Characterization of Mobile Augmented Reality Tasks Supported by Edge Computing Resources

Karlla Chaves Rodrigues[1], **Kleber Vieira Cardoso**[1], **Sand Luz Correa**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

{karlla, kleber, sand}@inf.ufg.br

Abstract. *Mobile Augmented Reality (MAR) applications rely on computationally intensive and latency-sensitive tasks such as Simultaneous Localization and Mapping (SLAM) and object detection, which challenge mobile devices. Offloading these workloads to edge servers is promising, yet realistic workload and traffic models remain scarce, especially under heterogeneous CPU–GPU infrastructures. This paper presents an empirical characterization of computational demand, memory usage, and IP traffic generated by SLAM and YOLO-based object detection using an enhanced MR-Leo prototype (eMR-Leo). Experiments are conducted under controlled conditions with USB tethering to isolate application-level traffic. Results show that SLAM is CPU-bound with limited gains from GPU acceleration, while object detection is highly parallel and benefits significantly from GPU offloading. Object detection latency is reduced from 148 ms to approximately 4 ms per frame (approximately 98% reduction), enabling real-time performance. Based on these measurements, we derive statistical workload models for both tasks, covering instruction counts, memory usage, and traffic patterns, supporting realistic simulation and performance evaluation of edge-assisted MAR systems.*

1. Introduction

Recent advances in mobile hardware, software frameworks, and wireless communication infrastructures have enabled increasingly immersive and context-aware Mobile Augmented Reality (MAR) applications, fostering their adoption in latency-sensitive scenarios. Despite this progress, mobile devices remain constrained in their ability to sustain the computational demands imposed by MAR workloads [Wang et al. 2025], especially for prolonged periods. One way to overcome this problem is to offload MAR computationally intensive tasks to edge servers [Siriwardhana et al. 2021]. Edge-assisted MAR applications leverage additional computational capacity closer to end users, reducing end-to-end latency, lowering energy consumption on the user equipment (UE), and mitigating thermal constraints. Furthermore, edge servers are increasingly equipped with specialized accelerators, such as GPUs, which can significantly enhance the performance of parallelizable tasks and improve the overall quality of experience.

A growing body of work has demonstrated the effectiveness of edge-assisted offloading for MAR applications [Chen et al. 2017, Pereira et al. 2021, Zhang et al. 2022, Hammad et al. 2023, Espindola et al. 2025]. However, most existing studies focus on system design and experimental evaluation, while comparatively few [Toczé et al. 2020, Morín et al. 2024] address the problem from a workload characterization and modeling

perspective. Nevertheless, such characterization is essential to enable systematic evaluation of resource allocation policies and guide the design of offloading strategies under diverse deployment conditions. Another limitation of prior work is the common assumption of homogeneous, CPU-only edge infrastructures. In practice, however, many MAR tasks benefit significantly from GPU acceleration, and failing to account for this resource can lead to inefficient performance of the applications.

In this work, we provide a detailed characterization of MAR workloads and their associated IP traffic under heterogeneous edge platforms. Focusing on two fundamental (usually independent) MAR tasks, namely, Simultaneous Localization and Mapping (SLAM) and object detection, we extend the MR-Leo prototype [Toczé et al. 2020] to support GPU-accelerated execution of ORB-SLAM2 [Mur-Artal and Tardós 2017] and integrate the application into an object detection pipeline based on YOLO [Redmon et al. 2016]. Using this enhanced prototype, called enhanced MR-Leo (eMR-Leo), we conduct an experimental evaluation under different hardware configurations, analyzing processing time, number of instructions executed per second, memory consumption, and the generated IP traffic. Based on the collected measurements, we derive a statistical workload that captures the resource demands and network traffic generated by each task. The contributions of this work are threefold. First, we enhance an open-source edge-assisted MAR prototype with an object detection pipeline. Second, we provide realistic workload and traffic traces for two distinct state-of-the-art MAR tasks running on heterogeneous edge infrastructures. Third, we derive and release a statistical workload and IP traffic model to support reproducible research and facilitate future studies on edge-assisted MAR systems.

The remainder of this paper is organized as follows. Section 2 reviews background and related work. Section 3 describes eMR-Leo. Section 4 presents the experimental evaluation and data collection methodology. Section 5 discusses the workload and traffic modeling. Finally, Section 6 concludes the paper and outlines future research directions.

2. Background and Related Work

The reality–virtuality continuum spans a range of experiences from the physical world to fully Virtual Reality (VR) [Milgram and Kishino 1994]. Within this continuum, Augmented Reality (AR) overlays virtual content onto the physical environment in real time, while MAR delivers such experiences through mobile devices, including smartphones and head-mounted displays (HMDs). To operate effectively, MAR systems rely on computationally intensive tasks, notably SLAM, object detection, and frame rendering [Morín et al. 2022]. The SLAM task estimates the device's pose in real-time, reconstructs the physical environment in a 3D model, and uses the pose and the 3D reconstruction to anchor the virtual content to the real scenario, ensuring that device movements do not affect the pose of the virtual object [Cadena et al. 2016]. Object detection extracts high-level semantic information from visual data by identifying objects present in a scene and estimating their spatial extent, typically represented by bounding boxes and associated confidence scores [Zou et al. 2023]. Finally, rendering blends virtual and physical elements in a visually consistent manner.

