USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Enabling SLO-Aware 5G Multi-Access Edge Computing with SMEC

Xiao Zhang and Daehyeok Kim, *University of Texas at Austin*

https://www.usenix.org/conference/nsdi26/presentation/zhang-xiao

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Enabling SLO-Aware 5G Multi-Access Edge Computing with SMEC

Xiao Zhang Daehyeok Kim

The University of Texas at Austin

Abstract

Multi-access edge computing (MEC) promises to enable latency-critical applications by bringing computational power closer to mobile devices, but our measurements on commercial MEC deployments reveal frequent SLO violations due to high tail latencies. We identify resource contention at the RAN and the edge server as the root cause, compounded by SLO-unaware schedulers. Existing SLO-aware approaches require RAN–edge coordination, making them impractical for deployment and prone to poor performance due to coordination delays, limited heterogeneous application support, and ignorance of edge resource contention. This paper introduces SMEC, a practical, SLO-aware resource management framework that facilitates deadline-aware scheduling through fully decoupled operations at the RAN and edge servers. Our key insight is that standard 5G protocols and application behaviors naturally provide information exploitable for SLO-aware management without extensive infrastructure or application changes. Evaluation on our 5G MEC testbed shows that SMEC achieves 90–96% SLO satisfaction versus under 6% for existing approaches, while reducing tail latency by up to 122×. We have open-sourced SMEC at https://github.com/smec-project.

1 Introduction

Multi-access edge computing (MEC) brings computational power closer to mobile devices by connecting with 5G cellular networks [4, 18]. It enables latency-critical (LC) applications, from smart stadium [21, 48] and AR/VR [17, 39] to cloud gaming [15, 16] and autonomous driving [13] to offload their compute-intensive tasks to edge servers. These applications typically operate through request-response interactions between clients and edge servers, where each request and response may span multiple packets, and each application must meet strict service-level objectives (SLOs) on request-to-response latency.

Unfortunately, today's MEC deployments fall short of this promise. Our measurement studies across three cities from three countries reveal that commercial MEC services suffer from high tail latencies that frequently violate application SLOs (§2). The root cause is resource contention at both the radio access network (RAN) and edge servers, compounded by resource schedulers that lack SLO-awareness, causing resource allocation decisions to directly impact request progress and SLO satisfaction.

Existing SLO-aware scheduling approaches suffer from practical limitations that prevent real-world deployment. Systems like Tutti [56] and ARMA [57] require explicit coordination between RAN schedulers and edge applications, which is impractical because RAN infrastructure and edge servers are typically managed by different entities (*e.g.,* Verizon operates the RAN while AWS provides edge compute). Even if such coordination were feasible, we find that feedback delays between edge servers and RAN prevent timely resource allocation when applications need it most. Moreover, these solutions focus narrowly on specific application types while ignoring edge compute resource management, failing to address the heterogeneous nature of MEC workloads where multiple applications with diverse SLO requirements compete for both network and compute resources.

This paper introduces SMEC,[1] a practical SLO-aware resource management framework that operates through complete decoupling. SMEC runs entirely independent resource managers at the RAN and edge server, each making deadline-aware resource allocation decisions without any coordination between them. Both managers prioritize latency-critical applications approaching their deadlines by estimating *remaining time budgets*, enabling timely resource allocation when applications need it most. This decoupled design enables deployment in realistic settings where different entities control the RAN and edge infrastructure. SMEC supports heterogeneous workloads, ensuring LC tasks meet their deadlines without starving best-effort (BE) traffic.

While our approach sounds promising, designing such a decoupled framework introduces three key technical challenges. First, the RAN scheduler must identify application request boundaries to estimate time budgets without packet payload inspection due to strict timing constraints. Second, the edge server must estimate the uplink transmission time already consumed by incoming requests and predict the future downlink transmission time for responses, all without visibility into RAN delays. Third, the edge server also must predict processing time across heterogeneous workloads without requiring intrusive application modifications.

Our key insight is that standard 5G protocols and applications' request-response patterns provide useful signals to solve these challenges independently. We find that these signals can be exposed with minimal or no application modifi-

[1] Short for **S**LO-aware **MEC**

cations. First, we exploit patterns in 5G control signals (*e.g.,* Buffer Status Reports) to infer new request arrivals at the RAN without payload inspection. Second, at the edge, we leverage the stability of downlink transmissions to estimate network latency via a lightweight probing protocol and a client-side API without coordinating with the RAN. Third, for processing time estimation at the edge, we exploit key lifecycle events of requests exposed through a server-side API, enabling SMEC to track execution history and predict processing times for incoming requests. Based on this information, SMEC enables the RAN and the edge to compute the remaining time budget for requests independently and manage resources based on these budgets.

We implemented SMEC as user-space resource managers in C++ and Python that run across the RAN, edge servers, and client devices. At the RAN, our resource manager operates as a pluggable scheduling module for srsRAN's MAC layer, implementing request identification and deadline-aware scheduling without affecting other RAN functionalities. At the edge, our resource manager runs as a user-space daemon that estimates network and processing times while dynamically managing CPU and GPU allocations. Client devices run a lightweight timing daemon to support the probing protocol for network latency estimation. We have open-sourced SMEC, including the evaluated applications and experiment scripts: `https://github.com/smec-project`.

We evaluated SMEC on our private 5G MEC testbed using three LC applications and one BE application running on 12 client devices. SMEC achieves 90–96% SLO satisfaction under both static and dynamic workloads, compared to less than 6% for existing approaches. It reduces tail latency by up to 122× for uplink-intensive applications and consistently improves P99 latency by 2–89× across all workloads. Importantly, SMEC allows BE applications to fairly share remaining bandwidth without prolonged starvation.

2 Background and Motivation

2.1 Primer on Multi-Access Edge Computing

Multi-Access Edge Computing (MEC) brings computational resources closer to end users by deploying compute infrastructure close to cellular base stations [20]. The MEC architecture consists of three key components: the RAN that manages wireless spectrum and radio resource scheduling, edge servers that provide CPU and GPU resources, and the core network that connects the RAN and the servers. In the typical MEC request-response model, LC applications offload compute-intensive tasks from client devices to edge servers. Clients send requests via RAN uplink, edge servers process them locally, and responses return via RAN downlink.

Table 1 shows the MEC applications we focus on in this paper, each with distinct SLO, network load, and compute resource requirements. We target realistic MEC deployments where multiple user equipment (UE) devices (*e.g.,* smartphones, cameras, AR headsets) simultaneously offload compute-intensive tasks to edge servers.

Applications	Offloaded Task	SLO	UL/DL Load	Compute Resource
Smart stadium [21]	Video transcoding	100ms	High/High	CPU
Augmented reality [40, 45]	Object detection	100ms	Med/Low	GPU
Video conferencing [2, 25]	Super resolution	150ms	Low/High	GPU

Table 1: Examples of MEC applications evaluated in this paper, each with distinct SLO, network (uplink/downlink) load, and compute resource requirements. Details are described in §7.1.

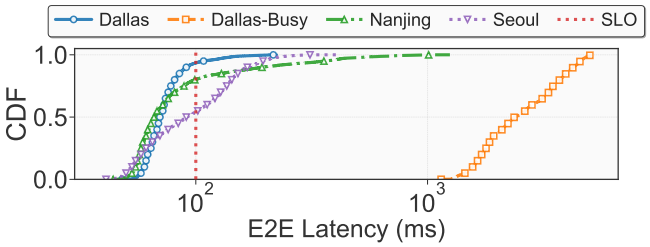


Figure 1: End-to-end latency for the smart stadium application without edge resource contention across MEC deployments in three cities. The dotted red line indicates the SLO.

However, as we demonstrate next, the reality of current MEC deployments falls short of these performance promises.

2.2 Unpredictable Performance of MEC

We benchmarked MEC deployments across three cities in the US, South Korea, and China (Dallas, Seoul, and Nanjing) to quantify this performance gap.

Measurement setup. We deployed the smart stadium and augmented reality applications from Table 1 using a laptop client connected to a 5G smartphone hotspot and edge VMs provisioned through edge service providers (*e.g.,* AWS in the US). Each application runs in isolation (*i.e.,* no contention on the VM), generates 10,000 requests, and we measure end-to-end latency from request transmission to complete response reception. For brevity, we focus our analysis on the smart stadium application, presenting results from all three cities that employ different combinations of cellular operators and cloud providers. We observe similar trends for the augmented reality application across all cities (§A.1).

Results. Figure 1 reveals unpredictable performance characterized by high tail latencies across all deployments, even without compute resource contention on the edge server. Specifically, 7%, 20% and 47% of requests exceed their SLO requirements in Dallas, Nanjing, and Seoul, respectively, during low network activity periods (measured at 2am). Although median latencies remain below the SLO threshold, the P95 and P99 latencies are substantially higher, resulting in an inconsistent user experience. This problem intensifies under higher network load: when additional UE devices access the 5G network

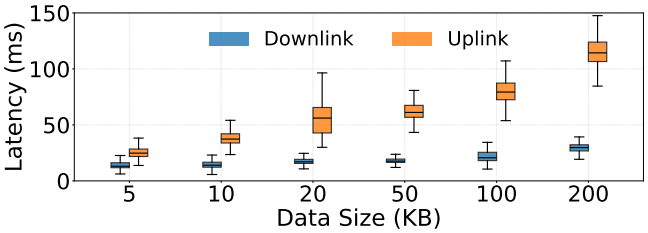


Figure 2: Network latency variability for uplink and downlink transmissions across different data sizes in Dallas. Data size refers to request size for uplink and response size for downlink.

and increase contention for RAN resources (Dallas-Busy), even the median latency exceeds the SLO requirements.

2.3 Root Causes of Unpredictability

To understand the sources of this unpredictability, we investigate what contributes to the tail latency in both the RAN and the edge server.

