ARISE: High-Capacity AR Offloading Inference Serving via Proactive Scheduling

Z. Jonny Kong*
Purdue University
West Lafayette, USA

Qiang Xu*
Purdue University
West Lafayette, USA

Y. Charlie Hu
Purdue University
West Lafayette, USA

ABSTRACT

With faster wireless networks and server GPUs, offloading high-accuracy but compute-intensive AR tasks implemented in Deep Neural Networks (DNNs) to edge servers offers a promising way to support high-QoE Augmented/Mixed Reality (AR/MR) applications. A cost-effective way for AR app vendors to deploy such edge-assisted AR apps to support a large user base is to use commercial Machine-Learning-as-a-Service (MLaaS) deployed at the edge cloud. To maximize cost-effectiveness, such an MLaaS provider faces a key design challenge, *i.e.,* how to maximize the number of clients concurrently served by each GPU server in its cluster while meeting per-client AR task accuracy SLAs. The above AR offloading inference serving problem differs from generic inference serving or video analytics serving in one fundamental way: due to the use of local tracking which reuses the last server-returned inference result to derive results for the current frame, the offloading frequency and end-to-end latency of each AR client directly affect its AR task accuracy (for all the frames).

In this paper, we present ARISE, a framework that optimizes the edge server capacity in serving edge-assisted AR clients. Our design exploits the intricate interplay between per-client offloading schedule and batched inference on the server via proactively coordinating offloading request streams from different AR clients. Our evaluation using a large set of emulated AR clients and a 10-phone testbed shows that ARISE supports 1.7x–6.9x more clients compared to various baselines while keeping the per-client accuracy within the client-specified accuracy SLAs.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → **Real-time system architecture**.

KEYWORDS

Mobile Augmented Reality, Edge Computing, DNN Offloading, DNN Serving, Machine-Learning-as-a-Service

*Both authors contributed equally to the paper.

MOBISYS '24, June 3–7, 2024, Minato-ku, Tokyo, Japan

ACM ISBN 979-8-4007-0581-6/24/06
https://doi.org/10.1145/3643832.3661894

ACM Reference Format:
Z. Jonny Kong, Qiang Xu, and Y. Charlie Hu. 2024. ARISE: High-Capacity AR Offloading Inference Serving via Proactive Scheduling. In *The 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24), June 3–7, 2024, Minato-ku, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3643832.3661894

1 INTRODUCTION

1.1 Motivation

To deliver a truly immersive and interactive experience to users, Augmented Reality (AR) and Mixed Reality (MR) apps need to perform a number of challenging computer vision tasks to understand and interact with the physical environment, such as object detection [18, 19, 41], depth estimation [45, 71], and odometry [13], at acceptable accuracy. Furthermore, all of the tasks in the AR app performed on each frame are latency-critical; the results for the current frame need to be available in the *current frame interval*, *e.g.,* 16.7 ms at 60 FPS [41], as otherwise they will miss the rendering task for the current frame.

Due to their high accuracy, Deep Neural Network (DNN) models have been increasingly used to support these complex AR tasks (*e.g.,* [16, 22, 29, 30, 51, 52]). However, running state-of-the-art DNN models on commodity mobile devices could take hundreds of milliseconds or even seconds [41, 45, 60]. For example, it takes 254 ms to run the DenseDepth [8] model on a Pixel 7 phone (on the mobile GPU with TensorFlow Lite). While lightweight models like FastDepth [59] and MobileNetV3-SSD [32] (for depth estimation and object detection respectively) are capable of running on-device in real-time, they fall short in accuracy; they achieve 0.35 AbsRel and 0.57 IoU on the datasets used in our evaluation (§6.2), compared to server-grade models such as DenseDepth [8] and ByteTrack [72], which achieve 0.12 AbsRel (lower is better) and 0.90 IoU (higher is better). To this end, offloading, also known as edge-assisted design, has been proposed [10, 23, 41, 44], where camera frames are uploaded to a cloud or edge GPU server for faster DNN inference.

To actually deploy AR apps based on edge-assisted AR design requires effective edge server support for serving potentially a large number of concurrent offloading requests of AR tasks from many mobile clients. Instead of maintaining its own servers, a cost-effective way is for an AR app vendor to use commercial Machine-Learning-as-a-Service (MLaaS) [54, 58], *e.g.,* deployed in the edge cloud (*e.g.,* AWS Wavelength [5], where edge servers are located in the same city and connected to the 5G infrastructure via low-latency links) to serve the AR clients in the vicinity. To effectively manage cost, MLaaS services are usually powered by DNN models under the control of service providers, *e.g.,* one or a few models for each AR task [11, 47]. A primary business objective of an MLaaS provider is to increase the capacity of the cluster (*i.e.,* the number of

AR clients that it can serve concurrently), or provision the cluster size for a given user base in order to maximize its cost-effectiveness.

In such a shared edge-cloud setting, the AR apps running on the mobile clients often exhibit diversity in task accuracy requirements. For example, healthcare apps [31] require high accuracy while tourism apps can get by with moderate accuracy [61]. In such a deployment, the vendor or the users of the AR apps specify an accuracy SLA for each AR task (*e.g.,* object detection) that is required to ensure satisfactory app QoE, and the MLaaS provider tries to maximize the capacity of the GPU cluster in concurrently serving multiple AR clients. In this paper, we focus on optimizing the capacity of individual servers, which is a key challenge and building block in increasing the capacity of a cluster.[1] We formally state the *AR offloading inference serving* problem as: *Given an edge/cloud server cluster serving offloaded inference tasks from AR clients, how to maximize the number of clients supported by individual servers while meeting the per-client AR task accuracy SLAs?*

AR apps often perform and need to offload multiple types of tasks. An efficient way of serving multiple types of AR tasks for a large client base is to divide up the servers in the cluster, such that each server serves one type of AR tasks, since such specialization unlocks more batching opportunities, as shown in previous serving systems [57, 67]. We follow the same design principle and each server only serves one type of AR task.

Difference from video analytics serving. Video analytics serving (*e.g.,* [14, 26, 33, 34, 36, 38, 40, 44, 49, 68, 73]) is similar to AR offloading serving in that one or multiple clients offload a stream of requests to the server for inference. However, there is a subtle but fundamental difference between AR offloading serving and video analytics serving: AR apps desire task results for every frame in order to render virtual objects per frame. When the E2E offloading latency of an offloaded frame is longer than a frame interval due to server inference and frame transfer delay, the AR client resorts to local tracking (detailed in §2) to derive result for the current frame based on the last server-returned result. As a result, the E2E offloading latency directly affects the staleness of the server-returned result which impacts the AR task accuracy for all the frames. *Most of the video analytics systems do not factor in the impact of E2E offloading latency on app accuracy.* While a few works take it into consideration [27, 42], they only support serving a single client.

Difference from generic inferences serving. Compared to generic inference serving (*e.g.,* [9, 20, 21, 24, 37, 50, 55, 57, 62, 67]), AR inference serving faces a significant new complication. Generic inference serving systems assume DNN inference requests are independent, each with its own latency deadline or accuracy constraint. In contrast, in AR offloading serving, the stream of continuous requests offloaded from each AR client are *not unrelated*: due to the use of local tracking, for each client, *the choice of which frame to offload (offloading frequency and timing) and E2E offloading latency of the offloaded inference directly affect the AR task accuracy of all the frames of that client.*

[1]We leave cluster-wide optimizations such as load balancing and dynamic scaling as future work.

1.2 Our Contributions

In this paper, we present a framework that tackles this important AR inference serving problem. We observe that the key design optimization in generic inference serving systems — batched inference — is only exploited *opportunistically*, since such systems assume the inference requests are independent and have no control over request arrivals. The above key difference of AR offloading serving, namely, the AR task accuracy for consecutive frames is also affected by the offloading frequency and E2E delay, also presents a unique design opportunity — *coordinating client offloading schedules to work synergistically with batched inference on the server to maximize the effectiveness of batched server inference while meeting per-client accuracy SLAs.* While some video analytics serving systems also control clients' offloading schedules [34, 38, 40, 68], they do not coordinate client schedules with server inference; coordinating client schedules can boost the effectiveness of batching, as requests within a batch are from different clients.

