To Offload or Not To Offload: Model-driven Comparison of Edge-native and On-device Processing In the Era of Accelerators

Nathan Ng
University of Massachusetts Amherst
Amherst, MA, USA
kwanhong@cs.umass.edu

David Irwin
University of Massachusetts Amherst
Amherst, MA, USA
irwin@ecs.umass.edu

Ananthram Swami
DEVCOM Army Research Lab
Adelphi, MD, USA
ananthram.swami.civ@army.mil

Don Towsley
University of Massachusetts Amherst
Amherst, MA, USA
towsley@cs.umass.edu

Prashant Shenoy
University of Massachusetts Amherst
Amherst, MA, USA
shenoy@cs.umass.edu

Abstract

Computational offloading is a promising approach to overcome resource constraints on client devices by moving some or all application computations to remote servers. With the advent of specialized hardware accelerators, client devices can now perform fast local processing of tasks such as machine learning inference, reducing the need for offloading. However, edge servers with accelerators also offer faster processing for offloaded tasks than was previously possible. In this paper, we present an analytic and experimental comparison of on-device processing and edge offloading across accelerator, network, multi-tenant, and workload scenarios, with the goal of understanding when to use local processing versus offloading. We present models that leverage queuing theory to derive explainable closed-form equations for their end-to-end latencies, which yield precise, quantitative performance crossover predictions to guide adaptive offloading. We validate our models across a range of settings and show that they achieve a mean absolute percentage error of 2.2% compared to observed latencies. We further use these models to develop a resource manager for adaptive offloading and demonstrate its effectiveness in dynamic multi-tenant edge environments.

CCS Concepts

• **Computing methodologies** → **Model development and analysis**;
• **Computer systems organization** → **Distributed architectures**.

Keywords

Edge offloading, hardware accelerators, performance modeling

ACM Reference Format:
Nathan Ng, David Irwin, Ananthram Swami, Don Towsley, and Prashant Shenoy. 2026. To Offload or Not To Offload: Model-driven Comparison of Edge-native and On-device Processing In the Era of Accelerators. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3777884.3797816

ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797816

1 Introduction

Over the past decade, cloud computing has become the predominant approach for running online services in domains ranging from finance to entertainment. In recent years, a new class of online services has emerged that requires low-latency processing to meet end-user requirements. Examples of such applications include autonomous vehicles, interactive augmented and virtual reality (AR/VR), and human-in-the-loop machine learning inference [26]. Edge computing, a complementary approach to cloud computing, is a promising method for meeting the needs of such latency-sensitive applications. Conventional wisdom has held that since edge resources are deployed at the edge of the network and are closer to end-users, edge-native processing can provide lower end-to-end latencies than cloud processing. However, recent research [1, 47] has shown that this conventional wisdom does not always hold—in certain scenarios, edge processing can yield *worse* end-to-end latency despite its network latency advantage over the cloud. In particular, since edge clusters are resource-constrained relative to the cloud, research has shown that resource bottlenecks can arise under time-varying workloads and increase the processing latency at the edge, negating its network latency advantage and requiring careful resource management to maintain its benefits [47].

A related trend is the emergence of *computational offloading* from local devices to remote resources such as those at the edge. Computational offloading [5, 41] involves using nearby edge resources to take on some, or all, of an application's processing demands. Such offloading is performed to overcome resource constraints on a device by leveraging the more abundant processing capacity at edge servers. Computational offloading is well-established in mobile computing where mobile devices, which are resource- or battery-constrained, offload local computations to edge servers [39]. In this case, local devices pay a so-called mobility penalty [42], which is the additional network latency to access edge servers, but then benefit from faster edge processing that outweighs this latency overhead. Conventional wisdom holds that remote edge processing is faster than local processing despite the network overhead of accessing remote resources due to significant resource constraints at local devices.

With the advent of hardware accelerators in recent years, the conventional wisdom about the benefits of computational edge offloading for local devices needs to be rethought. For example, the rise of the so-called AI PC with onboard AI accelerators [14] has enabled local AI processing for many applications without the need

for edge offloading. Similarly, mobile phones are equipped with neural accelerators [40] that enable efficient local inference, while IoT devices such as cameras have specialized accelerators to perform visual tasks (e.g., facial recognition) locally [27]. At the same time, edge servers can also be equipped with more powerful accelerators, such as GPUs, enabling offloaded tasks to also run more efficiently or faster than before. Thus, in the era of programmable accelerators, the question of whether to offload computations from local devices to the edge and when it is advantageous to do so needs to be reexamined. In particular, such a reexamination needs to address several questions: (i) Under what scenarios will local on-device processing using accelerators outperform edge offloading? (ii) Are there still scenarios where edge offloading has an advantage over local processing? (iii) How does multi-tenancy at edge servers affect the decision to offload or execute on-device?

Motivated by these questions, the primary contribution of this paper is to develop a model-driven approach to analytically compare on-device processing and edge offloading in the era of accelerators and to optimize adaptive offloading decisions in dynamic edge environments. Our hypothesis is that there is no a priori clear advantage of edge offloading over on-device processing in the presence of accelerators, and that the choice of where to perform the processing depends on hardware, network, and workload characteristics. The novelty of this work lies in applying queuing-theoretic results to develop explainable analytic models for end-to-end latency in local and edge processing. Our models yield precise, quantitative performance-crossover predictions that can be embedded in OS schedulers and resource managers to dynamically decide whether to run an application component on the device or at the edge. In developing our analytic models and validating our hypothesis, this paper makes the following contributions.