2.3.1 RAN Resource Contention

To analyze RAN-induced latency variability, we decompose end-to-end latency into uplink and downlink transmission phases. Using Precision Time Protocol (PTP) [14] for clock synchronization between client device and edge server via a stable out-of-band wired connection, we developed a synthetic application that measures uplink and downlink latency separately while varying request and response sizes. Specifically, we measured the time to receive complete requests at the server (uplink) and complete responses at the client (downlink) while varying data size.

Figure 2 reveals an asymmetry that explains MEC's unpredictable performance: uplink latency exhibits high variability, especially for larger request sizes, while downlink latency remains stable. This reflects how cellular networks provision fewer uplink slots than downlink slots, causing higher contention for uplink transmissions. We observe similar trends across all measured cities (§A.3).

To understand the specific mechanisms causing uplink jitter, we leveraged our srsRAN 5G [12]-based testbed for in-depth analysis (details in §7.1), since we have no visibility into the internals of public cellular networks. The srsRAN stack employs the proportional fair (PF) scheduling algorithm [33, 35] used in commercial deployments, allowing us to emulate real-world scheduling behavior.

We ran the smart stadium application on the testbed and monitored MAC layer state by collecting buffer status reports (BSRs) sent from each UE. As a background workload, we also deployed five file transfer UEs. BSRs indicate remaining data in UE-side transmission buffers, providing direct insight into whether the scheduler allocates sufficient RAN resources[2] to meet application demands.

[2]We use the term RAN resources to refer to uplink and downlink spectrum resources managed by the MAC scheduler (*i.e.,* Physical Resource Blocks in 5G).

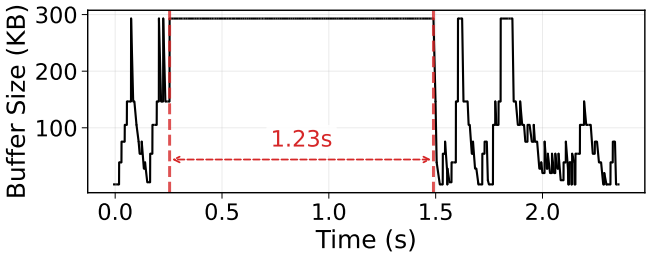


Figure 3: Smart stadium UE's uplink buffer status changes over time. 300 KB is the maximum for BSR from UE to the RAN, which means UE may buffer more than 300 KB.

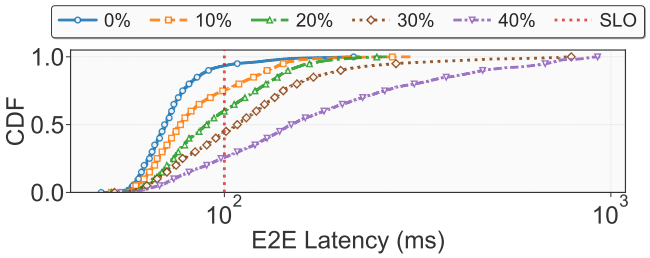


Figure 4: End-to-end latency for smart stadium under different levels of compute resource contention in Dallas. The dotted red line indicates the SLO.

Figure 3 shows persistent non-zero BSR (>1s), indicating uplink starvation from PF scheduling. This resource starvation directly translates to the observed uplink jitter; inadequate resource allocation causes unpredictable buffering delays that lead to SLO violations.

The root cause is PF schedulers' lack of SLO awareness. They prioritize UEs with relatively better channel conditions to their historical average throughput, balancing fairness and efficiency without considering SLO requirements. When multiple UEs compete for uplink resources, LC applications cannot obtain timely allocations, missing SLO requirements even when aggregate bandwidth is sufficient.

2.3.2 Edge Compute Resource Contention

While wireless resource allocation causes unpredictable uplink performance, edge compute resources present an additional bottleneck in MEC deployments. To quantify the impact of compute resource contention, we emulate competing offloaded tasks by running a CPU stressor (stress-ng [27]) with varying CPU usage levels. Figure 4 shows that compute resource contention significantly contributes to unpredictable performance of the smart stadium application. As CPU load increases, tail latencies grow substantially, confirming that inadequate management of edge compute resources causes processing delays that cascade into SLO violations. This problem extends to GPU-intensive applications and other MEC deployments (§A.2).

2.4 Prior Approaches and Limitations

Having identified the root causes of unpredictable MEC performance, we now examine existing resource management approaches and discuss why they fall short for our setting.

MEC-specific resource management. Recent MEC resource management proposals suffer from impractical deployment requirements and limited scope as we show in §7. Tutti [56] assumes explicit coordination between the RAN and edge applications, requiring servers to notify the RAN upon receiving requests and transport protocol modifications to identify "first packets." This is impractical because RAN infrastructure and edge servers are typically managed by different entities (*e.g.,* Verizon operates the RAN while AWS provides edge compute). Even setting deployment issues aside, Tutti fails to timely accelerate LC requests due to feedback delays between edge servers and RAN. Moreover, Tutti only supports homogeneous applications with identical SLOs. ARMA [57] similarly requires coordination between edge servers and RAN schedulers, is narrowly tailored to video analytics, and allows non-LC applications to block LC ones when their uplink bandwidth usage is high. Both systems focus solely on RAN scheduling while ignoring edge compute resource management for co-existing heterogeneous applications.

SLO-aware resource management in clouds. A natural approach for managing compute resources is to adapt existing SLO-aware resource management solutions from cloud environments. However, these solutions are ill-suited for our target LC applications. PARTIES [30] reactively adjusts server resource partitions based on SLO feedback from clients, but fails in MEC where wireless feedback delays mean multiple requests miss deadlines before adjustments take effect. Caladan [32] employs proactive CPU core allocation without requiring client feedback but requires extensive application modifications. ML-based schedulers like OSML [38] incur inference overhead and operate at second-level intervals, making them too slow to react within the tens of milliseconds required for LC applications. Similarly, UFO [46] relies on changes in system-wide metrics (*e.g.,* CPU utilization, scheduling frequency) captured at second-level intervals to infer potential SLO violations and makes scheduling decisions, which reacts too slowly for LC applications. Vessel [37] assumes all processes share one memory space and relies on hardware-assisted core allocation, limiting its applicability in multi-tenant MEC settings and beyond CPU scheduling.

End-to-end rate control mechanisms. End-to-end congestion control mechanisms [29, 34, 47] are ill-suited for LC applications in MEC environments. Congestion control requires hundreds of milliseconds to converge, causing SLO violations before it can react to uplink congestion. Also, the 5G uplink channel quality fluctuates rapidly due to limited UE transmission power and varying user counts, making it nearly impossible for congestion control to stabilize. Even approaches that leverage 5G control-channel information (*e.g.,* PBE-CC [55]) to react quickly run solely at the end host and cannot directly influence RAN uplink scheduling. Under severe wireless contention, the RAN may allocate minimal or no uplink resources to a UE, so sender-side adaptation alone is insufficient to satisfy SLO requirements. Additionally, many LC applications rely on constant or variable bitrate encoding to maintain video quality [31, 50], where reactive rate adaptation directly degrades quality.

URLLC. 5G Ultra-Reliable Low-Latency Communication (URLLC), a service category designed for mission-critical applications, achieves low latency and high reliability through conservative radio configurations and dedicated resource reservations [19]. However, this approach reduces spectrum efficiency: reserved resources often remain underutilized, and reliability-enhancing techniques consume additional radio resources without commensurate benefit. The inefficiency is particularly pronounced for uplink-heavy workloads where resource demand is dynamic, making static reservations ill-suited for our setting.

3 Overview of SMEC

We now present SMEC, a practical SLO-aware resource management framework for MEC that support multiple heterogeneous applications through a decoupled scheduling approach.

3.1 Design Goals

To address the limitations identified in §2.4, we aim to design SMEC around four goals:

G1: No coordination between RAN and edge servers. SMEC should operate with completely decoupled schedulers at the RAN and edge servers. This will address the practical deployment and technical challenges that make coordination-based approaches infeasible.

G2: Compatibility with existing infrastructure and applications. SMEC should require no significant modifications to existing 5G stacks, edge servers, or applications. This will enable incremental deployment without disrupting existing MEC infrastructures or requiring application re-engineering.

G3: Resource management for heterogeneous applications. SMEC should provide resource allocation across RAN and edge resources for multiple competing applications with diverse SLOs. This will enable MEC deployments where heterogeneous applications must coexist efficiently.

G4: SLO satisfaction through deadline-aware resource scheduling. SMEC should prioritize SLO satisfaction over conventional fairness objectives that often lead to latency variability. Scheduling decisions must account for application deadlines and current resource availability, ensuring LC applications obtain the resources they need while still preventing starvation of BE applications.

3.2 Challenges

Realizing SLO-aware scheduling without RAN–edge coordination presents three challenges:

C1: New request identification at the RAN. RAN protocol layers, especially the MAC layer where resource allocation decisions are made, lack visibility into application payloads and find it impractical to parse application data due to the tight timing requirements of RAN processing. This makes it

challenging to identify when new requests begin and trigger appropriate SLO-aware scheduling decisions.

C2: Network latency estimation at the edge. The edge server needs to estimate the network transmission latency that each request has consumed during uplink transmission and will consume during downlink transmission for responses to compute the remaining time budget before the SLO deadline expires. However, it has no visibility into these transmission delays, making SLO-aware resource allocation challenging.

C3: Processing latency estimation for dynamic workloads. Even with network latency estimation, the edge scheduler must also anticipate how long requests will take to process. Request processing times vary with workloads and resource contention, and are hard to predict without intrusive application changes. This makes lightweight yet accurate estimation essential for practical deployment.

3.3 Key Ideas

Our core insight is that standard 5G protocols and MEC application behaviors already expose the necessary signals for decoupled SLO-aware scheduling. SMEC exploits these readily available signals with no or minimal application modifications. This approach enables three key ideas that address the fundamental challenges of decoupled scheduling:

I1: Exploiting 5G control signal patterns for request identification (§4.1). Standard 5G control signaling between UE and base station naturally exhibits distinctive patterns when new application requests are generated. We find that buffer status reports (BSRs) and scheduling requests (SRs) provide reliable signatures that correlate with when the client sends a new request. SMEC leverages these existing control signals to detect when new requests begin at the RAN without payload inspection or protocol modifications.