Exploiting the above design opportunity, i.e., coordinating AR client offloading schedules to maximize the batched server inference, however, faces several challenges: (1) there exists complex relationship between the control knob values (client offloading schedule and server batch size) and the accuracy of *a single client*; (2) there exists interference among the effects of the control knobs *across clients*, as the schedule for any client affects server batching and hence E2E offloading latency (and accuracy) for other clients; (3) modeling per-client accuracy, a pre-requisite for online scheduling, is hard as it is not only affected by the per-client offloading schedule and server batch size, but also frame content, which can change dynamically during the execution of an AR app.

Our proposed AR offloading inference framework ARISE (AR Inference Serving Engine) untangles the intricate interplay among the control knobs across clients via a centralized but scalable scheduler that *proactively* coordinates the offloading schedules of AR clients and the batched inference on the edge server. We first develop a novel lightweight, online accuracy estimator that estimates the AR task accuracy for the current frame for each AR client under different offloading frequency, E2E latency, and dynamically changing frame content. We then design a novel scheduler that decouples deriving per-client offloading schedules and server batching schedule in two steps: (1) it first calculates a pseudo-optimal offloading frequency per-client leveraging the accuracy estimator; (2) it then greedily packs future offloaded requests from all clients into fewer large batches and coordinates client requests accordingly without violating per-client accuracy requirements.

We implemented the ARISE framework on commodity Android phones and GPU servers. Our evaluation using a large set of emulated AR clients and a small-scale testbed of 10 phones show that compared to Clipper-like [21] and Chameleon-like [34] systems, ARISE provides significantly higher capacity in serving AR clients, by up to 5.2x and 6.9x for the depth estimation task and by 1.9x and 2.0x for object detection on an NVIDIA A40 GPU server.

In summary, our main contributions are as follows:

- We present, to our best knowledge, the first framework for serving concurrent edge-assisted AR clients that maximizes the serving capacity of a server while satisfying the accuracy SLAs of originating AR apps.

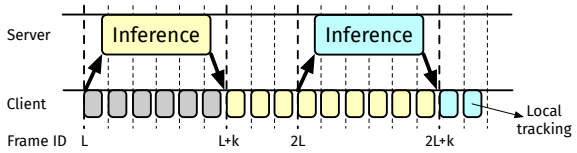


Figure 1: The offloading+local tracking paradigm, with offloading interval of L, and E2E offloading latency of k frame intervals. Frames $L+k$ to $2L+k-1$ reuse the inference result for frame L through local tracking.

- We present an AR inference serving scheduler that proactively coordinates offloading request streams from AR clients to maximize server batching opportunities.
- We present a lightweight AR task accuracy estimator under the commonly used offloading+local tracking based edge-assisted design (§2).
- We implement and experimentally validate our ARISE framework design by comparing it with various baselines including Clipper-like and Chameleon-like systems for two representative AR tasks.

2 BACKGROUND: AR TASK ACCURACY UNDER EDGE-ASSISTED DESIGN

2.1 The Offloading + Local Tracking Paradigm

In edge-assisted AR, even with powerful GPUs, typical DNN inferences still take tens of milliseconds, failing to return the result within the same frame interval. For example, models in Meta's object detection model zoo [2] have a median inference time of 52.5 ms on Tesla V100, much longer than the of 16.7 ms frame interval needed by AR apps running at 60 FPS [41], and the result of an offloaded frame may come back several frame intervals later.

Instead of simply using the last server-return result for the current frame, which was for the last offloaded frame, recent edge-assisted designs [7, 12, 18, 19, 27, 39, 41, 42, 45, 61, 63, 64, 66, 69] have adopted the local tracking technique to generate more accurate results for AR tasks including object detection, human pose estimation, odometry, and more. Specifically, a local tracker runs on mobile device and adjusts the DNN inference results for the last offloaded frame f_l sent back by the server to generate refined results for the current frame f_c, by analyzing the changes between the stale frame f_l and the current frame f_c, as shown in Figure 1. Such local trackers are fast and can typically finish in a fraction of the current frame interval. Local trackers are task-specific and often custom-designed for each type of tasks (*e.g.,* [41, 66]).

2.2 Impact of Tracking Stride on Accuracy

While local tracking improves the accuracy of the result for the current frame f_c (compared to directly reusing the last server-returned result), the gap between its accuracy and that of running the server DNN model directly on f_c (if we could) still widens with *tracking stride*, defined as the frame distance between f_l and f_c, due to increased staleness of the results for frame f_l. For example, in Figure 1, the client offloads every L-th frame, and E2E offloading latency is k. The result for frame L (offloaded at frame interval L) will return at frame interval L+k-1, and be used by local tracking to generate tracked results for frames L+k, …, $2L$+k-1, which would have tracking strides of k (min), k+1, …, L+k-1 (max), respectively.

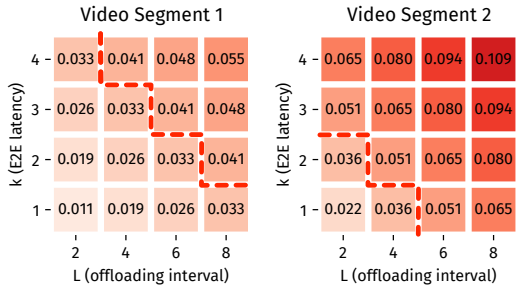


Figure 2: Accuracy drop (our accuracy SLA metric) of offloading the depth estimation task relative to running the DNN model directly, under different (L, k) combinations for two 2-second video segments. With an accuracy SLA of 0.040, only combinations below the red line satisfy the accuracy SLA.

In scenarios where AR task accuracy can be met with less frequent offloading, whether in single-client scenarios (e.g., [19, 41, 42]) or in multi-client scenarios (this paper), reduced offloading frequency (L) saves server and network resources used by the client. Furthermore, in multi-client scenarios, employing batched inference leads to variable E2E offloading latency (k) that is dependent on the batch size. This variability in offloading frequency and E2E latency has three immediate implications on AR task accuracy:

O1: Higher offloading frequency (smaller L) improves AR task accuracy. Offloading more frequently reduces the number of times a stale result is used (L), which improves tracking accuracy.

O2: Lower E2E offloading latency (smaller k) improves AR task accuracy. Lower E2E offloading latency reduces the staleness of last server-returned result used in tracking, which also improves tracking accuracy.

O3: The same accuracy SLA can be achieved by trading off offloading frequency (L) with E2E offloading latency (k). It follows from **O1** and **O2** that AR task accuracy can be improved by reducing either the offloading frequency (L) or the E2E offloading latency (k).

Accuracy measurements in Figure 2 corroborates our observations. We control the offloading of two 2-second video segments (60 FPS) under different offloading frequencies and E2E offloading latencies, and the server runs the DenseDepth [8] model for the depth estimation task. The AR task accuracy improves with lower offloading interval (**O1**) and lower E2E offloading latency (**O2**). It also shows that the same accuracy SLA can be achieved by trading off the two (**O3**). However, the range of satisfactory L and k varies with different video segments (contents), indicating the need for an accuracy estimator that takes frame content into account (§4.3).

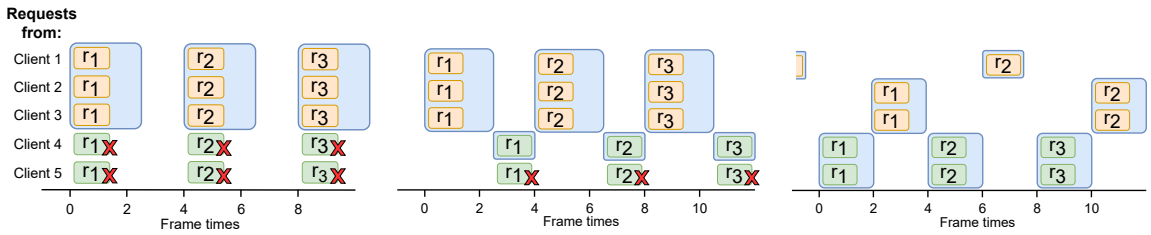
3 DESIGN OPPORTUNITIES AND CHALLENGES

3.1 Design Opportunities

Main idea. Generic inference serving systems maximize the server throughput while satisfying the latency or accuracy constraints

Feasible (k, L) Pairs			
Client 1, 2, 3		Client 4, 5	
k	L	k	L
2.0	8	1.5	6
2.5	4	2.0	4

Inference Latency	
Batch Size	Latency
1	1.5
2	2
3	2.5



(a) Generic inference serving.

