. Chen, Q. Mao, G. Dai, X. Ning, S. Yan, Y. Liang, and Y. Wang, "Sola: Optimizing slo attainment for large language model serving with state-aware scheduling," in *Eighth Conference on Machine Learning and Systems*, 2025.

[12] Y. Tang, T. Lan, X. Huang, H. Lu, and W. Chen, "Scorpio: Serving the right requests at the right time for heterogeneous slos in llm inference," *arXiv preprint arXiv:2505.23022*, 2025.

[13] L. Zheng *et al.*, "Lmsys-chat-1m: Real-world multi-turn llm conversations," *arXiv preprint arXiv:2309.11998*, 2024.

[14] R. Fan, X. Yu, P. Dong, Z. Li, G. Gong, Q. Wang, W. Wang, and X. Chu, "Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus," in *Proceedings of the European Conference on Computer Systems*, 2025, pp. 243–260.

[15] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[16] H. Asghar and E.-S. Jung, "A survey on scheduling techniques in the edge cloud: Issues, challenges and future directions," *arXiv preprint arXiv: 2202.07799*, 2022.

[17] X. Zhang, J. Nie, Y. Huang, G. Xie, Z. Xiong, J. Liu, D. Niyato, and X. Shen, "Beyond the cloud: Edge inference for generative large language models in wireless networks," *IEEE Transactions on Wireless Communications*, vol. 24, no. 1, pp. 643–658, 2025.

[18] Z. Liu, Q. Lan, and K. Huang, "Resource allocation for multiuser edge inference with batching and early exiting," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 4, pp. 1186–1200, 2023.

[19] S. Duan, D. Wang, J. Ren, F. Lyu, Y. Zhang, H. Wu, and X. Shen, "Distributed artificial intelligence empowered by end-edge-cloud computing: A survey," *IEEE Commun. Surv. Tutorials*, vol. 25, no. 1, pp. 591–624, 2023.

[20] 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," *arXiv preprint arXiv:2312.11514*, 2024.

[21] T. Wang, R. Fan, M. Huang, Z. Hao, K. Li, T. Cao, Y. Lu, Y. Zhang, and J. Ren, "Ripple: Accelerating llm inference on smartphones with correlation-aware neuron management," *arXiv preprint arXiv:2410.19274*, 2024.

[22] X. Shen, P. Dong, L. Lu, Z. Kong, Z. Li, M. Lin, C. Wu, and Y. Wang, "Agile-quant: activation-guided quantization for faster inference of llms on the edge," *arXiv preprint arXiv:2312.05693*, 2023.

[23] J. Lin, J. Tang, H. Tang, S. Yang, and W.-M. C. et al., "Awq: Activation-aware weight quantization for llm compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2024.

[24] Y. Liang, H. Chen, S. Han, and Z. Liu, "Paroquant: Pairwise rotation quantization for efficient reasoning llm inference," *arXiv preprint arXiv:2511.10645*, 2025.

[25] A. Tseng, Q. Sun, D. Hou, and C. D. Sa, "Qtip: Quantization with trellises and incoherence processing," *arXiv preprint arXiv:2406.11235*, 2025.

[26] X. Hu, Y. Cheng, D. Yang, Z. Xu, Z. Yuan, J. Yu, C. Xu, Z. Jiang, and S. Zhou, "Ostquant: Refining large language model quantization with orthogonal and scaling transformations for better distribution fitting," *arXiv preprint arXiv:2501.13987*, 2025.

[27] X. Lin, Z. Zhang, P. Yue, H. Li, J. Zhang, B. Fan, H. Su, and X. Gong, "Syncintellects: Orchestrating llm inference with progressive prediction and qos-friendly control," in *IEEE/ACM International Symposium on Quality of Service*, 2024, pp. 1–10.

[28] D. Park and B. Egger, "Improving throughput-oriented llm inference with cpu computations," in *International Conference on Parallel Architectures and Compilation Techniques*, 2024, pp. 233–245.

[29] L. Qian, P. Yang, M. Xiao, O. A. Dobre, M. D. Renzo, J. Li, Z. Han, Q. Yi, and J. Zhao, "Distributed learning for wireless communications: Methods, applications and challenges," *IEEE J. Sel. Top. Signal Process.*, vol. 16, no. 3, pp. 326–342, 2022.

[30] J. Bang, J. Lee, K. Shim, S. Yang, and S. Chang, "Crayon: Customized on-device llm via instant adapter blending and edge-server hybrid inference," *arXiv preprint arXiv:2406.07007*, 2024.

[31] Z. Yang, Y. Yang, C. Zhao, Q. Guo, W. He, and W. Ji, "Perllm: Personalized inference scheduling with edge-cloud collaboration for diverse llm services," *arXiv preprint arXiv:2405.14636*, 2024.

[32] A. Mudvari, Y. Jiang, and L. Tassiulas, "Splitllm: Collaborative inference of llms for model placement and throughput optimization," *arXiv preprint arXiv:2410.10759*, 2024.

[33] Y. Ding, C. Niu, F. Wu, S. Tang, C. Lyu, and G. Chen, "Enhancing on-device llm inference with historical cloud-based llm interactions," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 597–608.

[34] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Efficient execution of structured language model programs," *arXiv preprint arXiv:2312.07104*, 2024.

[35] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills," *arXiv preprint arXiv:2308.16369*, 2023.

[36] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[37] K. Zhu, Y. Zhao, L. Zhao, G. Zuo, Y. Gu, D. Xie, Y. Gao, Q. Xu, T. Tang, Z. Ye, K. Kamahori, C.-Y. Lin, S. Wang, A. Krishnamurthy, and B. Kasikci, "NanoFlow: Towards optimal large language model serving throughput," *arXiv preprint arXiv:2408.12757*, 2024.

Xinghan Wang is currently a Postdoctoral Researcher with Pengcheng Laboratory, Shenzhen, China. His research interests include federated learning, distributed optimization, and edge intelligence.

Xiaoxiong Zhong received the Ph.D. degree in computer science from Harbin Institute of Technology, China, in 2015. From 2016 to 2018, he was a Postdoctoral Research Fellow with Tsinghua University, China. He is currently an Associate Professor with Pengcheng Laboratory, Shenzhen, China. His research interests include network protocols, the Internet of Things, edge computing, and AI for networks.

Weihong Yang received the Ph.D. degree in computer science and technology from Harbin Institute of Technology, China, in 2022. He is currently an Assistant Researcher with the Department of New Networks, Pengcheng Laboratory, Shenzhen, China. From 2022 to 2024, he was a Senior Engineer with Huawei, where he worked on network protocols and related technologies. His research interests include network protocols, network optimization, and distributed systems.

Fangming Liu (S'08–M'11–SM'16) received the B.Eng. degree from Tsinghua University, Beijing, China, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud and edge computing, data-center and green computing, SDN/NFV/5G, and applied machine learning and artificial intelligence. He received the National Natural Science Fund of China for Excellent Young Scholars and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Awards of IEEE/ACM IWQoS 2019, ACM e-Energy 2018, and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of the Ministry of Education of China, and the Second Class Prize of the National Natural Science Award of China.

Weizhe Zhang (Senior Member, IEEE) received the Ph.D. degree from Harbin Institute of Technology, China, in 2006. He is currently a Professor with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China, and the Director of the Department of New Networks, Pengcheng Laboratory, Shenzhen, China. His research interests include cyberspace security, cloud computing, and high-performance computing. He is a Lifetime Member of the ACM.