I2: Leveraging downlink stability for network latency estimation (§5.1). 5G downlink transmission characteristics provide inherent signals for network latency estimation. We observe that downlink transmissions exhibit more predictable latency than uplink transmissions due to more wireless slots allocated for downlink, stable base station transmission power, and absence of scheduling jitter. SMEC exploits this asymmetry in 5G protocol behavior through lightweight probing that exchanges small timing packets between edge servers and client devices, enabling accurate latency tracking without operator infrastructure coordination.

I3: Utilizing application lifecycle events for processing time prediction (§5.3). MEC applications' request-response behaviors expose key lifecycle events that enable processing time estimation. SMEC tracks these naturally occurring events through server-side APIs and builds execution history without requiring invasive application changes. We show this lightweight approach provides sufficient accuracy for deadline-aware scheduling while maintaining practicality.

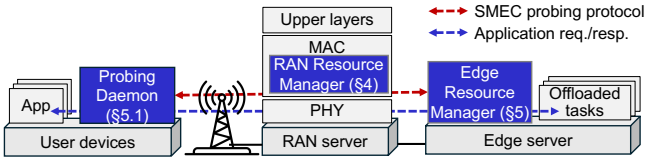


Figure 5: SMEC architecture: Blue boxes represent the main components of SMEC, while grey boxes denote existing elements in the MEC stack. For brevity, the 5G packet core and upper 5G protocol layers are omitted.

3.4 System Architecture

Figure 5 illustrates the SMEC architecture, which consists of two main components that operate independently:

RAN resource manager (§4) operates at the MAC layer and monitors UE–RAN control signal patterns to identify application request boundaries. It dynamically allocates RAN resources based on each request's remaining time budget, increasing resource allocations when requests approach their deadlines to accelerate transmission.

Edge resource manager (§5) estimates network transmission delays using a timing protocol between the UE-side daemon and server-side module. It also estimates processing delays for incoming requests based on recent execution history to calculate remaining time budgets. The manager then allocates heterogeneous compute resources (CPU and GPU) according to these time budgets and implements early drop mechanisms for requests unlikely to meet their deadlines.

Specifying SLOs to the RAN. LC applications communicate their SLO requirements to the RAN through standard 5G interfaces without requiring custom protocols. Edge servers can signal requirements through the Network Exposure Function (NEF) interface, or UE devices can convey QoS information when establishing Packet Data Unit (PDU) sessions [19]. SMEC leverages 5G QoS Identifier (5QI) classes to map application SLOs, aligning with how commercial network operators classify traffic rather than requiring fine-grained per-application specifications. This standards-compliant approach enables deployment within existing commercial MEC ecosystems while providing the necessary SLO information for effective resource management.

4 RAN Resource Management

This section presents the design of our SLO-aware RAN resource management, which operates at the RAN's MAC layer that makes a decision on RAN resource allocation.

4.1 Identifying Application Requests

As described in §3.2, RAN protocol layers lack visibility into packet payloads, preventing direct identification of application request boundaries. Even if payload access were available (*e.g.,* once the I/Q samples are demodulated into user data bits at the Packet Data Convergence Protocol (PDCP) layer), parsing application-layer data would violate the strict timing

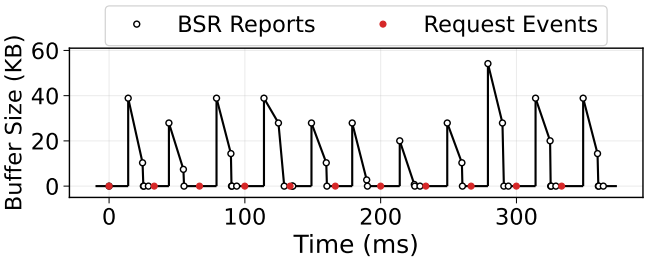


Figure 6: Correlation between bytes reported in BSR and application requests.

requirements of MAC scheduling, which must complete decisions within 500 µs or 1 ms to meet 5G timing requirement.

Our approach. Our key observation is that control signals from the UE to the RAN's MAC layer, particularly Buffer Status Reports (BSR), exhibit strong correlation with application-level requests. As shown in Figure 6, when a UE generates a new application request, additional data enters its uplink buffer, causing a noticeable increase in the reported BSR value. This provides a MAC-layer signal that reveals request boundaries without requiring payload inspection.

We leverage this correlation to detect application request boundaries through BSR increases. Specifically, the MAC layer identifies a new request boundary when the UE's reported BSR exhibits a step increase compared to the previous report. We define the request start time, t_{start}, as the timestamp when the MAC scheduler receives the first BSR report reflecting this increase.

This approach achieves per-request granularity when requests arrive with inter-generation times exceeding the BSR update interval. Under high sending rates or bursty workloads, multiple requests may be generated within a single BSR interval, appearing as one aggregated increase in the BSR. In such cases, the MAC layer treats these as a single *request group* sharing one t_{start}, and our scheduling decisions apply at the group level.

To handle multiple traffic types on the same UE (*e.g.,* traffic from different applications running concurrently), we leverage 5G logical channel groups (LCGs), which allow UEs to report separate BSRs per LCG to the MAC scheduler. By configuring uplink traffic into different LCGs based on SLO classes, the scheduler can track buffer status and infer t_{start} per traffic class rather than per UE.

Note that control signaling operates independently of user data transmission. The 5G protocol assigns higher priority to BSR transmission than to user data, ensuring that the MAC scheduler can observe buffer state changes even under heavy traffic conditions.

4.2 Deadline-aware RAN Scheduling

Once the MAC layer identifies the start time of a new request, it computes the remaining time budget as:

$$t_{budget}^{RAN} = \text{SLO} - (t_{current} - t_{start}) \quad (1)$$

With this information, the scheduler needs to decide how to allocate wireless resources across competing requests.

A 5G MAC layer allocates wireless resources[3] based on conventional metrics such as proportional fairness, which considers factors like channel quality and historical average throughput to balance efficiency and fairness among UEs. However, these approaches do not account for application SLO requirements.

Unlike prior designs [56, 57] that emphasize fairness between LC and BE requests, our scheduler explicitly prioritizes LC requests based on their remaining time budgets. This design choice stems from a key observation: the subsequent compute stage at the edge server introduces additional latency that the RAN cannot accurately observe or account for due to limited visibility and coordination across operators. Therefore, our scheduler aims to minimize deadline violations for LC requests while ensuring starvation freedom for BE requests when resources remain available.

To achieve this, we allocate resources to LC requests as quickly as possible, preserving sufficient time budget for the compute stage to meet end-to-end deadlines. Among competing LC requests, our scheduler prioritizes requests with the smallest remaining time budget. This ensures that requests approaching their deadlines receive higher priority, with already-violated requests receiving maximum priority to prevent buffer blocking.

Ensuring starvation freedom for BE requests. While prioritizing LC requests, our design ensures that BE requests remain starvation-free through two mechanisms. First, we assign higher priority to Scheduling Request (SR)-triggered resource allocations. In standard 5G MAC scheduling, when a UE has not received resources for an extended period, it sends an SR, a control signal that requests uplink grants from the MAC scheduler. Our scheduler assigns these SR-triggered allocations higher priority than regular scheduling decisions (even higher than LC requests), ensuring that BE UEs maintain forward progress. Since SR-triggered allocations are small (typically 1–2% of the wireless resources available for a slot), they do not impact LC request performance. Second, we implement dynamic priority reset: when an LC request completes transmission (detected when its BSR reaches zero), we immediately reset the UE's priority to zero, allowing BE UEs to fully utilize available resources and ensuring efficient bandwidth utilization.

5 Edge Resource Management

While the RAN resource manager prioritizes LC requests during uplink transmission, the edge server must allocate compute resources to meet SLO deadlines without any coordination with the RAN. The key challenge is estimating each request's remaining time budget by determining the network latency already consumed (uplink), the future network latency (downlink), and the expected processing time. This section

[3] Physical Resource Blocks in 5G terminology

API Call	Purpose
`request_sent(req_data*)`	Report new request sent
`request_arrived(req_data*)`	Report new request arrival
`processing_started(req_id)`	Report processing start
`processing_ended(req_id)`	Report processing completion
`response_sent(resp_data*)`	Report response transmission
`response_arrived(resp_data*)`	Report response arrival

Table 2: SMEC API for network and processing time estimation.

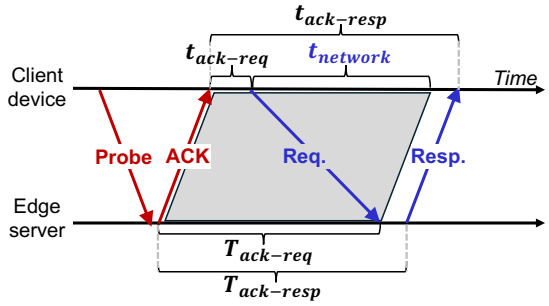


Figure 7: Probing-based network latency estimation. Red arrows indicate probing protocol packets while blue arrows indicate application requests and responses.

presents our approach that leverages timing signals naturally generated by applications during their request-response lifecycle, which can be captured via a lightweight API (Table 2).

5.1 Estimating Network Latency

Consider a request with a 100 ms SLO that has spent 60 ms in uplink transmission. The edge server has only 40 ms remaining to complete processing and downlink transmission. Without accurate timing information about both consumed uplink delays and future downlink delays, the edge server cannot distinguish between urgent requests approaching their deadlines and requests with ample time remaining.

However, measuring per-request network latency is challenging: the edge server only observes when requests arrive locally but lacks visibility into when they were originally sent from the client, the uplink delays they experienced, or the downlink delays that responses will experience.

Possible approach. A straightforward approach would piggyback a sending timestamp in each request, enabling the server to compute transmission delay upon reception. Unfortunately, this requires precise time synchronization between the UE and server, which is infeasible in MEC environments. Network Time Protocol (NTP) [41] introduces synchronization errors ranging from tens to hundreds of milliseconds, which are larger than the time budget of LC applications. PTP [14] is also unsuitable because it assumes network delays are symmetric. 5G networks exhibit inherent asymmetry where uplink latency is both higher and more variable than downlink latency due to protocol design and UE energy constraints.