(b) Controlling batching and offloading timing.

(c) Controlling batching, offloading timing and frequency.

Figure 4: Controlling offloading schedules (timing and frequency) of AR clients unlocks more batched inference opportunities and enables server to support more clients. The top-left table lists for each client the feasible choices of E2E offloading latency (k) and offloading interval (L) pairs due to accuracy SLA. The bottom-left table gives the maximum batch sizes that can be tolerated by different E2E offloading latencies (k). Each blue box corresponds to one batched inference on the server.

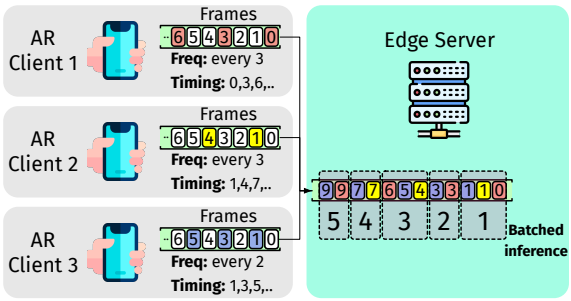


Figure 3: Control knobs in AR offloading serving, including (1) offloading frequency and (2) offloading timing for each client, and the (3) batch size for batched inference on server.

of individual requests. To this end, such systems employ adaptive batched inference as a main optimization technique. However, batched inference in such systems is only exploited *opportunistically*, since such systems assume the inference requests are independent and have no control over request arrivals. An AR client, on the other hand, offloads a stream of requests, and the AR task accuracy for consecutive frames is affected by both the offloading frequency and E2E delay. this key difference presents a unique design opportunity: *the client offloading schedules can be coordinated to work synergistically with batched inference on the server to maximize the effectiveness of batched server inference while meeting per-client AR task accuracy SLAs.*

Control knobs. In generic inference serving, adaptive batched inference is achieved by dynamically tuning one important control knob:

- **Server batch size:** It controls the number of requests to group together and perform DNN inference in a single shot. A larger batch size improves GPU efficiency and thus server capacity. The downside of a larger batch size is longer inference latency, which could negatively impact the application performance or accuracy.

The adaptive batched inference technique also applies to the AR offloading inference problem, where the choice of batch size affects the E2E offloading latency (k).

Additionally, the offloading schedule of individual AR clients can be dynamically adjusted via two client-side knobs (Figure 3):

- **Offloading frequency:** It determines *how many* frames each client offloads in a period of time (L), *e.g.*, the 3 clients in Figure 3 offload every 3rd, 3rd, and 2nd frames, respectively. Higher offloading frequency improves AR task accuracy (**O1**) and allows for relaxed E2E latency and hence a larger server batch size for a given accuracy SLA (**O3**).
- **Offloading timing:** It determines *which* (and hence *when*) frames are offloaded, *e.g.*, client 1 offloads its frames 0, 3, 6, etc., whereas client 2 offloads frames 1, 4, 7, etc.. As we will see below, tuning this knob impacts the grouping of requests into batches, and thus the server capacity.

An example. Figure 4 gives an example on how these additional knobs help unlock more batched inference opportunities and hence improve the server capacity. Under a given accuracy SLA, AR clients can trade off offloading frequency (L) with E2E offloading latency (k) (**O3**). For example, we assume clients 1–3 can either offload once every 8 frames, which allows an E2E latency of 2 frame times (resulting in batch sizes up to 2), or once every 4 frames which allows an E2E latency of 2.5 (batch size 3), while clients 4 and 5 have different tradeoffs due to content difference (§2).

Figure 4a shows one possible trace of generic inference serving where all clients offload requests every 4 frame times. While 5 requests arrive at the sever at the same time, the system can only process at most 3 (restricted by the first 3 clients' E2E offloading latency limit) in one batch, which will finish at time 2.5. By this point it is too late to process the other 2 requests, as they require a maximum E2E latency of 1.5 frame times but have already been queued for 2.5 frame times. As a result, requests from clients 4 and 5 are dropped and the server is unused from time 2.5 to 4.0.

In contrast, in an AR inference serving system, the system can control request arrivals by adjusting each client's offloading timing and frequency. In Figure 4b, by shifting the first request of client 4 to arrive at time 2.5, the server can additionally perform an inference of batch size 1, from time 2.5 to 4.0, increasing server capacity to support 4 clients.

Further, by controlling both request arrival timing and offloading frequency, the server can support all 5 clients, as shown in Figure 4c. By reducing the offloading frequencies of clients 1–3 to once every 8 frames, although this will result in requiring a tighter E2E latency (of 2 frame times) and thus a smaller batch size of 2 for these clients, the request arrival timing of the clients can be coordinated such that requests from all 5 clients can be served under their batch size constraints (and hence their accuracy SLAs).

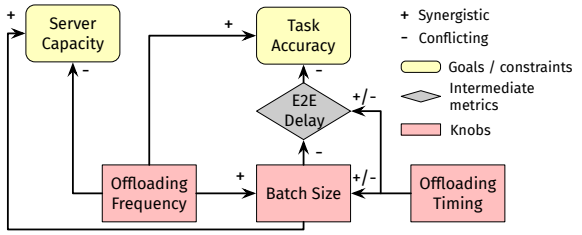


Figure 5: Relationships between control knobs and design goals in AR inference serving.

3.2 Design Challenges

Exploiting the above new design opportunity, *i.e.,* jointly tuning per-client offloading frequency and timing and server batching (control knobs) to maximize the server capacity (objective) while meeting the per-client accuracy SLAs, however, faces several challenges:

C1: Complex relationship between control knob values and design goals for a single client. For each client, tuning the offloading frequency and batch size can affect the server capacity and task accuracy in complex ways, as shown in Figure 5. (1) Increasing the *offloading frequency* improves the task accuracy (**O1**) and the chance of building larger batches which indirectly increases server capacity, but (directly) reduces the server capacity as each client imposes more load on the server. (2) Increasing the *batch size* directly improves the server capacity, but incurs higher queuing and batch inference time, which in turn results in higher per-request E2E delay and thus lower accuracy (**O2**). (3) The reduced accuracy from larger batch size can be compensated by higher offloading frequency (**O3**), which in turn reduces the server capacity. The amount of compensation depends on the relationship between accuracy, E2E delay, and offloading frequency. It is challenging to determine the optimal balance between batch size and offloading frequency.

C2: Intricate interplay among the control knobs across clients. Further, finding the combinations of control knobs for a set of clients becomes even more challenging as the choices for multiple clients can interfere with each other in complex ways. (1) The latency of a chosen batch size that is sufficient for some clients (*e.g.,* with lower accuracy SLAs) may be too long for others (*e.g.,* with higher accuracy SLAs). (2) Under local tracking based AR offloading (§2), interference among offloading requests by different clients, due to incompatible offloading timings, can further elongate the E2E offloading delay and hence lower the client accuracy (Figure 5). Consider the server is serving 10 clients and the minimum batch sizes calculated for the 10 clients in isolation is 5. If the requests from the 10 clients happen to arrive at the same time, the server has to either select a batch size of 5, which meets the accuracy target of the most stringent client, but postpones the rest 5 requests to the next batch, causing *batch-level* queuing delay and violation of accuracy constraint for those clients, or it selects a larger batch size which violates the accuracy target of the most stringent client. In an alternative scenario, if the requests from the 10 clients are evenly spaced out in time, creating a batch of 5 requests will require the first arrival request to wait for 4 more request arrival, causing *intra-batch* queuing delay.