The computational intensity and latency sensitivity of these tasks have motivated extensive research on offloading MAR processing to edge infrastructures. Prior work has explored different architectural solutions for edge-assisted MAR, primarily focusing on

reducing end-to-end latency perceived at the mobile device. [Chatzopoulos et al. 2017] discusses alternative offloading strategies, while [Siriwardhana et al. 2021] relates MAR requirements to 5G architectures. In [Chen et al. 2017], the benefits of edge computing for a MAR cognitive-assistance prototype are explored under varying edge server configurations. In [Pereira et al. 2021], a distributed architecture for browser-based AR applications, featuring a geospatial directory service and messaging bus, called ARENA, is presented. The work carried out in [Zhang et al. 2022] proposes EdgeXAR, an edge-assisted MAR framework featuring a hybrid tracking system to compensate for latency. A recent demonstration [Espindola et al. 2025] showcases that offloading object detection to GPU-equipped edge nodes can significantly improve performance while reducing CPU load and thermal stress on mobile devices. Overall, these works demonstrate the feasibility and performance gains of edge-assisted MAR but focus predominantly on latency reduction rather than workload characterization.

In contrast, studies that explicitly address the modeling and characterization of edge-assisted MAR workloads remain limited. The 3GPP Release 17 specifications [3GPP 2022] define traffic models for XR considering data rates of 45/30 Mbps and update rates up to 60 Hz, which are representative of several application scenarios. Similarly, the work in [Morín et al. 2022] relies on HoloLens 2 and JPEG compression to estimate frame sizes and bit rates associated with different MAR tasks. However, both approaches focus exclusively on network traffic modeling, and do not rely on real-world prototypes. A more comprehensive traffic-level analysis is presented in [Morín et al. 2024], which employs a real-world offloading architecture to characterize the video traffic generated by an object detection task. While the work provides valuable insights into IP traffic characteristics (e.g., frame size, inter-frame interval, and inter-packet arrival time) it does not address the computational and memory demands imposed on the edge infrastructure. Nevertheless, characterizing these demands is equally important, as edge environments are resource-constrained and typically shared by multiple services.

The MR-Leo prototype [Toczé et al. 2020] represents an important step toward addressing this gap by analyzing CPU and memory usage for a MAR application that offloads SLAM tasks to the edge. However, the study is limited to CPU-based execution, a single MAR task is investigated and network traffic is not modeled. Building upon this foundation, this work extends MR-Leo to support both SLAM and object detection pipelines and explicitly evaluates the impact of GPU acceleration on both tasks. By deriving statistical computation workload and IP traffic models based on empirical measurements, this work advances the state-of-the-art in the characterization of MAR applications executed on heterogeneous CPU–GPU edge platforms. To the best of our knowledge, this is the first work that provides a comprehensive characterization of the computational and IP traffic demands of the SLAM and object detection tasks, as summarized in Table 1.

3. Enhanced MAR Prototyping

We extend the original MR-Leo prototype by modifying its SLAM pipeline and introducing a completely new object detection module. The proposed implementation preserves the core architecture and functionalities of MR-Leo, while restricting changes to the MAR processing components. We call this enhanced version of the prototype eMR-Leo.

As illustrated in Figure 1, the eMR-Leo architecture follows a client–server model, where the client represents the UE and the server the edge node. On the client side, cam-

Work	Focus	Task	Real-Prototype	Resources
[Chatzopoulos et al. 2017]	Architecture	General AR	No	None
[Siriwardhana et al. 2021]	Architecture	General AR	No	None
[Chen et al. 2017]	Latency reduction	Cognitive assistance	Yes	None
[Pereira et al. 2021]	Latency reduction	Marker-based AR	Yes	None
[Zhang et al. 2022]	Latency reduction	6-DoF tracking	Yes	None
[Espindola et al. 2025]	Latency reduction	Object detection	Yes	None
[3GPP 2022]	Characterization	General AR	No	Network
[Morín et al. 2022]	Characterization	SLAM, object detection, rendering	No	Network
[Toczé et al. 2020]	Characterization	SLAM	Yes	CPU, memory, network
[Morín et al. 2024]	Characterization	Object detection	Yes	Network
This work	Characterization	SLAM, object detection	Yes	CPU, GPU, memory, network

Table 1: Summary of related work on edge-assisted MAR applications

era frames are captured, encoded, and transmitted to the edge. The server receives and queues frames, which are processed by one of two MAR tasks: SLAM or object detection. SLAM estimates device pose and builds a 3D representation of the environment, while object detection identifies objects with associated confidence scores. After processing, frames are sent to the rendering module, where visual augmentation is applied. In SLAM, enrichment consists of generating and overlaying a point cloud, enabling 3D object anchoring. In object detection, enrichment consists of drawing bounding boxes. The enriched frames are then returned to the client for display. The right-hand side of Figure 1 shows the visual outputs of each task. The client performs no image analysis, only encoding and transmission. Both client and server include a manager responsible for component configuration.

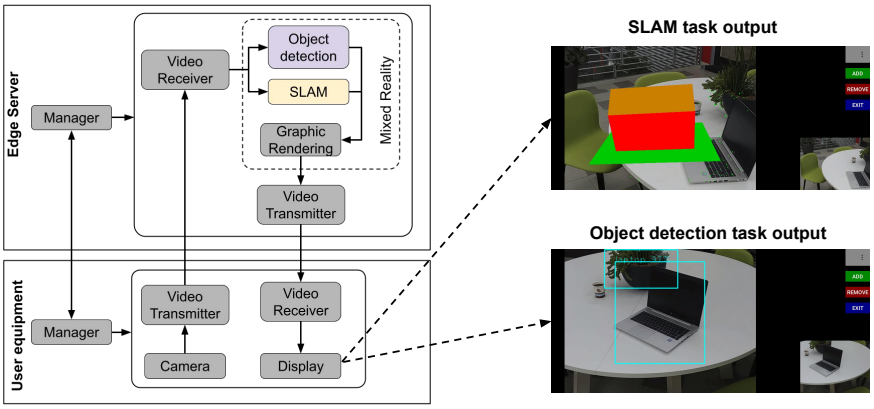


Figure 1: eMR-Leo architecture and example outputs of the implemented MAR tasks.