Our approach. We leverage our observation that while uplink latency in 5G networks is highly variable, downlink latency remains consistently stable (Figure 2). We exploit the stability of downlink through a lightweight *probing-based network latency estimation* protocol that establishes timing references using the stable downlink path.

As illustrated in Figure 7, the client device periodically sends probe packets to the server, which responds with ACKs over the stable downlink. To implement this protocol, SMEC runs a per-UE *probing daemon* on the client side (shown in Figure 5) and integrates the corresponding functionality into the edge resource manager on the server side. When an application sends a request, it reports the request to the timing daemon via the `request_sent` API (Table 2), which measures $t_{ack-req}$ (the time elapsed since receiving the latest ACK) and inserts this timing metadata into the request payload. Upon receiving the request, the server computes $T_{ack-req}$ (the time difference between sending the most recent ACK and receiving the current request). The stable downlink timing creates a *parallelogram* relationship (grey region in Figure 7), allowing the server to estimate network latency as $T_{ack-req} - t_{ack-req}$.

However, if the response size is significantly larger than the ACK size, there will be a gap in downlink transmission latency between the two (Figure 2). To compensate for this difference, when the server sends a response, it computes the elapsed time since it sent the last ACK ($T_{ack-resp}$) and reports this value to the client daemon as part of the response. Upon receiving the response, the client daemon uses this information to compute the time elapsed since it received the last ACK ($t_{ack-resp}$) and calculates a compensation factor (t_{comp}), which it reports to the server as part of the next probe. The client daemon maintains this compensation factor separately for each application. Using this correction factor, the server estimates the network latency while accounting for response size differences:

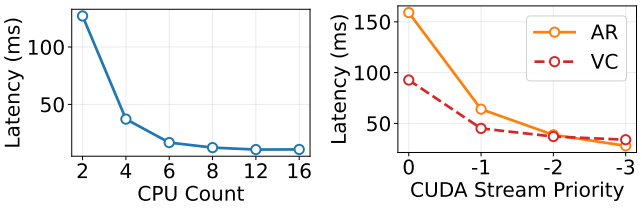
$$t_{network} = T_{ack-req} - t_{ack-req} + t_{comp} \quad (2)$$

To handle packet losses, each probe-ACK exchange carries a unique ID that both endpoints use to synchronize on the most recent successful exchange. The design incurs minimal overhead: the client sends small probe/ACK packets (<100 B) every few seconds, but *only while the UE is actively serving LC traffic*. When the UE is idle, the probing daemon pauses, avoiding interference with the UE's power-saving mechanism (*e.g.,* Discontinuous Reception (DRX)).

This approach extends to distributed scenarios where request initiators and response receivers are different devices (*e.g.,* smart stadium with camera initiators and audience UE receivers). In such cases, the initiator sends probe packets for network latency estimation while receivers sends probe packets to report compensation factors, enabling accurate end-to-end latency estimation across the distributed path.

5.2 Estimating Remaining Time Budget

Given the network latency estimates, the resource manager now needs to predict each request's *processing time* to com-



(a) CPU-based task **(b)** GPU-based task

Figure 8: Relationship between compute resources allocation and process latency.

pute an accurate remaining time budget. However, accurately predicting processing time while maintaining practical deployability presents challenges. Applications exhibit variable processing times due to workload change and resource contention, making accurate prediction difficult.

Possible approaches. One approach would be to infer processing time from system-wide signals like CPU utilization, scheduling frequency, and hardware performance counters (*e.g.,* cache misses, memory footprint), but such methods suffer from either high inference overhead or poor accuracy, as shown by existing work [38, 46]. Alternatively, one could extensively instrument applications to measure real-time workload information (*e.g.,* Caladan [32]); while this obtains precise timing information, it requires significant application modifications, limiting practicality.

Our approach. Building on the client-side request tracking for network latency estimation (§5.1), we extend this to the server side where applications report key lifecycle events (*e.g.,* request arrival, processing start/end) to the resource manager via the SMEC API (Table 2). By tracking these events, the resource manager estimates processing delays without requiring detailed application knowledge or extensive instrumentation.

The resource manager tracks two key metrics: waiting time (t_{wait}) defined as the time elapsed from request arrival until processing begins, and processing time using the median of last R requests as a robust predictor ($t_{process}$). While this median-based approach is simple and may introduce some prediction error, it performs well in practice (§7.6.2) while minimizing application modifications. The manager can then compute the remaining time budget as:

$$t_{budget}^{edge} = SLO - (t_{network} + t_{wait} + t_{process}) \quad (3)$$

5.3 Deadline-aware Proactive Edge Resource Scheduling

The resource manager determines urgency based on each request's remaining time budget. We define an urgency threshold as a fraction τ of the application's SLO (default $\tau = 0.1$), marking a request as urgent when $t_{budget}^{edge} < \tau \times SLO$. When a request becomes urgent, the manager *proactively* allocates additional resources to prevent SLO violations.

SMEC focuses on CPU and GPU management, which constitute the primary compute bottlenecks in edge servers. CPU cores are heavily contended among multiple applications, while GPUs accelerate ML inference and video processing. Algorithm 1 in Appendix B summarizes our deadline-aware scheduling approach.

CPU management. Our CPU manager partitions cores across applications using CPU affinity to ensure isolation and predictable performance. Based on the observation that increasing CPU core allocation reduces processing latency (Figure 8a), the manager allocates additional cores to urgent requests when they risk missing deadlines. This policy is most effective when an application can parallelize request processing (*e.g.,* via multi-threaded execution). To avoid thrashing from frequent core reallocations, the manager enforces a brief cool-down period (*e.g.,* 100 ms) after each allocation, and assigns another core only if requests still risk missing deadlines after the cool-down. This mechanism prevents wasteful oscillations while ensuring urgent requests receive the resources they need.

For *reclamation*, we use average CPU utilization rather than urgency signals. Urgency-based reclamation creates instability: removing a single core from a latency-critical application can cause an abrupt shift from meeting deadlines to missing many, leading to scheduler thrashing. Instead, when an application's utilization falls below a threshold (*e.g.,* 60%), the manager safely reclaims cores, providing stable resource management without oscillatory behavior.

GPU management. Commercial MEC offerings [22, 23] typically deploy inference-optimized GPUs such as NVIDIA L4 [7] and T4 [9]. These devices lack hardware-level partitioning capabilities like Multi-Instance GPU (MIG) [8]. Even when hardware partitioning is available, it only supports static allocation at application launch and cannot be dynamically adjusted during runtime.

To enable dynamic GPU scheduling, we leverage CUDA stream priorities, inspired by Orion [52]. NVIDIA's Multi-Process Service (MPS) [6] enables multiple applications to share the GPU while maintaining a unified priority hierarchy across their CUDA streams. When kernels from multiple applications contend for GPU resources, those launched from higher-priority streams receive preferential scheduling. This mechanism allows the resource manager to influence GPU scheduling decisions without requiring hardware partitioning or GPU driver modifications.

We exploit this capability by assigning stream priorities at request granularity based on urgency. As shown in Figure 8b, higher stream priorities reduce processing latency under contention. The resource manager assigns higher-priority streams to requests whose expected processing time closely matches their remaining time budget, while requests with slack use lower-priority tiers. This ensures urgent requests receive preferential GPU access without starving other workloads.

Early drop: handling overly urgent requests. When a request becomes overly urgent (*e.g.,* $t_{budget}^{edge} \leq 0$) due to excessive network or queuing delays, no amount of compute resources can recover the already-elapsed time. Processing such re-

quests wastes resources that could serve other requests still capable of meeting their deadlines. Therefore, when the edge server operates under load, the resource manager immediately drops overly urgent requests ("early drop"), redirecting resources to requests with viable time budgets.

6 Implementation

We have implemented a prototype of SMEC consisting of the RAN resource manager and edge resource manager, comprising approximately 7,700 lines of Python and C++ code [26].

RAN resource manager. We implement the RAN resource manager in the srsRAN 5G's MAC layer [12] with our request identification mechanism and deadline-aware scheduling algorithm. Since our design is not tightly coupled with srsRAN, it can be ported to other RAN stacks such as OAI 5G [1].

Edge resource manager. We implement the edge resource manager as a user-space daemon that coordinates with applications through the SMEC API. It consists of four core components: network latency estimation, processing time prediction, CPU/GPU management modules described below.

Network latency estimation. The network latency estimation module operates on both client and server to enable accurate per-request network latency measurement. The client periodically sends a probe message containing a 4-byte compensation factor tagged with a 4-byte probe ID to the server. The server replies with 12-byte ACK packets containing the same probe ID and the ACK's sending timestamp over the stable downlink path. In our prototype, we use a 1-second probing frequency to balance accuracy with overhead.

Processing time prediction. The processing estimation module maintains a sliding window of the past R requests' processing times for each application to predict future delays. By tracking both queueing and processing components through the SMEC API events, the system builds application-specific performance profiles. We use $R = 10$ as the window size in our prototype, providing sufficient history while remaining responsive to workload changes.

CPU management. Our CPU manager leverages Linux's `sched_setaffinity` system call [11] to dynamically bind application processes to specific CPU cores based on deadline urgency. This user-space approach enables fine-grained CPU allocation and reclamation without requiring kernel modifications or custom scheduling classes.

GPU management. Our GPU scheduler operates through NVIDIA's MPS, which enables CUDA stream priorities from different application processes to be compared on a unified scale. For example, a stream with priority -3 from one process correctly receives higher priority than a stream with priority 0 from another process. Each application creates multiple CUDA streams at initialization using the `cudaStreamCreateWithPriority()` API [44], with each stream assigned a distinct priority level. Based on deadline urgency feedback from the resource manager, incoming requests are dispatched to the appropriate stream, ensuring urgent requests execute on high-priority streams while less critical requests use lower-priority streams.