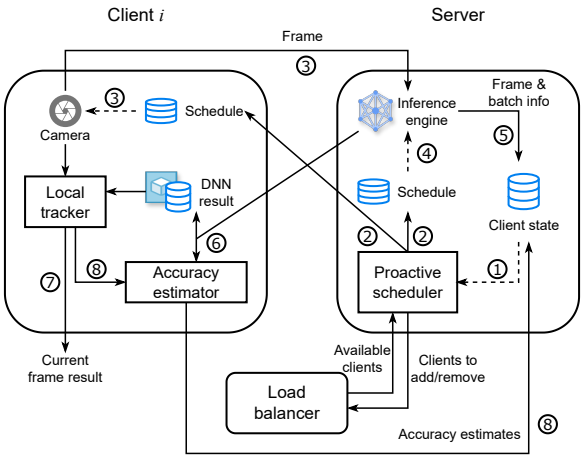


Figure 6: The workflow of ARISE with one of the served clients. The proactive scheduler coordinates the offloading schedules across clients and with batched inference. The accuracy estimator estimates task accuracy by exploiting unique properties of local tracking.

4 ARISE DESIGN

4.1 Design Rationale

ARISE schedules client offloading requests and server batched inferences to maximize the server capacity while satisfying the accuracy SLAs of all clients. One possible approach is letting each client decide on its own offloading schedule in a distributed manner. However, such a distributed approach makes it hard to tackle the intricate interplay among the control knobs across the clients (**C2**). For example, a client experiencing longer E2E request delay may react by increasing its offloading frequency to meet the accuracy SLA, which can lead to higher load on the server and even longer E2E delay. To this end, we propose a centralized scheduler on the server side that proactively coordinates requests from all clients and optimizes server capacity. We also design the scheduler's complexity to be linear in the number of requests, which ensures the scheduler is scalable to support a large number of clients. On the other hand, to ensure all clients meet their accuracy SLAs, an accuracy estimator is needed for each client to work in tandem with the scheduler. We design a lightweight accuracy estimator that runs on each client and sends the accuracy estimates to the server. With the accuracy estimates, the scheduler is able to calculate the accuracy of all clients in a specific scheduling plan and ensure all clients meet their accuracy SLAs.

4.2 Architecture Overview

Figure 6 shows the workflow of ARISE. We envision ARISE will be deployed on the servers in a cluster, where a load balancer is responsible for directing new clients to the servers and help migrate extra clients that exceed the capacity of a server. Each server's scheduler determines whether to take up more clients or to remove clients depending on the resource requirements of the current clients it is serving (evicted clients are redirected by the load balancer to other servers). As we will see shortly, frequent

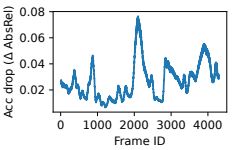


Figure 7: The accuracy drop (relative to reference setup) varies across frames.

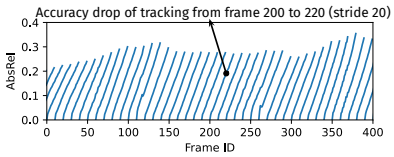


Figure 8: The local tracker accuracy drop (relative to DNN inference) under different tracking strides (from 1 to 30).

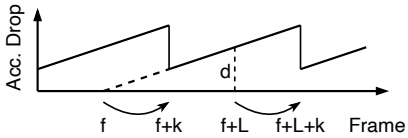


Figure 9: The estimation of accuracy drop d for frame $f+L$ is done by comparing two results for frame $f+L$, one obtained by local tracking on server-returned result for frame f, and one obtained by directly offloading frame $f+L$, whose result comes back at frame $f+L+k$.

scheduler invocation (every 200 ms) allows ARISE to respond to client arrival and departure promptly.

The rest of Figure 6 shows the workflow between the server and one of the served clients. The server consists of a DNN inference engine, a proactive scheduler, a data store for the schedule generated by the scheduler, and a data store for storing client states, including client task accuracy estimates and the timing of recently processed batches and frames. The client is equipped with a camera capturing frames in real time, *e.g.,* at 60 FPS, a stored copy of the latest schedule that dictates the frames to offload, a local tracker, and the accuracy estimator.

Periodically (every 200 ms), the scheduler generates a schedule for the near future, *e.g.,* 1 s, based on the latest client states ❶. The schedule contains IDs of the frames to be offloaded by each client and the grouping of the frames from multiple clients into batches, and is both stored on the server and sent to the clients ❷. The client offloads selected frames ❸, and the server performs batched inference of client requests ❹, both as dictated by the schedule. After inference, the server updates the client state (timing of both the batch and frames in it) for future scheduling ❺, and it sends the inference results back to each client for both local tracking and accuracy estimation ❻. The local tracker is executed for every frame using the latest DNN results and generates result for the current frame for use by upper level AR applications ❼. Finally, the accuracy estimator of each client calculates the current accuracy estimates based on DNN results and local tracking results for the same frame (the one with the last server-returned result), and updates the client states for future scheduling ❽.

4.3 Accuracy Estimation

ARISE expects each client to specify its accuracy SLA according to their application needs. In practice, the accuracy is upper-bounded by what can be achieved in an idealistic offloading scenario where each user is given a dedicated GPU server. Therefore, it is more meaningful for ARISE to meet accuracy drop targets (*e.g.,* within each time window) relative to this idealistic scenario, which we refer to as the *reference setup*. We envision ARISE users will specify their accuracy SLAs as the accuracy drop from the reference setup.

Challenges. As suggested in **O1**, **O2**, and **O3** (§2), a client's task accuracy is directly affected by the offloading frequency and per-request E2E latency. In addition, even under the same offloading frequency and E2E latency, the local tracker's accuracy may vary over time due to changing frame contents. Figure 7 shows the accuracy drop of offloading to a dedicated GPU server under batch size 1 and offloading interval 8 for depth estimation, with the task accuracy measured in absolute relative error (AbsRel). We observe that the accuracy drop varies significantly across frames (0.007–0.077). The detailed setup, including the dataset, DNN model, local tracker, GPU server, and network condition, can be found in §6.1.

Key insights. We make a key observation that all the factors affecting AR task accuracy mentioned above directly affect the staleness of the latest DNN result, which in turn affects the task accuracy, and thus a feature that captures the staleness of the DNN result (*i.e.,* staleness of the frame the result is for) could potentially bridge the gap between the affecting factors and resulting accuracy and be used to develop a lightweight accuracy estimator.

We observe that *tracking stride*, a unique feature in AR offloading (§2), is an ideal candidate to bridge accuracy estimation and impacting factors. First, tracking stride well captures the staleness of the DNN result. The relationship between the affecting factors and tracking stride is straightforward, as shown in observations **O1** and **O2**. Second, tracking stride directly correlates with the accuracy drop. In Figure 8, we plot for a sample video the accuracy drop compared to performing DNN inference on each frame individually. For each line, the x-intercept corresponds to the source frame of the local tracking, while each data point on the line corresponds to the accuracy of local tracking for a different destination frame. We observe that (1) The accuracy drop increases linearly with tracking stride, which enables us to estimate the accuracy drop rate (the slope) and multiply it with the tracking stride to get the accuracy drop under any tracking stride. (2) The accuracy drop rate exhibits temporal locality, *e.g.,* the slopes of the lines for frames between 150 and 250 stay the same. This allows as to approximate the accuracy drop rate of the current frame by estimating that of the last server-returned frame.

Accuracy drop estimator. We start with estimating the accuracy drop rate, as illustrated in Figure 9. Assuming the E2E offloading latency is k frame intervals and the offloading interval is every L frames. Frame f is offloaded, and its result returns at $f+k$ and is used by the local tracker till frame $f+L+k-1$ (including $f+L$). Similarly, frame $f+L$ is offloaded, and its result returns at $f+L+k$. At time $f+L+k$ the client holds both the DNN result and the local tracker result for $f+L$ (tracking result for frame $f+L$ is derived from DNN result for f), and it calculates the accuracy drop d between the two and divides it by the tracking stride $(f+L)-f=L$, which gives the slope f/L as the estimated accuracy drop rate.