In Figure 1, gray components correspond to unmodified functionalities, yellow denotes modified components, and purple highlights modules introduced in this work. The eMR-Leo server is implemented in C++, while the client software run on Android and has parts written both in Java and C. Rendering is performed using the Pangolin[1] library.

[1] https://github.com/stevenlovegrove/Pangolin

The video transmission and reception services on both the client and server use H.264 video encoding and the Real-time Transport Protocol (RTP) over UDP transmission. To encode the streams in H.264 and pack the data in RTP frames, we use GStreamer[2]. Although many mobile MAR systems rely on TCP due to its reliability, TCP congestion control may introduce throughput oscillations that are undesirable for latency-sensitive workloads [Morín et al. 2024]. Thus, we opt for using UDP, which allows clearer observation of processing-related delays. In the following, we describe the modifications applied to the SLAM task and the design of the newly introduced object detection task.

3.1. SLAM Task

MR-Leo originally implements SLAM using ORB-SLAM2 [Mur-Artal and Tardós 2017], which operates through three parallel threads: tracking, local mapping, and loop closing. Tracking extracts ORB features and estimates camera pose, local mapping updates the 3D map, and loop closing corrects drift to maintain global consistency. In its original version, all processing is CPU-based. To enable GPU acceleration, eMR-Leo integrates a CUDA-accelerated ORB-SLAM2 implementation [Muzzini et al. 2023], which offloads computationally intensive tracking operations to the GPU and executes multiple CUDA kernels concurrently while minimizing CPU–GPU data transfers. This required replacing the original SLAM module in MR-Leo and updating the build configuration to support NVCC compilation.

3.2. Object Detection Task

As a newly introduced feature, the object detection task was implemented from scratch. We adopt the YOLO framework [Redmon et al. 2016] due to its widespread adoption and suitability for real-time object detection. OpenCV is used to handle video frame acquisition, preprocessing operations such as resizing and normalization, and post-processing steps including bounding box rendering and non-maximum suppression. To integrate object detection into eMR-Leo, we developed a new C++ class, `YoloProcesser`, responsible for executing the detection pipeline. Although YOLO is commonly used through Python-based interfaces, eMR-Leo is implemented in C++, requiring a compatible integration approach. For this purpose, we employ the DarkHelp wrapper[3], which enables the invocation of Darknet-based YOLO models from C++ code.

4. Performance Evaluation

This section presents a comprehensive evaluation of the eMR-Leo prototype, focusing on both computational behavior and IP traffic characteristics of the SLAM and object detection tasks. In the following, we first describe the experimental setup (Subsection 4.1). Next, we analyze the computational and memory demands of both tasks (Subsection 4.2). Finally, we characterize their associated IP traffic characteristics (Subsection 4.3). All the data generated in this evaluation is available on Github[4].

4.1. Evaluation Setup

To ensure consistent and reproducible conditions across experimental runs, we use a prerecorded video instead of real-time camera capture. The video has a duration of 60 seconds, a resolution of 640×480 pixels, and a frame rate of 30 Frames per Seconds (FPS).

[2] `https://gstreamer.freedesktop.org/`

[3] `https://github.com/stephanecharette/DarkHelp`

[4] `https://github.com/LABORA-INF-UFG/eMR-Leo.git`

Each video frame follows a fixed processing pipeline: (a) playback on the smartphone, (b) encoding, (c) uplink transmission to the edge server, (d) decoding, (e) task execution and visual enrichment, (f) rendering, (g) encoding, (h) downlink transmission back to the smartphone, (i) decoding, and (j) display to the user. This end-to-end pipeline is illustrated in Figure 2.

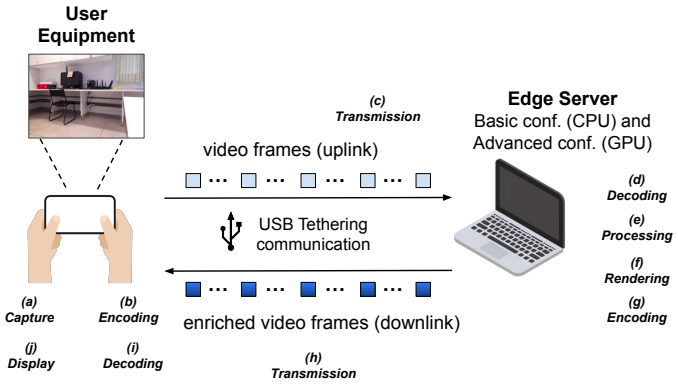


Figure 2: Evaluation setup used for testing.

The evaluation is conducted using a Lenovo Legion laptop as the edge server (running the server software) and a Samsung Galaxy A21s smartphone as the user device (running the client software). The edge server is equipped with an Intel Core i5-13420H processor, 16 GB of DDR5 RAM, and a NVIDIA GeForce RTX 3050 GPU with 6 GB of memory, running Ubuntu 24.04 LTS. The smartphone runs Android 12. Communication between the smartphone and the edge server is established via direct USB tethering to provide tighter experimental control. This setup allows traffic to traverse the full IP stack while avoiding the variability introduced by wireless technologies and intermediate network elements, allowing the analysis to focus on application-generated IP traffic characteristics rather than the behavior of a specific network technology. For processing, two execution scenarios are evaluated: one where all tasks run exclusively on the CPU (basic configuration) and another where computationally intensive tasks run on the GPU (advanced configuration). Each experiment is executed 30 times.