7 Evaluation

We evaluate SMEC on our private 5G MEC testbed that emulates commercial deployments with real applications.

7.1 Experimental Setup

Testbed setup. Our 5G MEC testbed consists of a UE emulator (Amari UE Simbox [3]), two x86 servers, and a USRP X310 [5] radio unit. **Server-1** acts as the RAN, running srsRAN [12] and Open5GS [10] with Intel Xeon Silver 4310 CPU, 128 GB memory, and Ubuntu 22.04. We configure the RAN in TDD mode with 80 MHz bandwidth and 2×2 MIMO on band 78, representing typical 5G deployments. **Server-2** serves as the edge server with NVIDIA L4 GPU [7], Intel Xeon Gold 6430 CPU (32 cores), 256 GB memory, and Ubuntu 24.04, reflecting commercial MEC offerings [22]. To emulate CPU contention, we disable hyper-threading and use 24 cores. Servers are connected via 25 GbE.

Applications. We implement three LC applications (Table 1) and one BE application. All LC applications are video-based, and we treat each video frame as a single request.

Smart stadium (SS) (SLO: 100 ms) [21,48]: 5G-enabled cameras upload high-resolution 4K video streams to the edge server, which transcodes each stream into multiple lower-bitrate versions and delivers them to subscribing clients. In our evaluation, we use the same UE to emulate both the camera and subscribing clients. This represents a CPU-intensive workload for live streaming services. We stream videos over RTP and implement transcoding using FFmpeg's H.264 codec. We use a video from the AdaPool dataset [51], re-encoded as a 4K 60 fps stream at 20 Mbps.

Augmented reality (AR) (SLO: 100 ms) [40,45]: AR devices stream videos over RTP to the edge server, which performs object detection using a YOLO model [49] and sends annotated results back to the AR devices. This represents a GPU-intensive workload for computer vision applications. We use a video from the MOT dataset [36], re-encoded as a 1080p 30 fps stream at 8 Mbps.

Video conferencing (VC) (SLO: 150 ms) [2, 25]: Client devices with limited connectivity send low-quality video streams to the edge server, which enhances them using super-resolution and streams the enhanced video back to the clients. This represents a GPU-intensive video enhancement workload. We stream videos over RTP and implement super-resolution using the Real-ESRGAN model [54]. We use a video from the ICME-VSR dataset [43], re-encoded as a 320p 30 fps stream at 800 Kbps.

File transfer (FT) (No SLO): Client devices transfer files with dummy content to a remote server (not the edge server) to simulate best-effort traffic.

Application workloads. To evaluate the impact of workload characteristics on SMEC, we use two types of workloads:

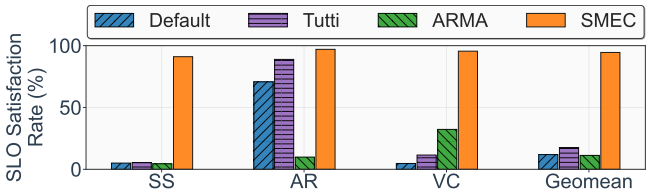


Figure 9: SLO satisfaction rate under static workload.

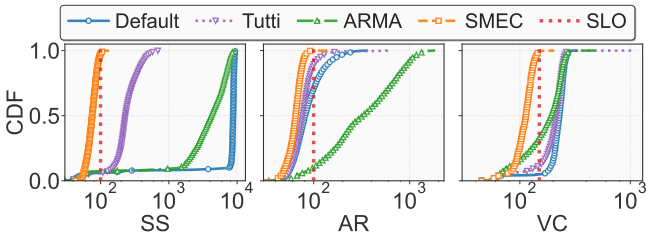


Figure 10: End-to-end latency (ms) under static workload.

Static: To evaluate how SMEC performs under sustained heavy load on both compute resources and the RAN, we design a static workload that creates continuous pressure on the system. We use 12 concurrent UEs: 2 for smart stadium, 2 for augmented reality, 2 for video conferencing, and 6 for file transfer. The LC UEs continuously send video frames at their respective target rates. The transcoding task of SS converts videos into three fixed resolutions (2K, 1080p, 720p), while AR performs object detection using the YOLOv8 medium model [53]. FT keeps sending 3 MB files repeatedly.

Dynamic: To evaluate how SMEC handles bursty requests and resource contention, we design a dynamic workload with fluctuating demand. We use 2 UEs for SS, but the transcoding task randomly varies the number of target resolutions (between 2 and 4), creating fluctuating compute demand. For AR and VC, we vary the number of UEs sending requests dynamically between 0 and 2. To amplify computational bursts, AR uses the larger YOLOv8 large model [53]. We use 6 UEs for FT, and each UE repeatedly uploads files with sizes uniformly chosen between 1 KB and 10 MB.

Baselines. We compare the performance of SMEC against three baselines: (1) Default scheduler (**Default**), (2) **Tutti**, and (3) **ARMA**. For the default scheduler, the RAN uses the PF scheduler, while the edge server employs the default Linux scheduler (EEVDF [24]) for CPU processes and the hardware scheduler in the L4 GPU for GPU tasks. For Tutti and ARMA, since neither considers edge resource scheduling, we pair them with the default scheduler at the edge server. To ensure a fair comparison, we implement early drop (§5.3) at the edge server for all baselines based on application queue length: we set the queue length to 10 and drop incoming requests when the queue exceeds this threshold.

7.2 Performance under Static Workloads

We first evaluate SMEC under static workloads that create sustained pressure on both network and compute resources.

SLO satisfaction. Across all three applications, SMEC exceeds 90% SLO satisfaction and outperforms the baselines (Figure 9). For SS, SMEC reaches 91% versus <6% for baselines; for VC, SMEC maintains 96% compared to only 5–35% for baselines. For AR, contention is modest under the static workload, so the headroom is smaller: SMEC improves SLO satisfaction by about 8% points over Tutti and about 26% points over Default.

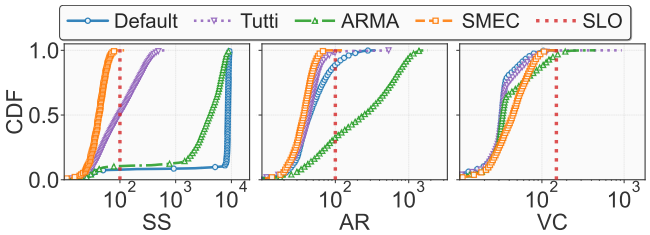


Figure 11: Network latency (ms) under static workload.

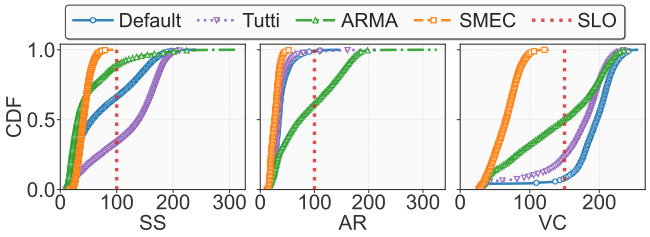


Figure 12: Processing latency (ms) under static workload.

Tail latency. SMEC substantially reduces tail latency (Figure 10): for smart stadium, P99 latency drops by 89×, 5.6×, and 84× relative to the Default, Tutti, and ARMA, respectively. For AR, SMEC reduces P99 latency by 2.9×, 2×, and 15.6× relative to the Default, Tutti, and ARMA, respectively. For VC, SMEC shows a smaller tail-latency gain (≈2×) because its uplink demand is low; VC is primarily impacted by compute contention rather than network latency.

Why baselines fall short. The latency breakdown pinpoints the root causes. ***Network*** (Figure 11). Default and ARMA rely on PF at the RAN, which allocates resources fairly across LC and BE UEs. This allows BE flows to occupy uplink resources that LC traffic urgently needs and starve LC apps. Uplink-heavy workloads suffer most: for SS, tail network latency approaches 10 s, leading to about 90 % of requests missing their SLO. Tutti also causes over 50 % SLO violations at network side for SS because it depends on server-side notifications to infer request start times; this delay prevents timely acceleration of urgent requests, leading to SLO violations. ***Edge server*** (Figure 12). All baselines ignore the effect of compute contention. Consequently, Tutti sees ∼70 % of SS requests misses SLO deadlines under CPU contention, and across Default, Tutti, and ARMA, video conferencing experiences ∼50–90 % SLO violations dominated by GPU contention. Notably, for smart stadium, Default and ARMA exhibit fewer processing-side SLO violations not because they handle compute contention, but because severe uplink congestion causes requests to backlog at the UE sending buffer. When this buffer is full, some requests are dropped, which in turn lowers CPU load at the server.

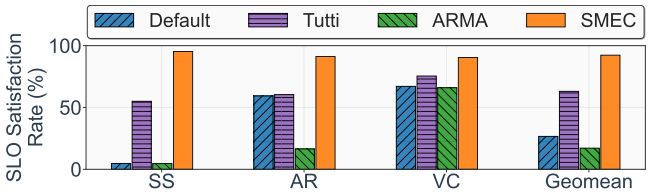


Figure 13: SLO satisfaction rate under dynamic workload.

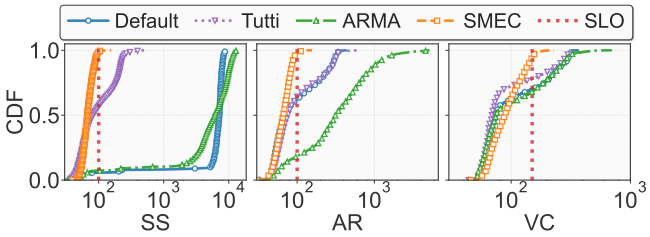


Figure 14: E2E latency (ms) under dynamic workload.

Why ARMA performs much poorer for AR. Under resource pressure, ARMA's RAN scheduler and reallocates uplink resources away from AR to prioritize SS; as a result, AR receives fewer grants and its uplink waiting time increases. This raises AR's *network* latency significantly (AR in Figure 11). A second-order effect further hurts *compute*: as AR grants resume, many backlogged AR requests arrive at the server nearly simultaneously, creating a burst that inflates queueing and causes deadline misses (AR in Figure 12).