We now use the accuracy drop rate to estimate the accuracy drop with regard to the reference setup. For both the schedule being profiled and the reference setup, we first estimate their accuracy drops with regard to offline DNN inference: we first calculate the tracking stride of each frame based on the offloading frequency (L) and E2E offloading latency (k), and then multiply the tracking strides with

the accuracy drop rate to get the accuracy drops compared to offline DNN inference. Next, we calculate the accuracy drop difference between the schedule being profiled and the reference setup, which gives us the accuracy drop with regard to the reference setup.

4.4 Proactive Scheduling

Using the lightweight task accuracy estimator discussed above, the scheduler tries to maximize the number of supported clients while ensuring clients meet their accuracy SLA (expressed as accuracy drop thresholds) by dynamically adjusting the control knobs. However, the intricate interplay among the knobs makes it complicated to directly derive the optimal settings of all control knobs at once. To this end, we first decouple the knobs and generate offloading schedule per-client and serving batching plan in two steps. In Step 1, we derive the pseudo-optimal offloading frequency and batch size for each client, since their relationship with server capacity and task accuracy can be expressed in closed form. In Step 2, we "fine-tune" the generated schedule by greedily packing requests into larger batches and proactively adjusting the request arrivals according to the batch schedule to coordinate client requests and further improve the server capacity.

4.4.1 Step 1: Pseudo-Optimal Offloading Settings. The scheduler first calculates the optimal offloading settings — offloading interval and batch size — for individual clients assuming no queuing delay. Even so, the optimal settings are hard to compute as the optimal settings for a client need to be determined collectively by considering the states of other clients as well. This is because the accuracy for a client depends on both the client's offloading interval and server batch size, while the optimal batch size in turn depends on the optimal settings of other clients in the same batch, *i.e.,* a cyclic dependency. To this end, we propose a heuristic where we first calculate the optimal settings for each client i assuming that all clients have the same accuracy drop rate and SLA as the client of interest. Under this assumption, all clients share the same optimal settings, which can be obtained by solving the following equation:

$$\max_{\text{oi,bs}} \frac{\text{bs}}{\text{lat}_{\text{bs}}} \Big/ \frac{\text{fps}}{\text{oi}} \quad \text{s.t.} \quad \frac{1}{\text{oi}} \sum_{s=\text{e2e}_{\text{bs}}}^{\text{e2e}_{\text{bs}}+\text{oi}-1} \text{dacc}_s - \text{dacc}_{\text{ref}} \le \theta$$

where oi, bs, and θ represent the offloading interval, batch size, and accuracy SLA respectively, lat_{bs} and e2e_{bs} are the inference latency (in seconds) and the end-to-end offloading latency (in frame intervals) under a specific batch size, dacc_s is the accuracy drop under tracking stride s, and dacc_{ref} is the accuracy drop under the reference setup.

The above equation maximizes the number of supported clients (pretending they are identical to client i), which equals to the server inference throughput ($\frac{\text{bs}}{\text{lat}_{\text{bs}}}$) divided by the offloading frequency of each client ($\frac{\text{fps}}{\text{oi}}$), subject to the accuracy SLA, which is calculated as the average accuracy drop across all the frames that rely on tracking the offloaded frame, *i.e.,* with tracking strides e2e_{bs} to e2e_{bs}+oi-1.

4.4.2 Step 2: Greedy Request Packing and Coordination. The optimal schedule calculated per client above assumes no queuing delay and perfect batching. In practice, offloading requests of uncoordinated clients may arrive at the server at any time, disrupting the above schedule. In Step 2, we perform *greedy request packing and coordination* that explicitly coordinates the timing of client request arrivals and batch formation. The algorithm consists of four steps: *simulate*, *adjust*, *verify*, and *apply* (Figure 10), which are described in detail below.

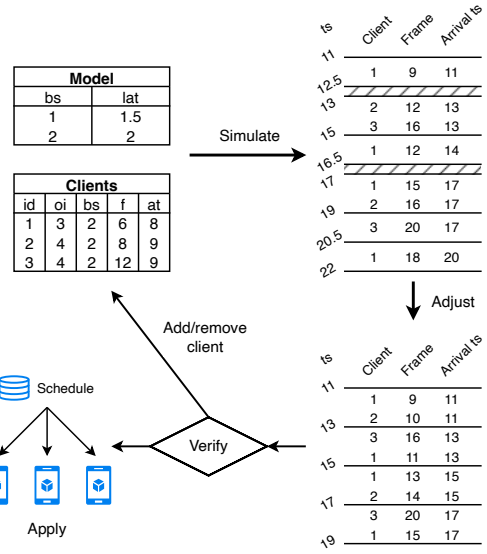


Figure 10: An example of greedy request packing and coordination. The "Model" table gives inference latencies of an example DNN model under different batch size. The "Clients" table gives the current states of the clients, including their ID, the selected offloading interval (oi) and batch size (bs), and the last offloaded frame (f) along with its arrival time (at). All latencies and timestamps are in frame intervals.

Simulate. Before packing and coordinating the client requests, we need to first simulate the request arrivals in the near future based on knob settings calculated in Step 1 and last frame's arrival time for each client. For example, in Figure 10, the last request of client 1 is frame 6, which arrives at time 8, and the offloading interval is 3. Thus, the next request will be frame 9 and should arrive by time 11. As a reference, in Figure 10, the requests are also grouped into batches similarly as how they would have been processed by a general inference serving framework, *i.e.,* opportunistic batching [21]. For example, at time 11, only 1 request is available, and thus a batch of size 1 is formed despite client 1 can tolerate a batch size of 2. On the other hand, at time 17, a batch of size 2 is formed despite 3 requests are available, since all the clients require a maximum batch size of 2.

Adjust. As the core step of greedy packing and coordination, we adjust both the batches and the individual requests. Algorithm 1 shows the adjust algorithm that produces the adjusted scheduling plan. The outer loop (lines 2–11) goes through all the requests in the near future (generated in the *simulate* step) and packs them into batches; the inner loop (lines 5–8) goes through each request in the batch and coordinates them with server batched inference. The algorithm runs in linear time in terms of the number of requests being processed. To ensure the server resources are fully utilized, we regroup the requests into batches greedily without considering the arrival times of the requests (the batch size restrictions still apply). For example, in Figure 10, the first and the second requests are now

Algorithm 1: Request packing and coordination (*adjust* step)

```
input : list of requests R from simulate step in arrival order
        completion time t_0 of current batched inference
output: scheduling plan P
i = 0, t = t_0;
while i < R.length do
    // pack
    bs = argmax_n{r.client.bs ≥ n, ∀r ∈ R[i:i+n]};
    batch = R[i:i+bs];
    // coordinate
    for r in batch do
        r.arrival = t;
        r.frameId = timeToFrameId(r.client, t);
    end
    P.add(batch);
    i += bs, t += InferenceLatency(bs);
end
```

grouped into the same batch that starts at time 11. Next, to resolve any conflicts between requests and minimize the queuing delay, we adjust the expected request arrival time and the frame ID of the requests according to the expected start time of the batches. For example, in Figure 10, client 1 and client 2 requests originally arriving at time 17 are now moved forward to time 15, no longer conflicting with frame 20 from client 3. This adjustment step minimizes both batch-level and intra-batch queuing delays.

Verify. As the adjusted schedule affects offload timing and batching, which in turn affects task accuracy, we next verify the adjusted schedule whether all clients still meet their accuracy SLAs, by going through the adjusted schedule and calculating the accuracy drop of each client using the per-client accuracy estimator. If not, we remove one client from the server, and if yes, we add one more client from the list of available clients (along with their state when running on the previous server) provided by the cluster load balancer. We then repeat the simulate-adjust process until there is no need to remove a client and adding a client is not possible.