- We develop queuing-theory-based models to characterize the behavior of device and edge accelerators when processing requests in on-device and edge offloading scenarios. Our modeling adopts a two-level approach: it first uses estimated service times obtained through profiling or prediction techniques as input, and then embeds them into queuing models to predict end-to-end response times. We develop closed-form analytic bounds using these models to determine when one approach outperforms the other across hardware, network, and workload scenarios. We further extend the analysis to model multi-tenant edge servers serving multiple clients, and demonstrate how the models can be extended to capture device–edge collaborative (i.e., split) processing.
- We experimentally validate our models and bounds across diverse device and edge accelerators, network configurations, and workloads spanning multiple model architectures, including deep neural network (DNN), recurrent neural network (RNN), and large language model (LLM). The results show that our models accurately predict on-device and offloading performance, achieving a 2.2% mean absolute percentage error, with 91.5% of predictions within $\pm 5\%$ and all within $\pm 10\%$ of the observed latency.
- We develop a resource manager that uses our models to adapt between on-device processing and edge offloading under real-world dynamics. We present a case study demonstrating that our models enable effective adaptation to fluctuating request rates at multi-tenant edge servers.

Table 1: Characteristics of common hardware accelerators.

Accelerator	Power	Memory	Workload
Google Edge TPU [9]	2 W	8 MB	ML Inference
NVIDIA Jetson TX2 [29]	15 W	8 GB	GPU compute
NVIDIA Jetson Orin Nano [32]	15 W	8 GB	GPU compute
NVIDIA A2 GPU [30]	60 W	16 GB	GPU compute

2 Background

This section provides background on computational offloading, hardware accelerators, and on-device processing.

2.1 Computational Offloading

Computational offloading is a well-known approach where resource-constrained client devices offload some or all of their application workload to a remote server. In the case of mobile devices such as AR/VR headsets, tablets, or smartphones, offloading can use nearby edge resources to perform latency-sensitive tasks [5, 41]. While such edge offloading involves a network hop to the edge server (also known as the mobility penalty [42]), subsequent processing at the edge is assumed to be much faster since edge resources are much less constrained than those of devices. In general, computational offloading offers benefits when local processing is constrained and the benefits of accessing faster processing at the edge or in the cloud outweigh the cost of additional network latency. While cloud offloading is also common, this paper focuses on edge offloading.

2.2 Hardware Accelerators

In recent years, a wide variety of accelerators have been developed to speed up diverse compute tasks, particularly machine learning workloads. Table 1 depicts the characteristics of several commonly deployed device and edge accelerators. For example, NVIDIA TX2, Orin Nano, and A2 are all tailored for AI workloads such as computer vision. With their small form factor and energy-efficient design, accelerators are increasingly integrated into mobile devices. For example, Apple devices include a Neural Engine to accelerate on-device machine learning tasks [40]. With accelerators, fast on-device processing of tasks such as machine learning inference has become feasible without relying on edge offloading. Meanwhile, edge servers can also be equipped with even more powerful GPUs. This enables offloaded tasks to also be accelerated at the edge, making edge offloading useful for more compute-intensive tasks that a device may not be able to handle using local resources.

2.3 On-device Processing versus Edge Offloading

As noted above, accelerators on client devices enable many tasks to run on-device, avoiding network hops and thus often outperforming edge offloading. However, edge servers equipped with GPUs or other accelerators can still outperform local devices for compute-intensive tasks. Thus, in the era of accelerators, local on-device processing may be faster in some cases, while offloading to edge accelerators can be faster in others (after accounting for network latency).

Even with extensive offline profiling, selecting the optimal execution strategy remains challenging due to real-world dynamics. On the one hand, mobile devices are subject to network variability, and since edge offloading requires transmitting input data, these fluctuations can unpredictably affect network delays and offloading performance. Additionally, devices running long-duration applications such as

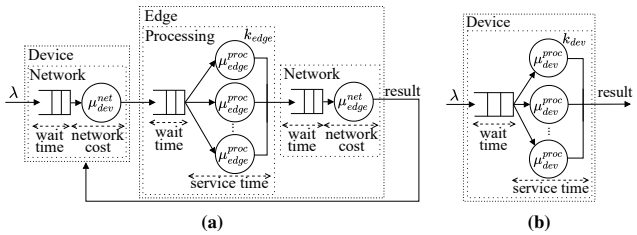

Figure 1: Modeling request execution under (a) edge offloading and (b) on-device processing.

live object detection may face battery constraints, and the OS may underclock processors to conserve energy, increasing on-device execution times. On the other hand, edge servers are *multi-tenant* systems shared by multiple devices, which can lead to interference and higher latencies. Since devices independently decide when to offload tasks, the edge server's load can fluctuate dynamically with the number and type of incoming requests. Unlike cloud data centers, edge servers are resource-constrained and cannot elastically scale all applications under high load. Workload spikes in multi-tenant scenarios can thus increase queuing delays and overall execution times. Moreover, GPUs and other accelerators on edge servers often lack isolation mechanisms such as GPU virtualization [34], leading to performance interference between applications. Consequently, execution times at the edge can vary, making it challenging to decide whether to offload under dynamic workloads.

The above observations motivate the primary research question addressed in this paper: *Considering the hardware processing capabilities of the device and edge server, along with the workload characteristics and real-world dynamics, under what scenarios does one strategy outperform the other?* Specifically, does edge offloading still have a role to play when devices are equipped with on-board accelerators? How do workload dynamics, network dynamics, and edge multi-tenancy impact each strategy and their relative performance with respect to one another? To address these questions, the next section develops analytic models for the end-to-end execution time of edge offloading and on-device processing, capturing the key factors that influence application performance.

3 Model-driven Performance Comparison

In this section, we develop analytic queuing models for edge offloading and on-device processing, and derive closed-form expressions for their expected latencies to compare the two approaches.

3.1 Analytic Queuing Models

Our analytic models are derived from queuing theory, a well-established mathematical tool for modeling request response times in computing systems. Queuing theory has previously been used to model the performance of edge and cloud applications [1, 11, 45], and here will serve as the foundation for analyzing the end-to-end latency of requests—from the time a task enters the system until it completes—under edge offloading and on-device processing.