7.3 Performance under Dynamic Workloads

We evaluate SMEC under dynamic workloads that create bursty traffic patterns and fluctuating compute demands, emulating real-world edge environments with varying workloads.

SLO satisfaction. SMEC maintains over 90 % SLO satisfaction across all three LC applications under dynamic workloads (Figure 13), demonstrating robust performance even with fluctuating demands. For SS, SMEC achieves 95.3 % satisfaction compared to 4.7 % (Default), 4.6 % (ARMA) and 55 % (Tutti), while for AR, SMEC sustains 91.2 % *vs.* only 59.3 % (Default), 16.4 % (ARMA) and 60.4 % (Tutti). For VC, the gap is smaller (90.4 % *vs.* 65–75 % for baselines) because VC has low uplink demand and is affected mainly by GPU contention rather than network latency. Under dynamic workloads, VC only misses deadlines during GPU bursts (*i.e.,* when all AR and VC clients issue requests concurrently) so SLO violations are concentrated in those burst windows, keeping the overall gap modest.

Tail latency. SMEC keeps tail latency within reasonable bounds, with delays rarely exceeding SLO targets by large margins (Figure 14). For SS, SMEC reduces P99 latency by 87× compared to the default scheduler, while achieving 3.2× and 122× improvements over Tutti and ARMA, respectively. For AR, SMEC reduces P99 latency by 3.2× *vs.* Default, 3.3× *vs.* Tutti, and 31× *vs.* ARMA. For VC, P99 improves by ∼2×; as in the static case, low uplink demand leaves VC compute-bound, so gains are smaller.

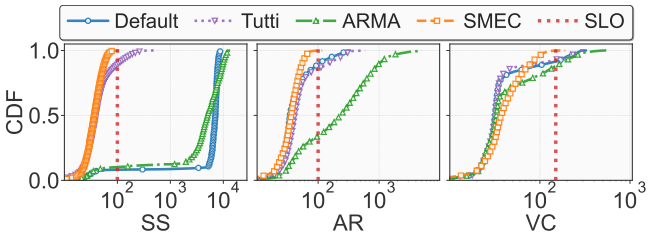


Figure 15: Network latency (ms) under dynamic workload.

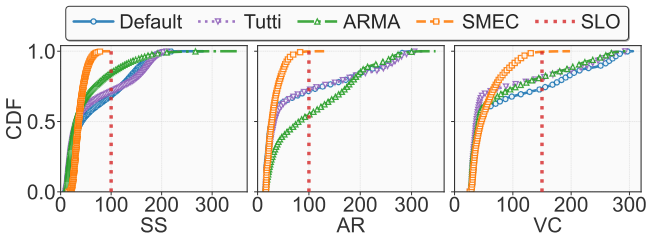


Figure 16: Processing latency (ms) under dynamic workload.

Why baselines fall short. To make the causes explicit, we similarly decompose end-to-end latency. ***Network*** (Figure 15). The reasons mirror the static case: Default and ARMA use PF at the RAN, allowing BE flows to take uplink resources urgently needed by LC traffic, starving LC apps; Tutti depends on delayed server-side start inference, so it cannot accelerate urgent requests in time. ***Edge server*** (Figure 16). The key difference in the dynamic setting is burstiness. Each burst overloads the edge server, creating long queues and around 30 % deadline misses for all baselines due to compute contention, which ignore compute contention. In contrast, SMEC proactively controls backlog (*e.g.,* by dropping a small fraction of hopeless requests) to relieve pressure and keep queues short, thereby preventing burst-induced misses.

7.4 Impact on Best Effort Applications

To verify that SMEC does not starve BE traffic, we measure the average throughput of all BE UEs under both static and dynamic workloads (Figure 17). In both cases, BE UEs fairly share the remaining bandwidth, around 1.8 Mbps under static workload and 3 Mbps under dynamic workload, and no UE experiences prolonged starvation throughout the experiments.

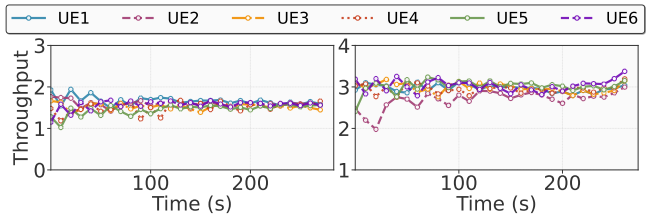
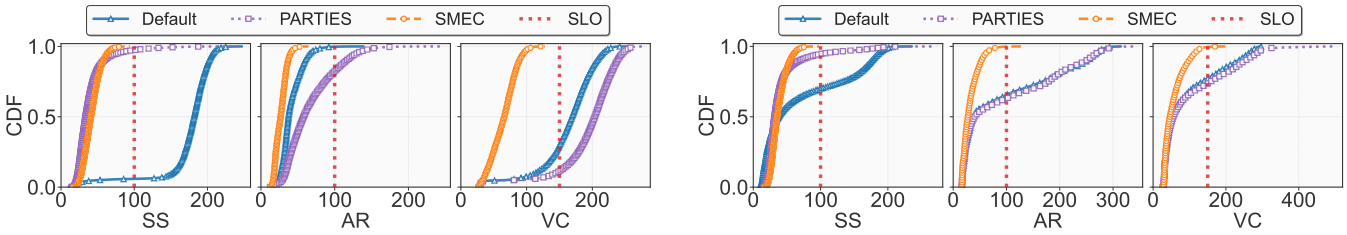


Figure 17: Throughput (Mbps) for each file transfer application while running static (left) and dynamic (right) LC workloads.



(a) Static workload **(b)** Dynamic workload

Figure 18: Processing latency (ms) at edge server with different resource schedulers.

7.5 Impact of Edge Resource Schedulers

We evaluate the impact of different edge resource schedulers by comparing Default, PARTIES [30], and SMEC's edge scheduler under static and dynamic workloads. All experiments use SMEC's RAN scheduler to isolate edge effects, with processing latency as the primary metric (Figure 18).

Static workload. SMEC reduces SLO violations across all applications and lowers P99 processing latency by 1.5–4× *vs.* Default and PARTIES (Figure 18a). PARTIES achieves reasonable performance for SS due to static workload but suffers from delayed feedback effects that inflate tail latency. For AR and VC, PARTIES underperforms Default by sometimes prioritizing both LC applications simultaneously, amplifying GPU interference.

Dynamic workload. SMEC maintains superior SLO satisfaction and 2–4× lower P99 latency than baselines (Figure 18b). PARTIES shows 10% more SLO violations compared to static workload due to increased variability. Default and PARTIES perform poorly for AR and VC because they lack deadline awareness and accurate early drop, causing queue buildup and widespread SLO violations during bursts.

7.6 Microbenchmarks

To complement our end-to-end evaluation, we conduct a set of microbenchmarks to evaluate the effectiveness of key components of SMEC.

7.6.1 Accuracy of Request Start Time Estimation

We evaluate the RAN scheduler's accuracy in estimating request start times using P99 absolute error (Figure 19). SMEC achieves much lower error than Tutti and ARMA. They infer request start times only after the edge server observes part of a request and notifies their RAN scheduler. When uplink resources are not promptly allocated to LC applications, the server observes requests much later, and the notification to the RAN scheduler is delayed, causing large start-time estimation errors. For example, for SS, ARMA shows 10 s of P99 error in both static and dynamic workloads, while Tutti incurs hundreds of milliseconds of P99 error. By contrast, SMEC relies on 5G control messages without any coordination with the edge server, making its estimation accurate and independent of uplink delays with only 10 ms of P99 error.

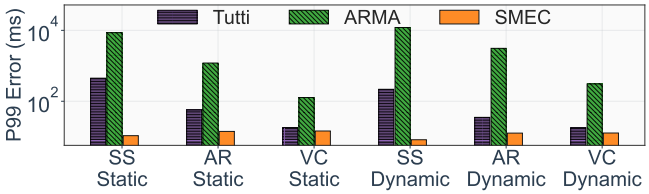
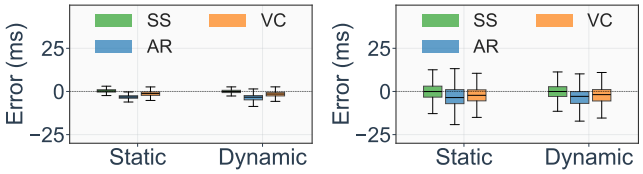


Figure 19: Accuracy of request start time estimation.



(a) Network latency estimation **(b)** Processing time estimation

Figure 20: Accuracy of network and processing latency estimation.

7.6.2 Accuracy of Latency Estimation

We evaluate the accuracy of network and processing time estimation, which directly determines scheduling precision (Figure 20). ***Network latency prediction*** is highly accurate, with errors typically within 5 ms across all LC applications (Figure 20a). The error mainly stems from downlink transmission latency differences between ACK packets and actual responses. Our compensation mechanism (§5.1) mitigates part of this difference, but a small residual error remains. ***Processing time estimation*** is also sufficiently accurate, with most requests showing errors within 10 ms (Figure 20b). Larger errors stem from inherent per-request variance (*e.g.,* key frames in SS, complex scenes in AR) and amplified variance under compute contention. Despite these errors, the prediction accuracy ensures that the majority of requests complete within their SLOs.

7.6.3 Effect of Early Drop

Lastly, we evaluate the effectivenss of the early drop mechanism by measuring SLO satisfaction under both static and dynamic workloads (Figure 21). Early drop consistently im-

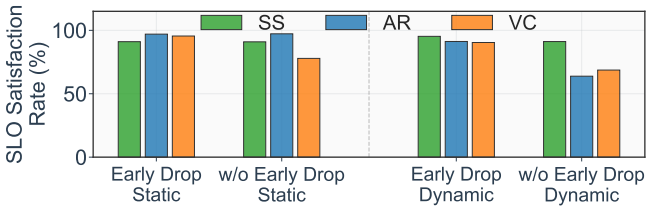


Figure 21: SLO satisfaction rate with and without early drops.

proves performance across both workloads. Under the static workload, where bursts are rare and mainly come from VC's sporadic traffic, early drop provides around 18% points improvement primarily for VC. Under the dynamic workload, where overloads are frequent, especially for GPU-heavy AR and VC, early drop brings over 20% points improvement by discarding overly-urgent requests, freeing resources for requests likely to meet their deadlines.