4.2. Computational and Memory Demands

To understand how different MAR workloads stress heterogeneous edge infrastructures, we analyze end-to-end latency and computational and memory resource usage. End-to-end latency is measured using the Frame Round Trip Time (FRTT), defined as the elapsed time between frame playback on the smartphone (step (a)) and display of the enriched frame (step (j)). Since the effectiveness of offloading is strongly influenced by processing delays at the edge, we also analyze the frame processing time at the server. In the following, we first present results for SLAM and, afterwards, for object detection.

SLAM

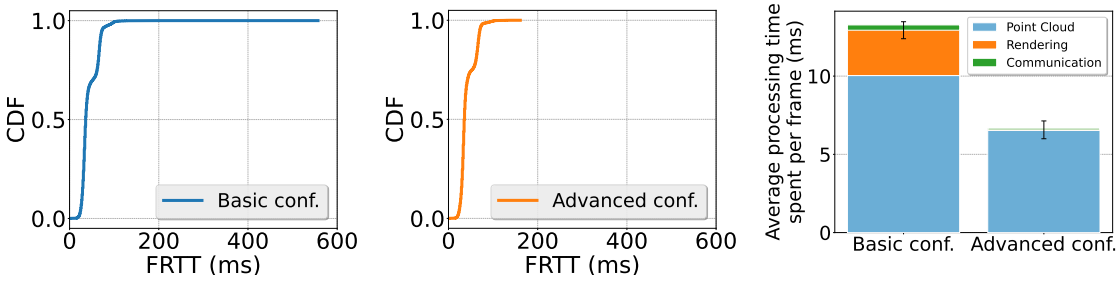
Figures 3a and 3b present the Cumulative Distribution Function (CDF) for the FRTT in the basic and advanced processing configurations, respectively. In both execution scenarios, 90% of the samples remain below 69 ms, sustaining an average throughput of

30 FPS. Because the client and edge server are connected via USB tethering, the measured FRTT primarily reflects application and edge-side processing delays, along with IP stack overhead. This leaves an approximate latency budget of 30 ms for the network segment to satisfy a typical end-to-end latency requirement of 100 ms for smartphone-based MAR applications [Toczé et al. 2020]. The figures also show that GPU acceleration has a limited impact on overall SLAM latency. This is better evidenced in Figure 3c, which shows the average processing time per video frame for SLAM in the edge server, broken down into point cloud generation, rendering, and communication. SLAM processing is dominated by point cloud generation, which remains largely CPU-bound in both scenarios. GPU acceleration reduces rendering time, but since rendering represents only a small portion of the total workload, the overall reduction in processing time is modest. Communication overhead stays minimal and unchanged across scenarios due to the controlled setup. Thus, for SLAM, offloading provides stable performance even without GPU usage.

Figure 4 further supports this conclusion. As illustrated in Figure 4a, CPU usage in the edge server peaks at 164% in the basic scenario, indicating multi-core utilization, while keeping higher than 100% in the advanced configuration. At the same time, in the advanced configuration, GPU peaks at only 32%, confirming that GPU acceleration provides limited benefit for SLAM. Figure 4c shows RAM usage in the basic scenario while Figure 4d illustrates RAM and Video RAM (VRAM) consumption for the advanced configuration on the edge server throughout the video transmission. In the basic scenario, RAM usage increases rapidly during the first seconds, driven by the vocabulary loading at startup and the feature extraction and pose estimation performed on the initial frames. After this period, ORB-SLAM2 enters a continuous tracking state in which data is typically processed one frame at a time. During this phase, the number of points in the spatial map may continue to grow, but at a lower rate than initialization. The advanced configuration uses a GPU-accelerated ORB-SLAM2 implementation library, which offloads rendering and selected image tracking to CUDA kernels. As a result, memory usage is split between RAM and VRAM. RAM still increases over time due to the expanding SLAM map, but at a slower rate, since some intermediate buffers and feature-processing data structures are now allocated on the GPU. VRAM usage rises early and then stabilizes, indicating that GPU memory is primarily used for fixed-size buffers and reusable CUDA data structures rather than for storing the long-lived SLAM state. Overall, these results show that the SLAM task remains predominantly CPU-bound, with GPU acceleration providing limited performance gains confined to rendering.

Object Detection

As with the SLAM task, we evaluate FRTT over UDP for the basic and advanced scenarios. Object detection presents a markedly different computational profile from SLAM. As shown in Figures 5a and 5b, GPU acceleration dramatically reduces FRTT, lowering the 90th percentile from 232 ms to 67 ms. This reduction is directly linked to the highly parallel nature of convolutional neural network inference, which maps efficiently onto GPU architectures. Only the advanced configuration satisfies typical MAR latency requirement (100 ms), reaching 30 FPS, demonstrating that CPU-only execution is insufficient for real-time object detection at the edge. The processing breakdown in Figure 5c highlights this difference. In the basic scenario, object detection dominates processing time

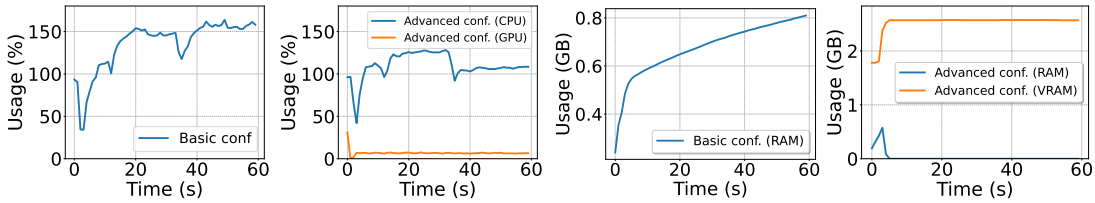


(a) FRTT for basic configuration

(b) FRTT for advanced configuration

(c) Breakdown for processing time

Figure 3: FRTT values for the SLAM task using the basic (3a) and the advanced scenarios (3b); (3c) breakdown of the average time spent to process a frame.