Edge offloading. To model end-to-end request latency under edge offloading, we represent the times a request spends at the device and edge server as two separate queuing systems, as shown in Figure 1a. We assume that a request arrives at the device and is then forwarded ("offloaded") to the edge over the network, incurring network queuing and transmission delays. At the edge, the request is queued for processing and subsequently scheduled for execution. Since accelerators can process multiple requests in parallel, we model the edge accelerator as a parallel system with degree of parallelism k_{edge}. After processing, the result is sent back to the device, incurring network queuing and transmission delays on the reverse path.

On-device processing. We model on-device processing as a queuing system shown in Figure 1b. Requests generated by the device enter a local queue and are scheduled onto one of the accelerator cores, processed with degree of parallelism k_{dev}. Upon completion, results are returned to the local application and the request exits the system. This model represents a client device generating tasks, executing them on its accelerator, and returning the results to the application.

In queuing theory, a request's processing time at each queue comprises two components: queuing delay (the time spent waiting) and service time (the time taken to execute). Let w^{proc}_{dev}, w^{proc}_{edge}, s_{dev}, and s_{edge} represent the wait times and request service times at the device and edge, respectively. Edge offloading additionally incurs network latency as requests are sent to the edge and results are returned to the device. Let w^{net}_{dev} and w^{net}_{edge} denote the respective network queuing delays on the device and the edge. Also, let n_{req} denote the network transmission delay for sending a request from the device to the edge, and n_{res} the delay for returning the response.

With the above notations, we now model the end-to-end latency of requests executed using each strategy:

$$T_{edge} = w^{net}_{dev} + n_{req} + w^{proc}_{edge} + s_{edge} + w^{net}_{edge} + n_{res} \quad (1)$$

and

$$T_{dev} = w^{proc}_{dev} + s_{dev} \quad (2)$$

where T_{edge} and T_{dev} denote the end-to-end request latency under edge offloading and on-device processing, respectively.

3.2 Deriving Service Times on Accelerators

Our models require the service time of a workload as input to compute end-to-end response times. The service time reflects the computational demand on the target accelerator and represents the time needed to process a request on that platform. It can be obtained either through empirical profiling or via a predictive deep learning model. For profiling, many applications already provide function-level performance logging or profiling hooks [31], which can be used to derive service times. Alternatively, workloads can be benchmarked directly on the target hardware using tools such as [33, 38]. For model-based prediction, a learned performance model can be trained to predict the service time on a given hardware. For example, prior work predicts DNN inference latency by analyzing network structure and parameter size [17, 48], RNN inference latency by examining temporal dependencies and sequence unrolling behavior [15, 19], and LLM inference latency by using internal layer embeddings to predict remaining output length [43]. While any prior technique can be used with our analytic models, in our experiments we adopt a simple regression model from [48] to predict service times.

3.3 Deriving and Analyzing End-to-End Latency

We now compute the expected latency for edge offloading and on-device processing. Our analysis focuses on applications that require

Table 2: Notations used in analytic models.

Notation	Description
T_{dev}, T_{edge}	End-to-end latency of local processing/edge offloading
λ	Request arrival rate
B	Bandwidth of the device-to-edge network path
D_{req}, D_{res}	Payload sizes of request/result
$w^{net}_{dev}, w^{net}_{edge}$	Network wait time at the device/edge
n_{req}, n_{res}	Network transmission time of request/response
$\mu^{net}_{dev}, \mu^{net}_{edge}$	Service rate of device/edge NIC
k_{dev}, k_{edge}	Parallelism level of processors at device/edge
$w^{proc}_{dev}, w^{proc}_{edge}$	Processing wait time at the device/edge
s_{dev}, s_{edge}	Service time at the device/edge
$\mu^{proc}_{dev}, \mu^{proc}_{edge}$	Service rate of device/edge

results on the device, such as AR/VR workloads. While the models account for network delay when edge offloading returns results to the device, they can be adapted for applications where results are consumed on the edge (e.g., IoT sensing) by omitting this delay. In the following, we leverage queuing theory to understand the conditions under which one strategy outperforms the other.

3.3.1 Comparing Edge Offloading and On-Device Processing. We first examine the conditions under which edge offloading results in higher average latency compared to on-device processing. For workloads that benefit from hardware acceleration such as those utilizing GPUs, processing times can be significantly reduced compared to processing on general-purpose CPUs. In the following discussion, we refer to such workloads as accelerator-driven workloads. While our analysis primarily focuses on DNN inference, a representative component commonly found in AR/VR applications, the results generalize to other AI workloads as discussed in Sec. 3.5. Table 2 summarizes the notation used in our models.

LEMMA 3.1. *For accelerator-driven workloads, edge offloading incurs a higher average latency than on-device processing when*

$$\begin{aligned} s_{dev} - s_{edge} < {} & \frac{\lambda}{\mu^{net}_{dev}(\mu^{net}_{dev} - \lambda)} + \frac{\lambda}{\mu^{net}_{edge}(\mu^{net}_{edge} - \lambda)} + \frac{D_{req} + D_{res}}{B} \\ & + \frac{1}{2}\left(\frac{1}{k_{edge}\mu^{proc}_{edge} - \lambda} - \frac{1}{k_{edge}\mu^{proc}_{edge}}\right) \\ & - \frac{1}{2}\left(\frac{1}{k_{dev}\mu^{proc}_{dev} - \lambda} - \frac{1}{k_{dev}\mu^{proc}_{dev}}\right) \end{aligned} \tag{3}$$

The proof of Lemma 3.1 is provided in our technical report [28]. The result implies the following two remarks.