8 Discussion

Admission control for poor wireless channel conditions. While SMEC's RAN resource management is independent of wireless channel status, poor channel conditions can still degrade performance. LC applications with high bandwidth demands but weak channel quality may consume nearly all wireless resources while missing their SLOs, degrading performance for other users. An admission control mechanism can address this by profiling application throughput requirements against UE channel status and terminating service when channel quality is insufficient. This preserves SLO satisfaction for UEs with acceptable channel conditions while maintaining efficient spectrum utilization. Recent work such as Zipper [28] provides techniques for designing such mechanisms.

Handling downlink contention. Although this work focuses on uplink RAN scheduling, downlink contention is also important to consider. Downlink transmissions may suffer from congestion, but the downlink channel is usually more stable and predictable than uplink. This stability enables end-to-end rate control mechanisms to converge to appropriate sending rates when timely client feedback is available. Prior work [42] leverages uplink ACKs for faster congestion feedback, but delayed uplink grants in 5G networks can hinder timeliness. By improving uplink scheduling, SMEC can prevent excessive queueing at UE buffers that would delay feedback packets (*e.g.,* TCP ACKs), ensuring timely congestion signals and enabling more effective downlink rate adaptation.

Dealing with UE handover. While the applications studied in this work do not involve UE mobility across cells, scenarios such as autonomous driving may trigger inter-cell handovers. In deployments where multiple cells are managed under the same base station, handovers can often be absorbed locally without session disruption. The more challenging case involves mobility across different base stations, which requires transferring both radio and edge-session state across sites. To support LC applications under such conditions, we envision proactively replicating UE state across base stations to enable seamless session continuation after handover. Designing such mechanisms introduces open questions around synchronization overhead and consistency.

Fairness vs. latency trade-off. SMEC prioritizes LC requests to meet their SLOs, consistent with our design goals (§3.1). Stronger throughput fairness between LC and best-effort (BE) traffic would delay LC transmissions under contention, increasing tail latency and reducing SLO satisfaction. Our evaluation reflects this tension: systems that prioritize fairness between LC and BE traffic (*e.g.,* Tutti and ARMA) incur more LC SLO violations than SMEC, as shown in Figure 11 and Figure 15. Accordingly, for BE workloads SMEC provides starvation freedom rather than strong throughput fairness under heavy contention. Achieving strong fairness guarantees while preserving tight latency bounds for LC applications under wireless contention remains an open problem.

Limitations. Our request identification approach at the RAN is constrained by the aggregation semantics of BSR. When requests arrive in rapid succession within a single BSR reporting interval, they appear as a single aggregated increase at the MAC layer. In such cases, SMEC can only infer a shared request start time and performs scheduling at the request-group level rather than achieving per-request granularity. This limitation affects scenarios with extremely high request rates but does not impact typical workloads where inter-request intervals exceed BSR reporting periods.

A second limitation concerns processing time estimation for compute resource scheduling. SMEC relies on processing time estimates to proactively allocate compute resources, which works well for applications with stable processing pipelines like those in our evaluation. However, applications with dynamic processing patterns (*e.g.,* adaptive VR rendering) can exhibit high variance in processing latency, making accurate estimation challenging. In such cases, estimation errors may reduce the effectiveness of SMEC's compute-side scheduling, though RAN-side scheduling remains unaffected and continues to minimize network delays.

9 Conclusions

We introduce SMEC, a practical SLO-aware resource management framework for 5G MEC that achieves predictable performance without sacrificing deployment practicality. By leveraging standard 5G control signals and natural application behaviors through lightweight APIs, SMEC enables fully decoupled, deadline-aware scheduling at the RAN and edge with minimal application modifications. Our 5G MEC testbed evaluation demonstrates that SMEC achieves 90–96% SLO satisfaction versus under 6% for existing approaches, reduces tail latency by up to 122×, and ensures starvation freedom for best-effort applications. Thus, SMEC provides a practical foundation for 5G MEC deployments that reliably meet the stringent demands of latency-critical applications.

Acknowledgements

We would like to thank the anonymous NSDI reviewers and our shepherd, Timothy Wood, for their insightful comments and constructive feedback. We also thank Xin Yuan, Li Shen, and Peirui Cao for their assistance with the measurements in this paper. This work is based upon work supported by U.S. National Science Foundation (NSF) Awards 2403026 and 2326576. Xiao Zhang was also supported by the Amazon AI PhD Fellowship.

References

[1] 5G RAN – OpenAirInterface. https://openairinterface.org/oai-5g-ran-project/. Accessed 2025-07-15.

[2] Accessing meeting and phone statistics (Zoom latency recommendation). https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0070504. Accessed 2025-07-15.

[3] Amarisoft Amari UE Simbox E-Series. https://www.amarisoft.com/test-and-measurement/network-testing/network-products/amari-ue-simbox-e-series. Accessed: 2025-09-13.

[4] AWS Wavelength. https://aws.amazon.com/wavelength/. Accessed 2025-07-15.

[5] Ettus USRP X310. https://www.ettus.com/all-products/x310-kit/. Accessed: 2025-09-13.

[6] Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed 2025-07-15.

[7] NVIDIA L4 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/l4/. Accessed 2025-07-15.

[8] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Accessed 2025-07-15.

[9] NVIDIA T4 Tensor Core GPU for AI Inference. https://www.nvidia.com/en-us/data-center/tesla-t4/. Accessed 2025-07-15.

[10] open5GC. https://open5gs.org/. Accessed 2025-07-15.

[11] sched_setaffinity(2) — Linux manual page. https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html. Accessed 2025-07-15.

[12] srsRAN. https://www.srsran.com/. Accessed 2025-07-15.

[13] 5G network as foundation for autonomous driving. https://www.telekom.com/en/company/details/5g-network-as-foundation-for-autonomous-driving-561986, 2020.

[14] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2020.

[15] Mobile cloud gaming – an evolving business opportunity. https://www.ericsson.com/en/reports-and-papers/mobility-report/articles/mobile-cloud-gaming, 2020.

[16] Xbox Cloud Gaming. https://www.xbox.com/en-US/cloud-gaming, 2022.

[17] NVIDIA CloudXR SDK. https://developer.nvidia.com/cloudxr-sdk, 2023.

[18] T-Mobile US taps up Google for edge compute. https://www.telecoms.com/public-cloud/t-mobile-us-taps-up-google-for-edge-compute, 2023.

[19] ETSI TS 123 501 V18.5.0; 5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 18.5.0 Release 18). https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/17.05.00_60/ts_123501v170500p.pdf, 2024.

[20] Multi-access Edge Computing (MEC);Framework and Reference Architecture. Technical report, ETSI GS MEC 003 V3.2.1, 2024.

[21] Smart Stadium – The Stadium's 5G Revolution). https://5glab.orange.com/en/realisations/smart-stadium-the-stadiums-5g-revolution/, 2024.

[22] AWS Outposts racks pricing—North America / Central America). https://aws.amazon.com/outposts/rack/pricing/north-central-america/, 2025.

[23] Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge, 2025.

[24] EEVDF Scheduler - The Linux Kernel. https://docs.kernel.org/scheduler/sched-eevdf.html, 2025. accessed: 2025-07-25.

[25] Latency (audio). https://en.wikipedia.org/wiki/Latency_(audio), 2025.

[26] SMEC Github Repository. https://github.com/smec-project, 2025.

[27] stress-ng Github repository. https://github.com/ColinIanKing/stress-ng, 2025. accessed: 2025-09-13.

[28] Arjun Balasingam, Manikanta Kotaru, and Paramvir Bahl. Application-Level Service Assurance with 5G RAN Slicing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 841–857, 2024.

[29] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 25(5):2629–2642, 2017.

[30] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[31] Fortinet. Understanding IP Surveillance Camera Bandwidth. White paper, Fortinet, Inc., 2020.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[33] Ahmad Jalali, Roberto Padovani, and Rajesh Pankaj. Data throughput of CDMA-HDR a high efficiency-high data rate personal communication wireless system. In *VTC2000-Spring. 2000 IEEE 51st Vehicular Technology Conference Proceedings (Cat. No. 00CH37026)*, volume 3, pages 1854–1858. IEEE, 2000.

[34] Ingemar Johansson and Zaheduzzaman Sarker. Self-clocked rate adaptation for multimedia. Technical report, 2017.

[35] Frank Kelly. Charging and rate control for elastic traffic. *European transactions on Telecommunications*, 8(1):33–37, 1997.

[36] Laura Leal-Taixé, Anton Milan, Ian Reid, Stefan Roth, and Konrad Schindler. Motchallenge 2015: Towards a benchmark for multi-target tracking. *arXiv preprint arXiv:1504.01942*, 2015.

[37] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 280–295, 2024.

[38] Lei Liu, Xinglei Dou, and Yuetao Chen. Intelligent resource scheduling for co-located latency-critical services: A multi-model collaborative learning approach. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 153–166, 2023.

[39] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th annual international conference on mobile computing and networking*, pages 1–16, 2019.

[40] Huizi Mao, Xiaodong Yang, and William J Dally. A delay metric for video object detection: What average precision fails to tell. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 573–582, 2019.

[41] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[42] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 193–206, 2022.

[43] Babak Naderi, Ross Cutler, Juhee Cho, Nabakumar Khongbantabam, and Dejan Ivkovic. ICME 2025 Grand Challenge on Video Super-Resolution for Video Conferencing. *arXiv preprint arXiv:2506.12269*, 2025.

[44] NVIDIA Corporation. CUDA C Programming Guide: Stream Priorities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2025. Section 6.2.8.5.7 "Stream Priorities". Accessed 2025-09-05.