Apply. Finally, we apply the final schedule, which grants improved server capacity and meets the accuracy SLAs of all clients at the same time, by sending it to all clients. Both client offloading and server batched inference will follow the new schedule till the next scheduling iteration.

Practical issues. We perform several optimizations to ensure that the scheduler works smoothly in practice. (1) We impose a small overlap between the old and new scheduling plans, *i.e.,* the tail of the old plan is the same as the head of the new plan, which ensures clients can transition smoothly to the new plan without accuracy degradation. (2) Due to network bandwidth fluctuation, requests may not arrive at the exact time in the schedule. Our schedule sets aside a grace period, *e.g.,* 10% of the inference time for every batch to tolerate late requests, which strikes a balance between flexibility of the schedule and wasted server resource. (3) Network bandwidth fluctuation may also make estimating future request arrival time simply based on the last request unreliable. To this end, we perform regression based on recent request arrivals of a client and predict future request arrivals based on the regression model.

As different AR tasks are served by different servers (§1), they are typically scheduled independently for applications that rely on multiple AR tasks. In certain cases, *e.g.,* whether to execute one task depends conditionally on other tasks [33, 57], the scheduler, which executes at a fine granularity (200 ms), can leverage the temporal locality of task results (§4.3) and determine whether to run the task based on the most recent result of the dependency task.

4.5 Other Optimizations

ARISE automatically achieves *pipelining* between network transmission and server inference in the adjust step of proactive scheduling, where clients requests are regrouped to remove gaps between batches and offloaded frames are adjusted so that they arrive right before the batches begin execution. Furthermore, we perform JPEG encoding on both frames and DNN results (if applicable), which is efficient and has minimal impact on accuracy [46].

5 IMPLEMENTATION

We have implemented the ARISE server in about 2K lines of C++ code. We use TensorRT [3] as our DNN inference engine, and perform GPU-accelerated JPEG encoding and decoding using nvJPEG [4]. We implement two client implementations — an Android client and an emulated client. The Android client is implemented in a mix of Java and C++ and runs on the Android phone. The number of Android clients that can run is limited by the phones we have. To evaluate our system for a larger number of AR clients, we implemented an emulated client, written in Python, that emulates the behavior (including the computational latencies) of the Android client but runs on a server. In particular, the emulated client offloads the frames and receives the results like the real client, but emulates the local tracker latency and performs table lookup to get the local tracker accuracy, and hence is light-weight; a single server is able to host hundreds of emulated clients.

6 EVALUATION

6.1 Evaluation Setup

Emulation. We run ARISE and other baselines on a server with an NVIDIA A40 GPU. We first evaluate our system with emulated clients. We emulate the mean and variance of dynamic 5G mmWave (1715 ± 57 Mbps downlink, 152 ± 6 Mbps uplink, 14 ± 2 ms RTT) and LTE (110 ± 17 Mbps downlink, 44 ± 8 Mbps uplink, 32 ± 5 ms RTT) network conditions [28] for each client using the tc tool (the clients' network conditions are independent). We simulate the client arrival following a Poisson process (on average 10 clients per second). Each client randomly selects a video from the dataset (see below) and replays frames in that video. Each experiment lasts 10 minutes. During the experiment, as the clients come and go, the server decides whether to add or remove clients depending on the clients' resource requirements. While in real-world deployment, a load balancer will help migrate the clients between servers in the cluster, for this experiment, we simply set up another server to serve the extra clients that exceed the capacity of the server being measured. The average number of concurrent clients served by the first server is measured and compared against the baselines.

Figure 11: The testbed setup (server is not shown).

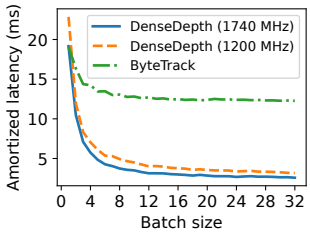


Figure 12: Amortized DNN inference latency.

Testbed. To verify the performance of ARISE against real mobile phones, we next evaluate our system on a small scale testbed setup (Figure 11) that consists of 10 smartphones (7 Google Pixel 5 and 3 Google Pixel 2). All phones are connected to an 802.11ac AP, which connects to the server through a 1 Gbps link. We still emulate the dynamic 5G mmWave network condition on top of it. Upon arrival, a client is assigned to one of the available phones and an instance of the Android client implementation is started on the phone. The phone will be free again when the client session finishes. An arriving client is rejected when all phones are busy. The rest of the setup are the same as above.

6.2 Evaluation Tasks

We test our system with two representative AR tasks separately: depth estimation and object detection. Due to the lack of AR-specific datasets, we follow the practice of recent AR systems [39, 41, 65] and use datasets collected for the target tasks.

Depth estimation. Depth estimation is an essential AR task that estimates the depth map — the distance of each pixel relative to the camera — given an RGB frame. We employ the popular DenseDepth [8] model. Figure 12 (1740 MHz) shows the amortized DNN inference latencies (batch inference latency divided by batch size) under different batch sizes. We use warping [15] as the local tracker. We use a dataset generated by CARLA [25], which contains 20 videos with resolution 448×128 (the input resolution of DenseDepth[2]), each lasting for 70 seconds at 60 FPS with diverse frame content and varying accuracy drop rates. We evaluate the accuracy of the depth maps using the absolute relative error (AbsRel, lower is better) [17].

Object detection. Object detection is another important AR task that helps AR devices to understand the semantics of the surrounding environment. We use ByteTrack [72], a video object detection model that provides smooth object trajectories compared to image object detection models, as our DNN model. Figure 12 shows the amortized DNN inference latencies. We use a Lucas-Kanade based local tracker that estimates new bounding box locations based on the optical flow [1]. We use videos from MOT17 [48] and evaluate object detection accuracy using Intersection over Union (IOU).

6.3 Baselines

We compare ARISE against the following baselines.

Static. This baseline uses the same offloading interval and batch size during the experiment and across all clients. The maximum number of clients it can support and the corresponding configuration values and are determined offline based on pilot experiments that search through the configurations, such that the averaged accuracy across all frames for each client meets its accuracy SLA. The clients are equipped with local trackers.

[2]While AR device cameras usually have high resolution, the camera frames are typically down-sampled to match the input resolution of DNN models [19, 39, 41].

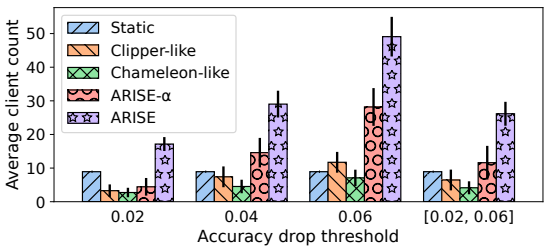


Figure 13: Average number of concurrent clients and the standard deviation under different accuracy SLAs.

Clipper-like (dynamic batching, no dynamic offloading interval). DNN serving systems like Clipper [21] are not directly comparable to ARISE due to their focus on per-request accuracy (which is latency oblivious) or just latency. To this end, we implement a Clipper-like system that is enhanced with local tracking and lightweight accuracy estimation as in ARISE. It performs dynamic batching in the same way as in the simulate step in §4.4. However, all clients offload at a fixed offloading interval (the value is chosen offline in the same way as Static), since requests from the same client are treated independently by such DNN serving systems.

Chameleon-like (dynamic offloading interval, no dynamic batching). We also implement a system that resembles video analytics pipelines like Chameleon [34] but enhanced with local tracking. Chameleon uses profiling-based accuracy estimation which is too compute-intensive to scale, and thus we replace it with the lightweight, online accuracy estimation method in ARISE. The offloading interval of each client is dynamically determined based on the accuracy estimate. However, batching is not employed.

ARISE-α. This baseline takes advantage of most techniques employed by ARISE, including lightweight accuracy estimation, dynamic offloading interval, and dynamic batching. However, the server-side scheduling is done reactively, without the greedy packing in the *adjust* step in §4.4. Essentially, the offloading schedule is the output of the *simulate* step, governed by the *verify* step to ensure that the SLA is met.