REMARK 3.1. *On-device processing is likely to outperform edge offloading for workloads with low computational demand.*

Explanation. Service time is determined by the workload's computational demand divided by the hardware's processing capacity. For edge offloading to outperform on-device processing, requests with lower demand must be processed much faster at the edge to overcome the network penalty. As demand decreases, the difference in service times $s_{dev} - s_{edge}$ shrinks, reducing the impact of the processing capacity gap between device and edge. Further, since service rate is the inverse of service time ($s = 1/\mu$), the terms with μ^{proc}_{dev} and μ^{proc}_{edge} in (3) also decrease for lighter requests. Meanwhile, network overhead terms remain unchanged, as they depend on payload size rather than processing demand. Consequently, the inequality in (3) is likely to hold for requests with low computational demand. □

REMARK 3.2. *On-device processing is likely to outperform edge offloading on slow networks or for workloads with large payloads.*

Explanation. The term $\frac{D_{req}+D_{res}}{B}$ in (3) increases when either the network bandwidth B decreases or the request and result sizes D_{req} and D_{res} become larger, thereby increasing the right-hand side of the inequality. In addition, longer transmission times reduce the NIC service rates at the device and edge (μ^{net}_{dev} and μ^{net}_{edge}), further increasing the right-hand side. As a result, the inequality is likely to hold under slow networks or for workloads with large payloads, making on-device processing faster than edge offloading. □

Practical takeaways. Lemma 3.1 establishes a bound on the expected service time difference between the device and the edge for edge offloading to be more effective. The bound depends on: (i) relative processing speeds of device and edge (s_{dev}, s_{edge} and corresponding μ_{dev}, μ_{edge}), (ii) workload (request rate λ), and (iii) network transmission overheads. In general, edge processing capacity must exceed device capacity by a factor to compensate for network penalties; otherwise, on-device processing is preferable.

Notably, request rate λ affects offloading decisions in two competing ways. As λ increases, both device and edge experience longer queuing delays, though the edge queue grows more slowly due to its faster processing speed, favoring offloading. However, under slow networks or for large-payload workloads, higher λ increases network queuing delay, penalizing offloading. Thus, the optimal strategy depends on the workload's payload size relative to network bandwidth and the performance gap between device and edge.

Remark 3.1 indicates that edge offloading is likely to outperform on-device processing for workloads with high computational demand as the edge's processing speed advantage dominates, while on-device processing is more efficient for light workloads since network overhead can outweigh the edge's advantage. Remark 3.2 shows that edge offloading becomes less efficient with limited bandwidth or large payloads, which increase network queuing and transmission delays, whereas on-device processing remains unaffected.

Extending to collaborative processing. Collaborative (i.e., split) processing involves partially processing requests locally on-device before offloading them to a remote edge server, reducing the data sent over the network [3, 4, 12, 13, 17]. To compute end-to-end response time, we combine our models for on-device processing and edge offloading into a tandem queuing network that accounts for both local computation and subsequent transmission and processing at the edge. Specifically, a request first enters the device components (Figure 1b) and is partially processed with service time s'_{dev}. The intermediate output of size D_{inter} is then transmitted (D_{inter}/B) and processed on the edge with service time s'_{edge} (Figure 1a). This combined model can then be used to estimate performance and inform decisions among local, edge, or collaborative processing.

3.4 Impact of Multi-Tenancy

Lemma 3.1 assumes a dedicated edge server for each client device. In practice, however, edge servers are shared across multiple devices and applications. Hence, we extend our analysis to incorporate multi-tenancy scenarios.

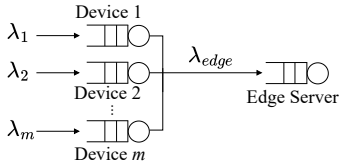

Figure 2: Edge application multiplexing.

Modeling edge application multiplexing. When an edge server is dedicated to a single client device, the workload (i.e., the request arrival rate) at the edge and at the device λ_{dev}^{proc} and λ_{edge}^{proc} are identical. However, when the edge server is shared across m devices, its workload under offloading becomes the sum of the individual device workloads, as illustrated in Figure 2. This higher workload increases queuing delays at the edge and may cause end-to-end offloading latency to exceed on-device processing latency. Since request arrivals at each client device are governed by independent Poisson processes, and the composition of independent Poisson processes results in a new Poisson process [37], it follows that the aggregate edge workload is also Poisson with an arrival rate $\lambda_{edge} = \sum_{i=0}^{m} \lambda_i$, where λ_i denotes the workload at the i-th device.

We assume that multiplexed applications have different service time demands, making the aggregate service time at the edge arbitrary. To capture the variability in service times across applications, we model the edge system as an $M/G/1$ system, where M denotes Poisson arrivals and G denotes the general (arbitrary) service time distribution seen at the edge. Let s_i be the expected service time of requests from device i at the edge. Then, s_{edge} is the weighted average of these service times $s_{edge} = \sum_i \frac{\lambda_i}{\lambda_{edge}} s_i$ with effective service rate $\mu_{edge}^{proc} = 1/s_{edge}$ and aggregate utilization $\rho_{edge} = \lambda_{edge}/\mu_{edge}^{proc}$. This leads to the following lemma for the multi-tenant case, with its proof provided in our technical report [28].

LEMMA 3.2. *For multi-tenant edge servers, the average latency of edge offloading is higher than on-device processing when*

$$\begin{aligned} s_{dev} - s_{edge} < & \frac{\lambda_{dev}}{\mu_{dev}^{net}(\mu_{dev}^{net} - \lambda_{dev})} + \frac{\lambda_{edge}}{\mu_{edge}^{net}(\mu_{edge}^{net} - \lambda_{edge})} \\ & + \frac{D_{req} + D_{res}}{B} + \frac{\rho_{edge} + \lambda_{edge} k_{edge} \mu_{edge}^{proc} Var[s_{edge}]}{2(k_{edge}\mu_{edge}^{proc} - \lambda_{edge})} \\ & - \frac{1}{2}\left(\frac{1}{k_{dev}\mu_{dev}^{proc} - \lambda_{dev}} - \frac{1}{k_{dev}\mu_{dev}^{proc}}\right) \end{aligned} \quad (4)$$

Practical takeaways. The expected wait time on multi-tenant edge servers is heavily influenced by service time variability, and co-located applications with highly variable processing times can significantly increase queuing delays. Application designers should therefore consider service time variability when assigning applications to edge servers. Grouping applications with similar demands can reduce variability, minimizing queuing delays and improving offloading performance. Further, the above analysis assumes that isolation features such as GPU virtualization *are not available on edge accelerators*. If such features were available, each device could be assigned to an isolated virtual GPU on the edge, and the offloading situation reduces to the one considered in Sec. 3.3.1 where each virtual GPU provides $\frac{1}{m}$-th of the capacity. However, many edge accelerators lack such features, resulting in a shared accelerator that services a combined workload.