[45] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for Memory-Efficient, Real-Time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, 2023.

[46] Yajuan Peng, Shuang Chen, Yi Zhao, and Zhibin Yu. UFO: The Ultimate QoS-Aware Core Management for Virtualized and Oversubscribed Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1511–1530, 2024.

[47] M Ramalho and S Mena. Network-assisted dynamic adaptation (NADA): a unified congestion control scheme for real-time media. 2020.

[48] Red5 Team. How to Perfect 5G In-Venue Experiences with Real-Time Precision. https://www.red5.net/blog/perfect-5g-in-venue-experiences/, 2023.

[49] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016.

[50] Sony Corporation. Sony BRC-X400 PTZ Network Camera — Product Specifications, 2024. Bitrate control: CBR/VBR (selectable); bitrate range 512 kbps–50 Mbps. Accessed 2025-09-12.

[51] Alexandros Stergiou and Ronald Poppe. AdaPool: Exponential Adaptive Pooling for Information-Retaining Downsampling. 2021.

[52] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.

[53] ultralytics. yolov8: real-time object detection. https://yolov8.com/#what-is, 2025. accessed: 2025-09-13.

[54] Xintao Wang, Liangbin Xie, Chao Dong, and Ying Shan. Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data. In *International Conference on Computer Vision Workshops (ICCVW)*.

[55] Yaxiong Xie, Fan Yi, and Kyle Jamieson. Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 451–464, 2020.

[56] Dongzhu Xu, Anfu Zhou, Guixian Wang, Huanhuan Zhang, Xiangyu Li, Jialiang Pei, and Huadong Ma. Tutti: coupling 5G RAN and mobile edge computing for latency-critical video analytics. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 729–742, 2022.

[57] Juheon Yi, Goodsol Lee, Seokgyeong Shin, Minkyung Jeong, Daehyeok Kim, and Youngki Lee. Towards End-to-End Latency Guarantee in MEC Live Video Analytics with App-RAN Mutual Awareness. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pages 1–14, 2025.

A Additional Measurement Results from Commercial MEC Deployments

This appendix presents additional measurement results from our commercial MEC deployment studies that complement the findings presented in §2.

A.1 End-to-End Latency of Augmented Reality

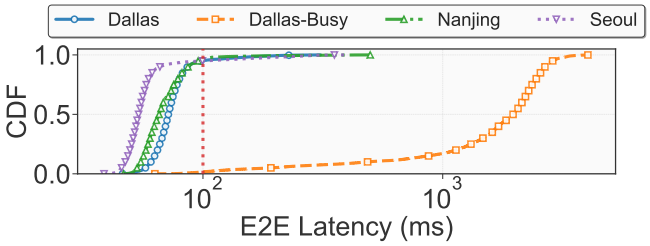


Figure 22: End-to-end latency for the augmented reality application without edge resource contention across MEC deployments in three cities. The dotted red line indicates the SLO.

Figure 22 shows that augmented reality (AR) exhibits a similar long-tail latency distribution as smart stadium during periods of low network activity. Since AR requires lower uplink throughput than smart stadium, only about 5 % of requests miss their SLO across all three cities. However, under high network activity (Dallas-Busy), wireless resource contention causes over 98 % of requests to violate their SLO.

A.2 Effect of Compute Contention on End-to-End Latency

We show the remaining results under compute resource contention for smart stadium in Nanjing and Seoul and augmented reality in all three cities.

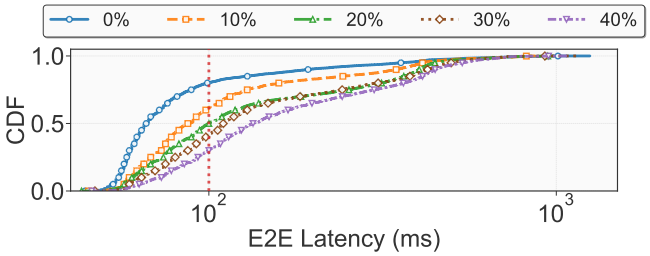


Figure 23: End-to-end latency for smart stadium under different levels of compute resource contention in Nanjing. The dotted red line indicates the SLO.

End-to-end latency of smart stadium. Figure 23 and Figure 24 further confirm the effect of CPU resource contention on the end-to-end latency of CPU-intensive applications like smart stadium. As CPU load increases, more requests exceed their SLO requirements and suffer from long tail latency.

End-to-end latency of augmented reality. To further validate the impact of GPU contention on the end-to-end latency of GPU-intensive applications such as augmented reality, we run experiments in Dallas, Nanjing, and Seoul under varying levels of GPU load. We implement a GPU stressor using CUDA to emulate different contention levels. As shown in Figure 25, Figure 26, and Figure 27, higher stress levels lead to more requests exceeding their SLOs and experiencing longer tail latencies.

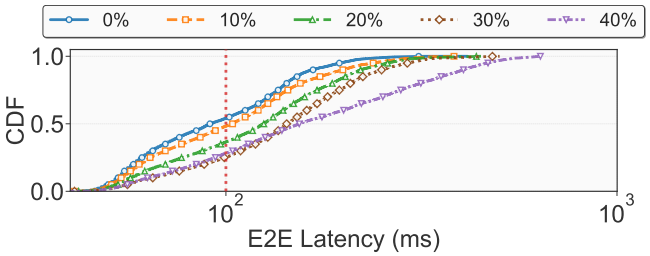


Figure 24: End-to-end latency for smart stadium under different levels of compute resource contention in Seoul. The dotted red line indicates the SLO.

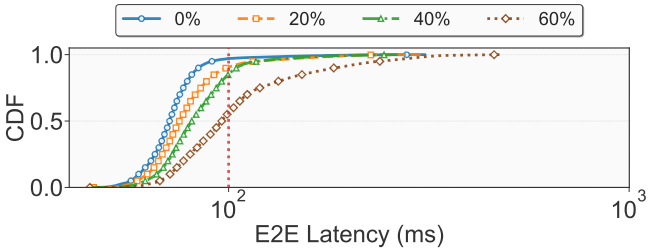


Figure 25: End-to-end latency for augmented reality under different levels of compute resource contention in Dallas.

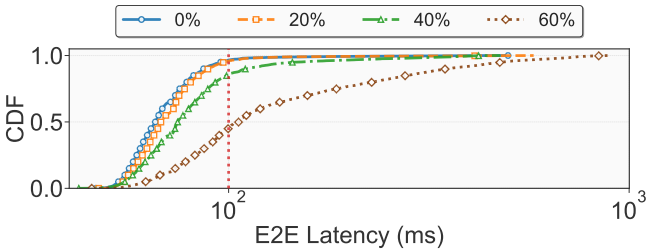


Figure 26: End-to-end latency for augmented reality under different levels of compute resource contention in Nanjing.

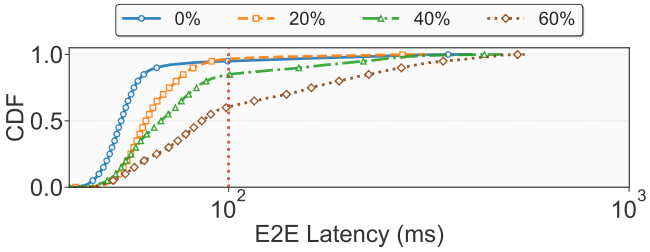
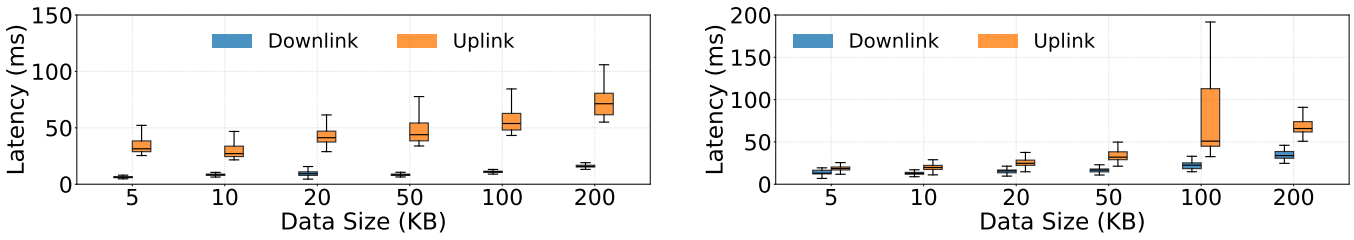


Figure 27: End-to-end latency for augmented reality under different levels of compute resource contention in Seoul.



(a) Nanjing **(b)** Seoul

Figure 28: Network latency variability for uplink and downlink transmissions across different data sizes in Nanjing and Seoul.

A.3 Variability of Network Latency

Figure 28 reveals consistent asymmetry in both Nanjing and Seoul. As request size grows, uplink latency shows pronounced variability, whereas downlink latency remains largely stable. This resembles our observation that uplink paths suffer from high variability, while downlink paths maintain relative stability.

B Deadline-aware Proactive Edge Resource Scheduling Algorithm

Algorithm 1: Deadline-aware proactive edge resource scheduling.

Notation: r denotes a request; a denotes the application that serves r; $\texttt{type}(r) \in \{\text{CPU}, \text{GPU}\}$.

if $t^{edge}_{budget}(r) \leq 0$ **then**
    drop(r)
    return

$urgency(r) \leftarrow t^{edge}_{budget}(r)/SLO_a$

if $type(r) = \text{CPU}$ **then**
    if $urgency(r) < 0.1$ **then**
        if $now - t_{last\_cpu\_alloc}(a) \geq 100\,\text{ms}$ **then**
            assign_one_more_core(a)
            $t_{last\_cpu\_alloc}(a) \leftarrow \text{now}$
    if $cpu\_util(a) < 60\%$ **then**
        reclaim_one_core(a)
else
    $prio(r) \leftarrow \texttt{map\_urgency\_to\_prio}(urgency(r))$
    set_cuda_stream_priority(r, prio(r))

process_request(r)