(a) CPU usage for basic configuration

(b) CPU and GPU usage for advanced configuration

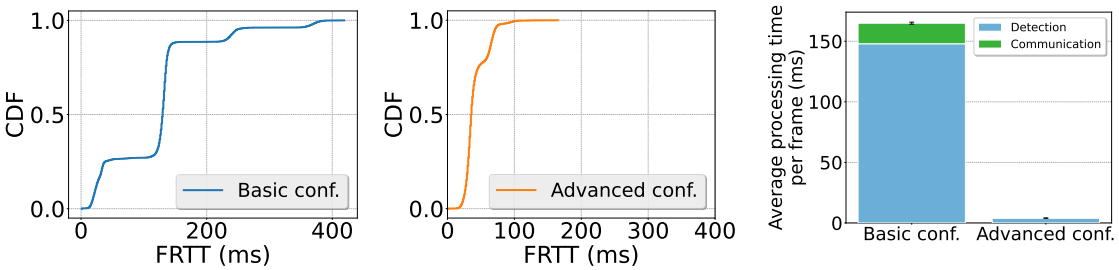
(c) RAM usage for basic configuration

(d) RAM and VRAM usage for advanced configuration

Figure 4: Computing and memory usage for the SLAM task.

(148 ms per frame), making it the primary performance bottleneck. GPU acceleration reduces this cost to approximately 4 ms per frame (a reduction of about 98%), enabling real-time performance. Communication overhead remains a minor component, again due to the USB tethering.

Figure 6 shows the computational and memory resource usages of the object detection task. Figure 6a shows the full CPU saturation (730%) in the basic execution scenario, confirming that object detection is compute-intensive and parallelizable. In contrast, GPU offloading in the advanced configuration (Figure 6b) reduces CPU load to 47% while GPU utilization stabilizes around 34%. Memory consumption for object detection is shown in Figures 6c and 6d. In the basic scenario (Figure 6c), RAM usage increases steadily during the first 40 seconds, which coincides with sustained high CPU utilization (Figure 6a). Because YOLO inference is executed entirely on the CPU, intermediate tensors, feature maps, and processing buffers are continuously allocated in system memory. Under prolonged high CPU load, garbage collection becomes less effective, leading to gradual accumulation of allocated memory. After approximately 40 seconds, RAM usage drops noticeably. This reduction correlates with a large number of frames being dropped, which reduce the number of detections processed. With fewer active inference operations, memory previously used for intermediate tensors can be released, resulting in the observed decline in RAM consumption. In contrast, in the advanced scenario, most intermediate tensors are allocated in VRAM and reused efficiently across frames. VRAM therefore rises early and stabilizes, while CPU usage drops significantly. With lower CPU

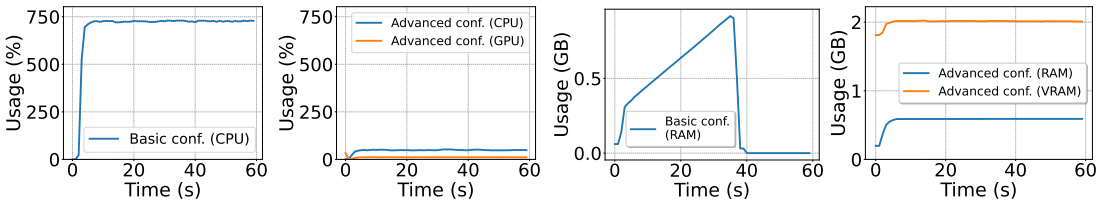


(a) FRTT for basic configuration

(b) FRTT for advanced configuration

(c) Breakdown for processing time

Figure 5: FRTT values for the object detection task using the basic (5a) and the advanced configurations (5b); (5c) breakdown of the average time spent to process a frame.



(a) CPU usage for basic configuration

(b) CPU and GPU usage for advanced configuration

(c) RAM usage for basic configuration

(d) RAM and VRAM usage in advanced configuration

Figure 6: Computing and memory usage for the object detection task.

pressure, RAM usage remains stable because fewer large buffers are handled on the CPU. Overall, the object detection task is highly parallel and benefits substantially from GPU acceleration, which shifts the computational burden away from the CPU, stabilizes memory usage by allocating most intermediate tensors in VRAM.

4.3. IP Traffic Analysis

We analyze IP traffic based on three key metrics: i) inter-packet interval, the time between successive packets within an individual frame; ii) inter-frame interval, the time between individual frames; and iii) frame size, defined as the size of each frame. For this analysis, we consider only the execution scenario that delivered the best performance for each task, namely the basic (CPU-only) configuration for SLAM and the advanced (GPU-accelerated) configuration for object detection. The IP traffic metrics are extracted from packet captures collected using `tcpdump`[5]. To isolate the communication behavior of MAR workloads, network traffic measurements were conducted separately from computational profiling. The experiment consisted of replaying 10 minutes of the pre-recording video used in Subsection 4.2, during which packet captures were collected and stored in pcap files. These traces were subsequently parsed to extract all IP traffic metrics used in this analysis. We divide the IP traffic generated during the experiments into three distinct

[5]https://www.tcpdump.org/

streams, each analyzed independently. *Stream 1* corresponds to the uplink transmission of encoded video frames from the smartphone to the edge server. This stream is the same for both tasks (SLAM and object detection) since the uploaded video is the same. *Stream 2* comprises the downlink transmission of enriched video frames from the edge server to the smartphone produced by the SLAM tasks. Finally, *Stream 3* represents the downlink transmission of enriched video frames from the edge server to the smartphone produced by the object detection tasks.