3.5 Generalizing to Other ML Models

While the above analysis focuses on DNN workloads with deterministic service times, our models extend to other ML models by incorporating appropriate queuing formulations. For example, recurrent models have variable service times depending on input length, and LLM inference exhibits variable service times due to differing numbers of autoregressive decoding steps. To capture such variability, we model the processing stages as $M/M/1$ queues with exponential service times and effective service rates $k\mu$. The resulting closed-form response time expressions and derivations are provided in [28]. By selecting queueing formulations that match workload variability, our models can adapt to diverse ML workloads and provide accurate response time estimates as shown in Sec. 4.4.

4 Model Validation

In this section, we empirically validate the accuracy of our models for end-to-end response time predictions across diverse configurations. Additional experiments validating the impact of network bandwidths and request rates are provided in [28].

4.1 Experimental Setup

Workloads. We validate our models across DNN, RNN, and LLM workloads. For DNN workloads, we focus on image classification tasks for AR applications using three widely deployed architectures with distinct computational demands and input sizes: MobileNetV2 (224×224 inputs, ≈ 0.6 GFLOPs), InceptionV4 (299×299 inputs, ≈ 6.3 GFLOPs), and YOLOv8n (640×640 inputs, ≈ 8.7 GFLOPs). All requests are executed using TensorFlow 2.13.0 on frames extracted from a 720p video. For RNN, we use a 3-layer Long Short-Term Memory (LSTM) model implemented in TensorFlow Keras for sentiment classification, using the Amazon Alexa Topical Chat dataset [10]. For LLM, we use the Llama-3.2-1B model [25] run via llama.cpp [8], with prompts from the Hugging Face ShareGPT dataset [46]. To model dynamic workloads (e.g., triggered by motion detection), we implement a workload generator that generates requests following a Poisson process, running on a separate machine to avoid performance interference with the device.

Hardware. For client devices, we use the NVIDIA Jetson TX2 and Jetson Orin Nano, two widely deployed IoT platforms. For edge servers, we use a Dell PowerEdge R630 with an NVIDIA A2 GPU and a server with an NVIDIA RTX 4070 GPU with 1 Gbps network connectivity. These platforms span a wide performance range, from 1.3 TFLOPS (TX2) to 2.6 TFLOPS (Orin Nano), 4.5 TFLOPS (A2), and 29.1 TFLOPS (RTX 4070) in peak FP16 throughput. All platforms run Ubuntu 18.04, and we use the Linux Traffic Control (TC) subsystem to emulate different network bandwidths. For each accelerator, the degree of parallelism k depends on both the number of cores and the workload's processing and memory demands, and is not directly observable. We estimate k empirically by measuring how response time varies with request rate for each workload and selecting the value that best captures the observed scaling behavior.

4.2 Parameter Estimation

We now describe how the parameters used by our models to compute end-to-end response times are obtained in our evaluation. Other practical approaches for estimating them are discussed in Sec. 3.2.

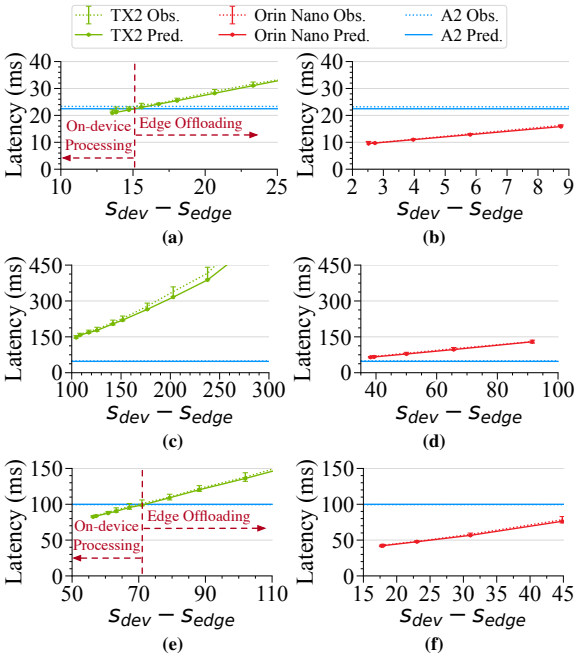

Figure 3: Latency comparison for DNN workloads: (a-b) MobileNetV2, (c-d) InceptionV4, and (e-f) YOLOv8n.

Estimating service times. We estimate service times using profiling and a regression–based prediction approach. For DNNs, we monitor inference durations using NVIDIA's nvidia-smi tool [33], which reports per-process execution times. For RNN and LLM workloads, we profile request latencies on a representative input set and use the average latency as input to the models. This profiling data informs our edge-offloading and on-device processing experiments. For collaborative processing, a tree-structured linear regression model is trained as described in [48] to predict partial service time, avoiding the profiling overhead of all possible split configurations.

Estimating network delays. We estimate the available network bandwidth using iperf [6]. In addition to queuing and transmission delays captured by our models, network delays include propagation and processing delays. Since requests are offloaded to a nearby edge server, propagation delay (signals traveling at the speed of light) is minimal and excluded from our models. Processing delay can be estimated from the round-trip time of probe packets and is excluded due to its negligible impact (<1 ms on gRPC) on overall latency.

Estimating system utilization. The system utilization ρ is calculated as the ratio of the arrival rate λ to the service rate μ of the system. The arrival rate λ can be estimated by applying a sliding window over incoming request timestamps. The service rate μ can be estimated from system logs based on the number of completed requests within a given time interval.