Note that the first 3 baselines are "strong baselines" beyond practical algorithms because key offline parameters are chosen based on prior knowledge of the target workload.

Load adjustment for the baselines work as follows. For Clipper-like, Chameleon-like, and ARISE-α, clients are added or removed based on the presence of pending requests, *i.e.*, a fixed number of clients are removed when some requests are delayed till the next batch, which cause them to miss their accuracy SLA, while more clients are added if no requests are delayed within 400 ms. For the Static baseline, the number of clients to support is fixed, as discussed above.

6.4 Main Results

We first evaluate the systems with emulated clients and the A40 GPU on the depth estimation task.

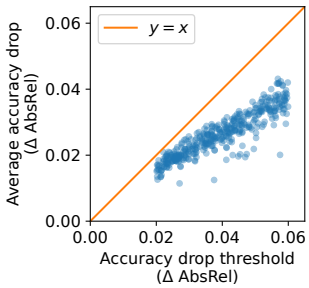


Figure 14: Average accuracy drops vs. accuracy SLAs for ARISE with SLAs in [0.02, 0.06].

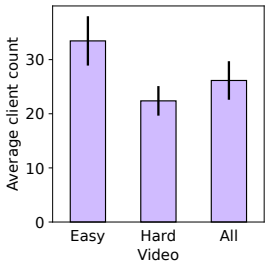


Figure 15: Average client count under different frame difficulties and accuracy SLAs in [0.02, 0.06].

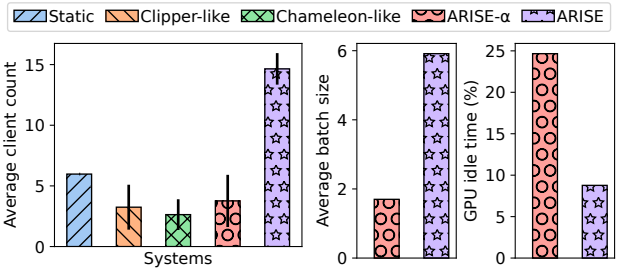
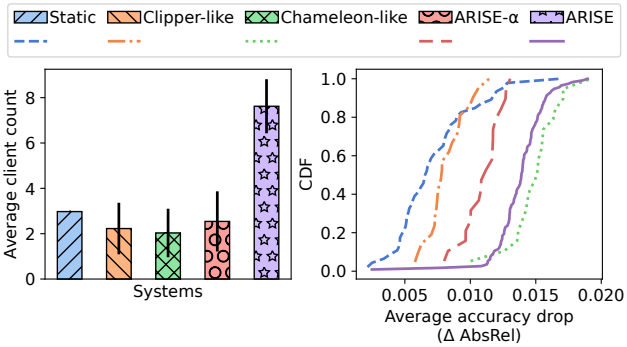


Figure 16: Average client count under LTE at accuracy SLA 0.02.

Figure 18: Average batch size and server idle time of ARISE-α vs. ARISE at accuracy SLA 0.02.



(a) Average number of concurrent clients.

(b) CDF of per-client average accuracy drops.

Figure 17: Testbed results at accuracy SLA 0.02.

First, Figure 13 compares the number of clients supported by different systems under 5G mmWave when all clients have the same accuracy SLAs 0.02, 0.04, 0.06 (SLA values are picked as a fraction of the typical range of the task's accuracy metric), and when the SLAs are drawn randomly from a uniform distribution of range [0.02, 0.06]. In all experiments, Static is configured with fixed offloading interval 3 and batch size 5, while Clipper-like has a fixed offloading interval of 3. The average accuracy for all clients meet the SLAs. For example, in Figure 14, which plots the average accuracy drops vs. the accuracy SLA when ARISE serves clients with accuracy SLA drawn from [0.02, 0.06], all data points are below $y = x$, indicating that ARISE meets the accuracy SLAs of all clients.

We make the following observations about Figure 13. (1) ARISE improves over all baselines under all accuracy SLA choices by 1.7x–6.9x. ARISE improves over ARISE-α by 1.7x–3.9x, indicating the importance of proactive scheduling in resolving the conflicts between requests and increasing the number of clients supported by the server. (2) On the other hand, the improvement of ARISE-α over Clipper-like (by 1.8x–2.4x) and Chameleon-like (by 1.6x–4.0x) shows that tuning both dynamic offloading intervals and batching is important in improving serving performance. (3) Static performs better than other baselines such as ARISE-α under tighter accuracy SLAs. This is because Static's configuration values are selected so that the per-client average accuracy drops are with the SLA, while all other baselines instead strive to ensure that the clients meet their accuracy SLAs at *all times* (by adjusting the number of clients). Thus, while Static allows extra delays caused by uncoordinated requests as long as the per-client average accuracy drops are within the SLA, the dynamic baselines without proactive scheduling try to keep request conflicts minimal, which causes the server to be underutilized, as we will discuss in §6.6. (4) The ratio of average client count between ARISE and other dynamic baselines (Clipper-like, Chameleon-like, and ARISE-α) becomes smaller as the accuracy SLA becomes larger, since the impact of request conflicts becomes smaller as clients can tolerate longer E2E offloading latencies.

Impact of frame content. To study the impact of frame content on ARISE performance, we partition the 20 videos in the CARLA dataset into two groups of 10 videos each, one with high accuracy drop rates (on average 0.007 per frame) and the other with low accuracy drop rates (on average 0.005 per frame). We next run ARISE with only high accuracy drop videos (hard clients), low accuracy drop videos (easy clients), and all videos together, respectively. Figure 15 shows the average number of clients (accuracy SLA [0.02, 0.06]) that ARISE can support. Compared to serving hard clients, ARISE can support 50% more easy ones. In either case, ARISE dynamically adjusts the number of supported clients to ensure all clients meet their accuracy SLAs.

Impact of network condition. Figure 16 shows the average number of clients supported by each system under LTE network condition, which entails longer network latency and different network dynamics. In this experiment, Static is configured with offloading interval 3 and batch size 3 for all clients to meet the accuracy SLA (0.02), and Clipper-like uses fixed offloading interval 3. While all systems support fewer clients compared to running under 5G network condition, ARISE still significantly improves over other baselines and supports 2.5x–5.6x more clients, which demonstrates the robustness of ARISE under different network conditions.

6.5 Testbed Verification

To verify that our framework works on real clients, we next evaluate the systems on the small-scale testbed. As the number of clients supported by some of the baselines exceeds the number of phones we have, which makes it hard to compare between the systems, we restrict the GPU clock frequency to 1200 MHz, and Figure 12 shows the longer batch inference latency compared to no GPU clock frequency limit.

Figure 17 shows the number of clients supported and per-client accuracy drops for different systems under 5G mmWave when all

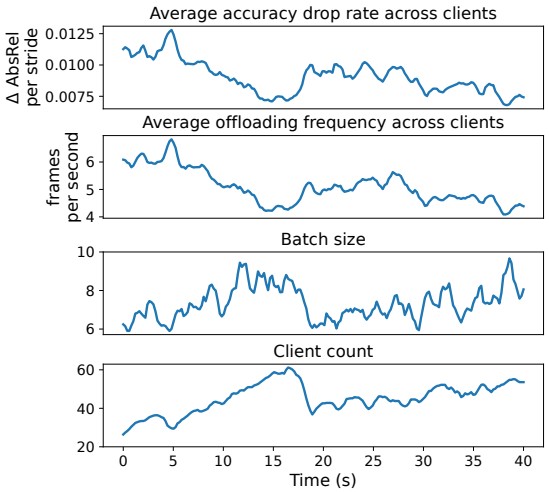


Figure 19: Timeline of server execution stats at accuracy SLA of 0.06.

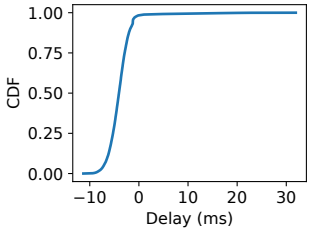


Figure 20: CDF of actual start time minus scheduled start time of the batches at accuracy SLA 0.06.