For the SLAM task, Stream 1, as shown in Figure 7, exhibits regular inter-packet and inter-frame intervals aligned with the 30 FPS video rate, confirming that video generation and encoding remain stable. Stream 2 shows similar temporal behavior but with slightly larger and more variable frame sizes. This variability reflects periods of denser SLAM visual augmentation (e.g., richer point cloud rendering), which increase spatial complexity and produce larger frames, and periods of more stable or sparsely augmented scenes, resulting in smaller frames. Despite this variability, timing regularity is preserved, indicating that SLAM augmentation increases payload size without disrupting frame pacing. For object detection, Stream 3, shown in Figure 8, exhibits a more pronounced increase in frame size variability compared to SLAM, reflecting the addition of bounding boxes and metadata. Importantly, inter-frame timing remains stable, demonstrating that object detection affects bandwidth demand rather than transmission timing.

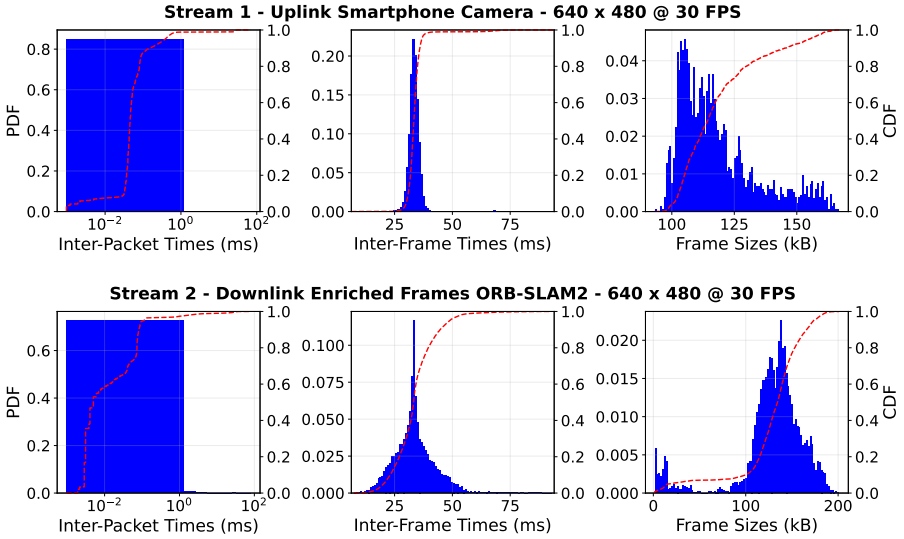


Figure 7: IP traffic metrics for Stream 1 and Stream 2 in the SLAM task.

5. Workload Modeling

Based on the collected data covering computational demand, memory usage, and IP traffic, we derive statistical workload models for both MAR tasks (SLAM and object detection). As in the traffic analysis, we consider only the execution scenario that delivered the best performance for each task. The number of instructions per frame is measured using the `perf` tool[6] for SLAM and NVIDIA Nsight Compute CLI[7] for object detection.

[6]https://man7.org/linux/man-pages/man1/perf.1.html

[7]https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

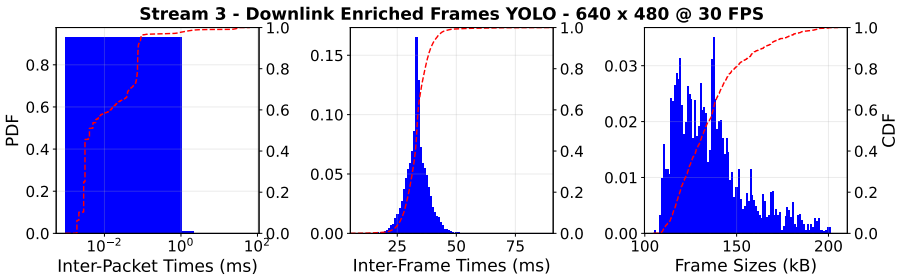
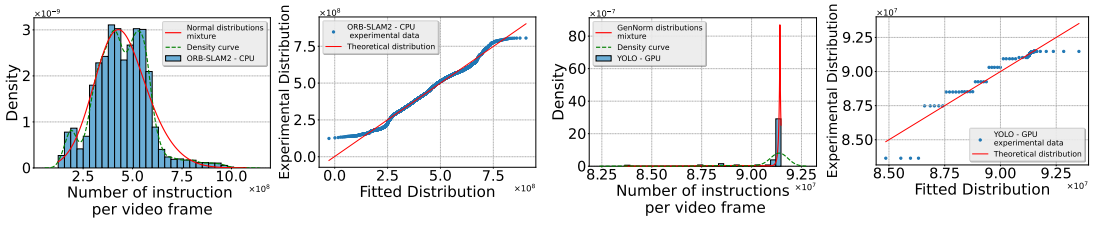


Figure 8: IP traffic metrics for Stream 3 in the object detection task.

Statistical fitting is performed using the Python `fitter` library[8] .