4.3 Impact of Workload Characteristics

We first study how workload characteristics influence offloading decisions using the three DNN models by comparing on-device processing on the TX2 and Orin Nano against offloading to an A2 GPU at 2 requests/s (RPS) over a 5 Mbps network, with the resulting average latency shown in Figure 3. For each device, we test all available GPU frequency settings to emulate varying device conditions, with each data point representing one setting. Figures 3a and 3b show that for MobileNetV2, TX2 achieves lower average latency in its top three frequency levels, while Orin Nano consistently outperforms offloading across all frequency levels. The trend reverses for InceptionV4 as shown in Figures 3c and 3d, where offloading provides lower average latency across all device configurations. Lastly, Figures 3e and 3f show that despite YOLOv8n's high compute demand, TX2 outperforms offloading in its top six frequency levels, while Orin Nano achieves lower average latency across all settings. We observe similar results when offloading to an RTX 4070 GPU and omit the details here due to space constraints.

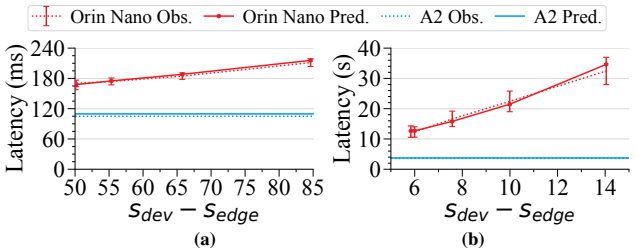

Figure 4: Latency comparison of execution strategies for (a) LSTM and (b) Llama-3.2-1B models.

Collectively, these results align with the qualitative observations stated in Remark 3.1 and Remark 3.2. While the performance ratio between device and edge is fixed, the absolute advantage of offloading scales with the workload's computational demands. For MobileNetV2 (low compute), the edge's relative speed advantage results in only small absolute gains that can be overshadowed by network delays of offloading. However, for InceptionV4 (high compute), the same performance ratio yields larger absolute improvements that outweigh the network delays. Moreover, YOLOv8n (large payloads) favors on-device processing because it avoids any network delay. In all cases, the model predictions closely match the observed response times for both approaches, achieving a mean absolute percentage error (MAPE) of 2.2%, with 91.5% of predictions falling within $\pm 5\%$ of the observed latency and 100% within $\pm 10\%$.

Key takeaway. *Offloading reduces average latency when the edge's performance gain exceeds its associated network delay. Our models accurately predict latency for both strategies with an MAPE of 2.2%.*

4.4 Impact of Complex Deep Learning Models

Figure 4 compares average latency for RNN and LLM workloads on the Orin Nano versus offloading to the A2 GPU under moderate loads. Since service times vary across requests in these models, we use the $M/M/1$ queuing model with effective service rates $k\mu$ to account for this variability. Figure 4a shows that for the LSTM model, offloading achieves lower average latency as the input utterances are short and their payloads incur only minimal network delay. In the LLM case shown in Figure 4b, which has a higher computational demand than LSTM, the latency reduction from offloading becomes even more pronounced, reaching up to 30 seconds. In both cases, our model predictions closely match the observed latencies.

Key takeaway. *Our models generalize to other deep learning workloads by integrating appropriate queuing models.*

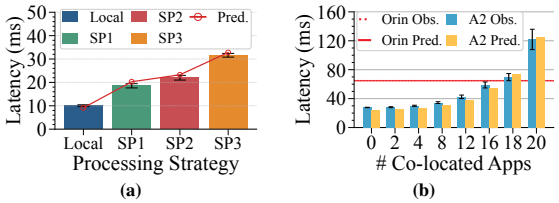

Figure 5: Latency comparisons under different (a) collaborative processing configurations and (b) degrees of multi-tenancy.

4.5 Impact of Collaborative Processing

To validate our models' ability to capture collaborative (split) processing, we partition MobileNetV2 across the Orin Nano and A2 over a 50 Mbps network, evaluate three split points that progressively offload computation to the edge against full on-device execution, with early layers running on the device and remaining layers executed at the edge. Figure 5a plots the observed latencies of each configuration alongside model predictions. As shown, the latency increases with the amount of compute offloaded. This is because the intermediate results of later layers grow in size, adding to the network transfer overhead, making local processing faster than split processing for MobileNetV2 under the chosen network configuration. Note that other models may exhibit different performance characteristics, which we omit due to space constraints. Across all split points, the model-predicted latencies closely align with observed values.

Key takeaway. *Our models can accurately predict the performance of device-edge collaborative processing.*

4.6 Impact of Multi-Tenancy

Figure 5b compares average latency of the two strategies as the number of co-located InceptionV4 applications on the edge increases, with each application receiving 2 RPS from its client. The results show that offloading yields lower latency than on-device processing when a single application is deployed, but latency increases with more co-located applications due to resource contention on the A2 GPU, which lacks isolation mechanisms. Notably, there is a crossover point at 18 applications, beyond which on-device processing becomes more efficient. Note that the on-device processing latency remains stable regardless of the number of concurrent applications, as each device operates independently. In all cases, our models closely predict the latency for both strategies.

Key takeaway. *Offloading latency increases with the degree of edge multi-tenancy due to resource contention.*

5 Models in Action

In this section, we show how a resource manager can leverage our models to adaptively switch between execution strategies. We outline its algorithm and demonstrate its effectiveness through a case study highlighting adaptation to multi-tenant edge servers. Another study on network variability is presented in [28].

5.1 Model-Driven Adaptive Resource Management

The resource manager runs locally on the device and collects runtime metrics including network bandwidth, edge server load, and request arrival rates every 5 seconds. At each epoch, it inputs these metrics into our models to estimate the average latency for each strategy, and selects the strategy with the lowest predicted latency to execute requests. Algorithm 1 outlines the decision-making process of the resource manager. Line 1 predicts the average latency of on-device processing based on the current arrival rate and the device's estimated service time. Lines 2-5 compute the predicted offloading latency for each edge server, accounting for server load, service rate, and network conditions. Finally, Lines 6-9 execute the request using the strategy with the lower predicted latency. This decision algorithm is lightweight, taking under 2 ms as it relies solely on closed-form computations in the analytic models.