To characterize SLAM computation, we analyze the number of instructions executed per frame. As shown in Figure 9a, most frames require between 300 and 600 million instructions (MI), indicating relatively stable computational demand. The distribution is right-skewed, with occasional higher-cost frames. The fitted Gaussian mixture model captures this behavior using two dominant components: one centered around 404–407 MI and another near 529 MI, corresponding to regular tracking and more computationally intensive events such as relocalization or map updates. The Q–Q plot in Figure 9b confirms a good fit, with minor deviations only at the tails.

For the object detection task, Figure 9c shows a much more concentrated distribution, with most frames requiring approximately 91 MI. The workload is modeled as a mixture of two Generalized Normal distributions: a dominant low-variance component (82%) representing typical inference and a secondary higher-variance component (18%) capturing more complex scenes. Both components exhibit heavy-tailed behavior ($\beta < 2$), reflecting occasional deviations in processing demand. The Q–Q plot in Figure 9d shows good agreement in the central region, with expected deviations at the tails due to the stochastic nature of deep learning inference.



(a) SLAM (b) Q-Q plot (c) Object detection (d) Q-Q plot

Figure 9: Empirical and fitted distribution of the number of instructions per frame for the SLAM task (9a); quality of fitting for the SLAM (9b); Empirical and fitted distribution of the number of instructions per frame for the object detection task (9c); and quality of fitting for the object detection (9b).

Finally, Figure 10 presents the Kolmogorov–Smirnov (KS) test results for IP traffic metrics. For Stream 1 (uplink), the Johnson SU distribution best models frame size, inter-frame interval, and inter-packet interval. For Streams 2 and 3 (downlink), Johnson

[8]https://fitter.readthedocs.io/en/latest/

SU also fits frame size and inter-frame intervals, while inter-packet intervals are better represented by a lognormal distribution, indicating higher packet-level timing variability. Table 2 summarizes the statistical workload model, including the fitted distributions and parameters for computation, memory usage, and IP traffic of both SLAM and object detection tasks.

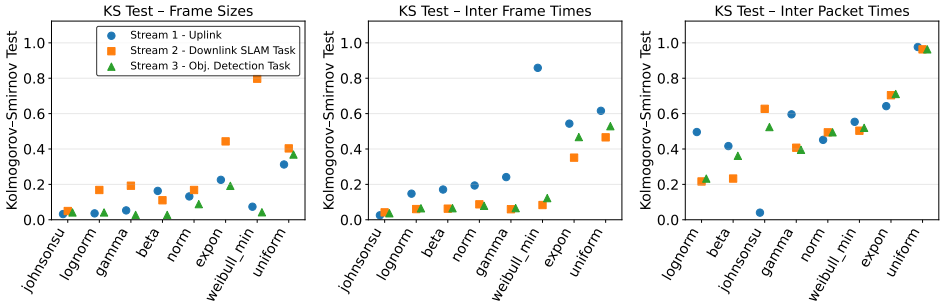


Figure 10: KS test for all the metrics and streams used for IP traffic.

Based on all data obtained during the characterization phase, a statistical model was developed describing the workload of the application's tasks. Table 2 presents this developed statistical model. The SLAM task is CPU-bound and is modeled using two normal distributions, characterized by their mean (μ) and standard deviation (σ) parameters, as summarized in the computation model presented in Table 2. Downlink traffic metrics for frame size and inter-frame intervals are modeled by a Johnson SU distribution, while inter-packet intervals follow a lognormal distribution. RAM usage starts at 408 MB (base) plus 288 MB per additional client. YOLO's workload is modeled via a generalized normal distribution with location (μ), scale (α), and shape (β parameters. Similar to SLAM, downlink frame and inter-frame metrics fit a Johnson SU distribution, while inter-packet intervals fit a lognormal distribution. Memory consumption begins at 208 MB of RAM, increasing by 416 MB per client to account for combined RAM and VRAM.

Task	**Hardware**	**Activity**	**Model**
Camera Stream	UE	Task arrival	Periodically, every 33 ms
		Uplink traffic	For frame size: $JohnsonSU(\gamma = -3.71,\ \delta = 1.54,\ \xi = 94.20,\ \lambda = 3.57)$ For inter-frame interval: $JohnsonSU(\gamma = -0.17,\ \delta = 1.04,\ \xi = 33.04,\ \lambda = 1.64)$ For inter-packet interval: $JohnsonSU(\gamma = -0.466,\ \delta = 0.416,\ \xi = 0.04090,\ \lambda = 0.00351)$
ORB-SLAM2	CPU	Task arrival	Periodically, every 33 ms
		Computation	For 43%: $\mathcal{N}(\mu = 494\,\text{MI}, \sigma = 147.97\,\text{MI})$ For 53%: $\mathcal{N}(\mu = 404\,\text{MI}, \sigma = 110.99\,\text{MI})$
		Downlink traffic	For frame size: $JohnsonSU(\gamma = 0.46,\ \delta = 1.04,\ \xi = 144.17,\ \lambda = 20.98)$ For inter-frame interval: $JohnsonSU(\gamma = -0.37,\ \delta = 1.58,\ \xi = 30.48,\ \lambda = 10.84)$ For inter-packet interval: $Lognormal(s = 1.91,\ \text{loc} = 9.54 \times 10^{-4},\ \text{scale} = 0.0103)$
		Memory (edge only)	408 MB
		Additional memory per client	288 MB
YOLO	GPU	Task arrival	Periodically, every 33 ms
		Computation	For 82%: $\mathcal{GN}(\mu = 91\,\text{MI}, \alpha = 0.40\,\text{MI}, \beta = 0.80)$ For 18%: $\mathcal{GN}(\mu = 89\,\text{MI}, \alpha = 2.05\,\text{MI}, \beta = 1.25)$
		Downlink traffic	For frame size: $JohnsonSU(\gamma = 2.74,\ \delta = 3.56 \times 10^{4},\ \xi = 104.56,\ \lambda = 4.06 \times 10^{5})$ For inter-frame interval: $JohnsonSU(\gamma = -0.10,\ \delta = 1.24,\ \xi = 32.88,\ \lambda = 4.44)$ For inter-packet interval: $Lognormal(s = 1.97,\ \text{loc} = 9.53 \times 10^{-4},\ \text{scale} = 0.00909))$
		Memory (edge only)	208 MB
		Additional memory per client	416 MB RAM 416 MB VRAM

Table 2: Overview of the load generation model.

6. Conclusion and Future Works

This work presented a comprehensive workload characterization of two fundamental MAR tasks, SLAM and object detection, executed in an edge-assisted architecture. By extending the MR-Leo prototype to support heterogeneous CPU–GPU processing, we evaluated computational demand, memory behavior, and IP traffic patterns under controlled conditions. The results reveal distinct resource profiles for the two tasks. SLAM exhibits a predominantly CPU-bound behavior, with multi-modal instruction demand and steadily increasing memory usage associated with map growth. GPU acceleration provides limited overall latency reduction, mainly improving rendering rather than core SLAM processing. In contrast, object detection demonstrates a highly parallel workload, where GPU acceleration drastically reduces processing latency while stabilizing memory usage by shifting inference buffers to VRAM. Traffic analysis further shows that both tasks maintain stable inter-frame timing aligned with 30 FPS, while enrichment increases frame size variability in the downlink. The resulting statistical models capture these distinct computational and traffic characteristics, enabling realistic workload generation for edge computing research. Although the specific workload values depend on the chosen software libraries and implementations (e.g., ORB-SLAM2 and YOLO), the characterization remains valuable because the literature currently lacks empirical models that jointly describe computational demand, memory usage, and IP traffic for edge-assisted MAR tasks. Thus, this work provides a necessary reference point and methodological foundation for future studies, even as implementations evolve.

Future work will extend this characterization along three main directions. First, we plan to incorporate additional MAR tasks, such as semantic segmentation and depth estimation, to broaden the coverage of vision-based workloads. Second, experiments will be conducted over real wireless networks (e.g., Wi-Fi and 5G) to study the interaction between workload variability and network dynamics. Finally, the derived workload models will be integrated into simulation and orchestration frameworks to evaluate resource allocation strategies, multi-user scalability, and adaptive offloading policies in heterogeneous edge environments.

Acknowledgments

This work was partially supported by CAPES under the grant no. 88887.849792/2023-00, MCTIC/CGI.br/FAPESP under grant no. 2020/05127-2 (SAMURAI project), by CNPq under grant no. 306283/2025-5, and by RNP/MCTIC under grant no. 01245.020548/2021-07 (Brasil 6G project).

References

3GPP (2022). Study on XR (extended reality) evaluations for NR, V17.0.0. [Online], Available at `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3736`.

Cadena, C. et al. (2016). Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. *IEEE Transactions on Robotics*, 32(6):1309–1332.

Chatzopoulos, D. et al. (2017). Mobile Augmented Reality Survey: From Where We Are to Where We Go. *IEEE Access*, 5:6917–6950.

Chen, Z. et al. (2017). An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proc. of the Second ACM/IEEE Symposium on Edge Computing*. Association for Computing Machinery.

Espindola, G. et al. (2025). Demonstrating the Advantages of Computational Offloading of XR Services via WebAssembly. In *2025 IEEE Network Operations and Management Symposium*, pages 1–3.

Hammad, N. et al. (2023). V-Light: Leveraging Edge Computing For The Design of Mobile Augmented Reality Games. In *Proc. of the 18th International Conference on the Foundations of Digital Games*. Association for Computing Machinery.

Milgram, P. and Kishino, F. (1994). A taxonomy of mixed reality visual displays. *IEICE Trans. Inf. Syst.*, 77(12).

Morín, D. G. et al. (2022). Toward the Distributed Implementation of Immersive Augmented Reality Architectures on 5G Networks. *IEEE Communications Magazine*, 60(2):46–52.

Morín, D. G. et al. (2024). An eXtended Reality Offloading IP Traffic Dataset and Models. *IEEE Transactions on Mobile Computing*, 23(6):6820–6834.

Mur-Artal, R. and Tardós, J. D. (2017). ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262.

Muzzini, F. et al. (2023). Brief Announcement: Optimized GPU-accelerated Feature Extraction for ORB-SLAM Systems. In *Proc. of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery.

Pereira, N. et al. (2021). ARENA: The Augmented Reality Edge Networking Architecture. In *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 479–488.

Redmon, J. et al. (2016). You only look once: Unified, real-time object detection. In *Proc. of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

Siriwardhana, Y. et al. (2021). A Survey on Mobile Augmented Reality With 5G Mobile Edge Computing: Architectures, Applications, and Technical Aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192.

Toczé, K. et al. (2020). Characterization and modeling of an edge computing mixed reality workload. *Journal of Cloud Computing*, 9(46).

Wang, X. et al. (2025). Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models. *ACM Comput. Surv.*, 57(9).

Zhang, W. et al. (2022). EdgeXAR: A 6-DoF Camera Multi-Target Interaction Framework for MAR with User-Friendly Latency Compensation. *Proc. ACM Hum.-Comput. Interact.*, 6(EICS).

Zou, Z. et al. (2023). Object Detection in 20 Years: A Survey. *Proceedings of the IEEE*, 111(3):257–276.