Algorithm 1: Model-Driven Adaptive Offloading

Input:
λ_{dev}: Device request arrival rate
D_{req}, D_{res}: Request/result payload size
$s^{proc}_{dev}, s^{proc}_{edge}$: Service times of device and edge
$\mathscr{E}$: Number of edge servers
B: Network bandwidth of device-to-edge network path
$\{\mu^{proc}_{edge,E}\}_{E=1}^{\mathscr{E}}, \{\lambda_{edge,E}\}_{E=1}^{\mathscr{E}}$: Edge server aggregated service and arrival rates
Output:
Execution strategy

1 $T_{dev} \leftarrow M/D/1(\lambda^{proc}_{dev}, 1/s^{proc}_{dev}) + s^{proc}_{dev}$
2 $T^{req}_{net} \leftarrow M/M/1(\lambda_{dev}, B/D_{req}) + D_{req}/B$
3 $T^{res}_{net} \leftarrow M/M/1(\lambda_{edge,E}, B/D_{res}) + D_{res}/B$
4 **for** $E \leftarrow 1$ **to** $\mathscr{E}$ **do**
5 $T_{edge,E} \leftarrow T^{req}_{net} + M/G/1(\lambda_{edge,E}, \mu^{proc}_{edge,E}) + s^{proc}_{edge} + T^{res}_{net}$
6 **if** $T_{dev} < \min(\{T_{edge,E}\}_{E=1}^{\mathscr{E}})$ **then**
7 **return** *on_device_processing()*
8 **else**
9 **return** *offload*$(\arg\min_E T_{edge,E})$

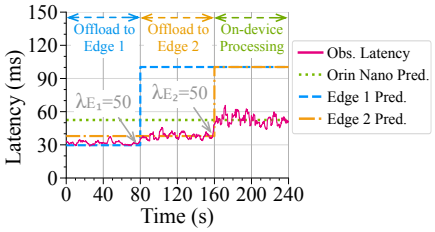

Figure 6: Latencies under varying edge server workloads as the resource manager adapts.

5.2 Case Study: Multi-Tenant Edge Servers

We use two edge servers, E_1 and E_2, running YOLOv8n inference on A2 GPUs with initial request rates of 10 and 30 RPS, while the TX2 runs the resource manager with a workload of 10 RPS. Figure 6 shows the observed latencies alongside the predicted latencies for offloading to E_1, E_2, and processing on TX2. At $t = 0$, the resource manager offloads requests to E_1, as it has the lowest predicted latency. At $t = 80$, we increase λ_{E_1} to 50 to emulate a workload spike. The updated predictions identify E_2 as the lower-latency option, prompting the manager to redirect requests accordingly. At $t = 160$, we increase λ_{E_2} to 50. With both servers at 50 RPS, on-device processing becomes the lowest predicted latency, and the manager switches to local execution. This sequence demonstrates how model predictions drive dynamic adaptation to changing edge workloads.

Key takeaway. *Our models enable adaptation to workload changes on multi-tenant edge servers to maintain low latency.*

6 Related Work

Edge offloading. Edge offloading is widely adopted to enable resource-constrained devices to execute compute-intensive tasks using edge resources [18, 20, 35, 36, 49]. Prior work selectively offloads parts of application pipelines, such as mobile AR [18], or dynamically offloads DNN layers under energy constraints [20]. In contrast, our work adopts a model-driven approach for latency estimates and generalizes to collaborative processing.

Model-aware resource allocation. Queuing models have been widely used to predict the performance of traditional CPU-bound workloads [2, 7, 16, 45, 47]. More recently, prior work has modeled hardware accelerators, such as GPU batching latency in DNN inference [24] and inference performance for DNN placement [21]. In contrast, our models extend beyond DNN inference to other ML workloads, as demonstrated in Sec. 4.4.

Modeling distributed workloads. Several studies use analytic models for distributed computing, e.g., FogQN [44] for fog–cloud processing and Loghin et al. [22] for MapReduce in hybrid edge–cloud environments. However, they ignore accelerator-specific queuing and multi-tenancy effects. Our work complements existing efforts by addressing these missing dimensions through a unified queuing-based modeling framework that informs resource management decisions.

7 Conclusions, Limitations, and Future Work

This paper presented analytic models for on-device processing and edge offloading with accelerators, providing performance predictions for adaptive execution decisions. We validated their accuracy across a range of scenarios and developed a resource manager that applies these predictions to adapt to real-world dynamics.

Our models assume independent Poisson arrivals and infinite queue capacity. These assumptions can be relaxed using a $G/G/1$ formulation and finite-buffer queuing models with known results from prior work [11, 23]. The closed-form expressions also rely on steady-state assumptions typical of queuing theory, which may not fully capture rapid fluctuations in network conditions. As future work, we plan to evaluate the models in real cellular and wireless networks and extend them to multi-stage pipelines.

Acknowledgments

We thank the reviewers for their valuable comments. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, 23091241, 2325956, and the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196 (IoBT CRA).

References

[1] Ahmed Ali-Eldin et al. 2021. The hidden cost of the edge: a performance comparison of edge and cloud latencies. In *SC'21*. ACM.
[2] Pradeep Ambati et al. 2020. Waiting Game: Optimally Provisioning Fixed Resources for Cloud-Enabled Schedulers. In *SC'20*. IEEE.
[3] Amin Banitalebi-Dehkordi et al. 2021. Auto-Split: A General Framework of Collaborative Edge-Cloud AI. In *KDD'21*. ACM.
[4] Yichong Chen et al. 2025. CEED: Collaborative Early Exit Neural Network Inference at the Edge. In *INFOCOM'25*.
[5] A. Davis et al. 2004. Edgecomputing: extending enterprise applications to the edge of the internet. In *WWW'04*. ACM.
[6] Jon Dugan et al. 2024. iPerf - Network Bandwidth Measurement. https://iperf.fr/
[7] Anshul Gandhi et al. 2019. Leveraging Queueing Theory and OS Profiling to Reduce Application Latency. In *Middleware'19*. ACM.
[8] Georgi Gerganov et al. 2023. *llama.cpp*. https://github.com/ggml-org/llama.cpp
[9] Google. 2026. Coral Edge TPU. https://www.coral.ai/products/accelerator.
[10] Karthik Gopalakrishnan et al. 2023. Topical-Chat: Towards Knowledge-Grounded Open-Domain Conversations. arXiv:2308.11995
[11] M. Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.
[12] Chuang Hu et al. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *INFOCOM'19*. IEEE.
[13] Jin Huang et al. 2020. CLIO: enabling automatic compilation of deep learning pipelines across IoT and cloud. In *MobiCom'20*. ACM.
[14] Intel Corporation. 2025. The AI PC Powered by Intel is Here. https://www.intel.com/content/www/us/en/products/docs/processors/core-ultra/ai-pc.html
[15] Yong Ji et al. 2024. An Active Learning based Latency Prediction Approach for Neural Network Architecture. In *NNICE'24*.
[16] Lili Jiang et al. 2021. Performance analysis of heterogeneous cloud-edge services: A modeling approach. *Peer-to-Peer Networking and Applications* 14 (01 2021).
[17] Yiping Kang et al. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017).
[18] Zeqi Lai et al. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *MobiCom'17*. ACM.
[19] Zhuojin Li et al. 2023. Predicting Inference Latency of Neural Architectures on Mobile Devices. In *ICPE'23*. ACM.
[20] Zengpeng Li et al. 2024. Energy-efficient offloading for DNN-based applications in edge-cloud computing: A hybrid chaotic evolutionary approach. *J. Parallel Distrib. Comput.* 187, C (May 2024).
[21] Qianlin Liang et al. 2023. Model-driven Cluster Resource Management for AI Workloads in Edge Clouds. *ACM Trans. Auton. Adapt. Syst.* (March 2023).
[22] Dumitrel Loghin et al. 2019. Towards Analyzing the Performance of Hybrid Edge-Cloud Processing. In *EDGE'19*.
[23] K. T. Marshall. 1968. Some Inequalities in Queuing. *Oper. Res.* 16, 3 (June 1968).
[24] Matthew L. Merck et al. 2019. Characterizing the Execution of Deep Neural Networks on Collaborative Robots and Edge Devices. In *PEARC'19*. ACM.
[25] Meta. 2024. Llama-3.2-1B. https://huggingface.co/meta-llama/Llama-3.2-1B
[26] Eduardo Mosqueira-Rey et al. 2022. Human-in-the-loop machine learning: a state of the art. *Artif. Intell. Rev.* 56, 4 (Aug. 2022).
[27] Netatmo. 2026. Smart Camera. https://www.netatmo.com/smart-outdoor-camera.
[28] Nathan Ng et al. 2025. To Offload or Not To Offload: Model-driven Comparison of Edge-native and On-device Processing In the Era of Accelerators. arXiv:2504.15162
[29] NVIDIA. 2017. JETSON TX2: High Performance AI at the Edge. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.
[30] NVIDIA. 2021. A2 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/products/a2/
[31] NVIDIA. 2023. Accelerated Inference for Large Transformer Models Using NVIDIA FasterTransformer. https://github.com/NVIDIA/FasterTransformer/.
[32] NVIDIA. 2023. Jetson Orin Nano Developer Kit. https://developer.nvidia.com/embedded/learn/get-started-jetson-orin-nano-devkit.
[33] NVIDIA. 2024. NVIDIA System Management Interface (nvidia-smi). https://developer.nvidia.com/system-management-interface.
[34] NVIDIA. 2025. Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.
[35] Samuel S. Ogden et al. 2021. PieSlicer: Dynamically Improving Response Time for Cloud-based CNN Inference. In *ICPE'21*.
[36] Samuel S. Ogden et al. 2023. Layercake: Efficient Inference Serving with Cloud and Mobile Resources. In *CCGrid'23*. IEEE.
[37] Hossein Pishro-Nik. 2014. *Introduction to Probability, Statistics, and Random Processes*. Springer.
[38] Vijay Janapa Reddi et al. 2020. MLPerf inference benchmark. In *ISCA'20*. IEEE.
[39] Jinke Ren et al. 2019. An Edge-Computing Based Architecture for Mobile Augmented Reality. *IEEE Network* 33, 4 (2019).
[40] Tom S. 2017. Apple's Neural Engine Infuses the iPhone with AI. https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/
[41] Mahadev Satyanarayanan et al. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (2009).
[42] Mahadev Satyanarayanan et al. 2019. The Seminal Role of Edge-Native Applications. In *EDGE'19*. IEEE.
[43] Rana Shahout et al. 2024. Don't Stop Me Now: Embedding Based Scheduling for LLMs. arXiv:2410.01035
[44] Uma Tadakamalla et al. 2018. FogQN: An Analytic Model for Fog/Cloud Computing. In *UCC Companion'18*.
[45] Bhuvan Urgaonkar et al. 2005. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS'05*. ACM.
[46] Vicuna team. 2024. ShareGPT Dataset Collection. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.
[47] Bin Wang et al. 2024. INVAR: Inversion Aware Resource Provisioning and Workload Scheduling for Edge Computing. In *INFOCOM'24*. IEEE.
[48] Shuochao Yao et al. 2018. FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices. In *SenSys'18*.
[49] Josip Zilic et al. 2025. FRESCO: Fast and Reliable Edge Offloading with Reputation-based Hybrid Smart Contracts. *IEEE Trans. Services Comput.* (2025).