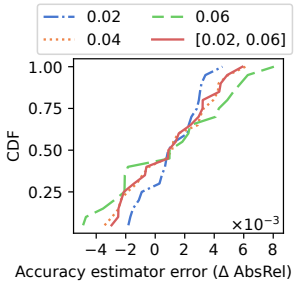


Figure 21: CDF of accuracy estimation error over different videos and under different accuracy SLAs. A positive value means over-estimation.

clients have an accuracy SLA of 0.02. The Static baseline is configured with fixed offloading interval 3 and batch size 3, while Clipper-like has a fixed offloading interval of 3, based on the aforementioned configuration search (§6.3). While all systems ensure that all clients meet their accuracy SLAs, ARISE supports on average 7.6 clients and significantly improves over other baselines by 2.6x–3.8x, validating that the performance gain of ARISE is similar as in the larger-scale emulation experiment. We note that the per-client average accuracy drops of Clipper-like and ARISE-α are only up to 0.011 and 0.013 respectively, which are far from the accuracy SLA of 0.02. This is due to the smaller batch sizes and shorter end-to-end latencies as the server in both baselines try to minimize extra delays caused by request conflicts, as we will discuss in §6.6.

System overhead. The proactive scheduler takes 0.8 ms per execution, which is negligible compared to the scheduler execution interval of every 200 ms. The depth estimation local tracker (warping) takes 3.8 ms on Pixel 5 and 4.0 ms on Pixel 2, while the accuracy estimator takes 2.0 ms on Pixel 5 and 2.7 ms on Pixel 2, both satisfying the real-time requirement at 60 FPS.

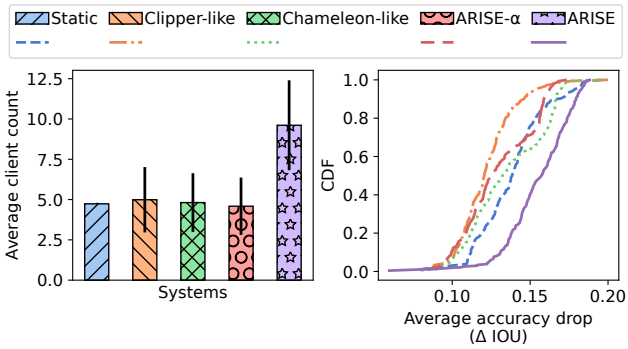
6.6 In-Depth Analysis

Benefits of proactive scheduling. The number of clients that a server can support depends on both client offloading frequencies and the server inference throughput, *i.e.*, the number of requests the server processes in a unit time, which in turn depends on both inference batch sizes and GPU idle times (periods when no DNN inference is performed). While reactive and proactive scheduling require clients to offload at similar frequencies, the server's performance is different due to the uncoordinated requests in reactive scheduling. We plot the average batch size and server idle time of ARISE vs. ARISE-α with accuracy SLA of 0.02 in Figure 18. While both scheduling algorithms select similar batch sizes for the clients, ARISE-α fails to reach the chosen batch sizes. In reactive scheduling, requests arrivals are uncoordinated and may arrive densely at times and sparsely at other times. When they arrive densely, the server will remove some clients to maintain the normal batch size that it can handle. When they arrive sparsely, the server cannot fill up the batch in time (and it cannot add clients quickly enough), and the resulting smaller batch size will lead to shorter end-to-end offloading latency and thus better task accuracy per-client. For the same reason, ARISE-α has a much higher percentage of GPU idle time (24.7%) compared to that of ARISE (8.8%), which is just below the 10% grace period (§4.4).

Server adaptation analysis. To see how ARISE adapts as the client content changes, in Figure 19 we plot the timeline of the average accuracy drop rate across clients, the average offloading frequencies across clients, the average batch size, and the number of concurrent clients with a 1 second moving window. Firstly, the average accuracy drop across clients shows high fluctuation over time, ranging from 0.007 to 0.013, which is due to changes in video content. Secondly, the average offloading frequency closely follows changes of the accuracy drop rate. The batch size is also affected by the accuracy drop rate. For example, the batch size reaches its peak value when the average accuracy drop rate drops to the lowest at around the 40-th second. By controlling both the offloading frequency and batch size accordingly as client accuracy drop rate changes, ARISE is able to dynamically adjust the number of supported clients (bottom figure) so that the number of supported clients is maximized while keeping accuracy drops within SLAs.

Consistency between schedule and runtime. In practice, variations such as network dynamics may cause the runtime behavior to deviate from the generated schedule. We plot the difference between the scheduled start time and the actual start time of each batch in Figure 20. We see that most of the differences are below 0 (but greater than -16.7 ms), meaning that the batches start a little earlier than expected. This happens as the scheduled starting time assumes a buffering time (10% of inference time), but can start right away if all the expected requests for the batch have arrived. On the other hand, only 1.6% of the batches start later than the scheduled time, indicating that the runtime behavior closely follows the schedule by the proactive scheduler.

Accuracy estimation error. In Figure 21, we evaluate ARISE's accuracy estimator by measuring the difference between the estimated and actual average accuracy drops for each video and under different accuracy SLAs. The accuracy estimator shows minimal errors. For example, at accuracy SLA 0.06, the maximum estimation error is 0.008, which is below 15% of the accuracy SLA and



(a) Average number of concurrent clients. **(b) CDF of per-client average accuracy drops.**

Figure 22: Performance comparison on the object detection task with accuracy SLA 0.2.

is sufficient for guiding scheduler decisions. Furthermore, we notice that the accuracy estimator tends to be more accurate under stricter SLAs. At accuracy SLA of 0.02, the estimation error narrows down to between -0.002 and 0.004. This is because clients offload more frequently under tighter SLA, which results in more recent frames being used for accuracy estimation (§4.3), and thus smaller estimation error.

6.7 Evaluation on Object Detection

To evaluate the generalizability of our framework, we evaluate ARISE against the baselines on a second task — object detection with emulated clients. Figure 22 shows the average number of supported clients and the CDF of per-client accuracy drops with all clients having an accuracy SLA of 0.2. The Static baseline is configured with fixed offloading interval 5 and batch size 5, while Clipper-like has a fixed offloading interval of 9. All baselines ensure that all clients meet the accuracy SLA. However, ARISE is able to support 9.6 clients on average, outperforming other baselines by 1.9x–2.1x, which shows the generalizability of ARISE to object detection. Compared to depth estimation, the object detection DNN model has a higher inference latency, and the benefit of batching diminishes faster (Figure 12).

The characteristics of DNN models have an impact on ARISE's tradeoff between offloading frequency and batching. For example, in the object detection experiment in Figure 22, clients have an average offloading frequency of every 13.0 frames, and the average batch size is just 1.2. On the other hand, the depth estimation experiment in Figure 17 supports similar number of clients, but the average offloading frequency is every 4.6 frames and the average batch size is 3.1. ARISE prefers longer offloading interval over larger batch size in the presence of heavy DNN models, which exhibits its ability to dynamically adjust to different task characteristics. The difference in inference latency also explains the relatively higher number of clients supported by Chameleon-like, which does not support batching, compared to that in depth estimation.

7 RELATED WORK

The large amount of prior work on DNN inference serving fall into generic inference serving, video analytics pipelines, or single-AR-client serving.

Generic DNN serving. As discussed in §3.1, the large number of generic DNN serving systems (*e.g.,* [9, 20, 21, 24, 37, 50, 55, 57, 62, 67]) serve requests from multiple clients but do not assume any correlation among requests. Additionally, cluster-level DNN serving optimizations, *e.g.,* auto scaling [67], assignment of requests to servers [37], and assignment of different tasks to servers [20, 37], are complementary to our work, which focuses on optimizing individual servers.

Single-client and multi-client video analytics. Similar to AR offloading serving, video analytics clients also offload a stream of frames and the frames are processed for video analytics tasks, *e.g.,* object detection, on the edge server. A major distinction between video analytics serving and AR offloading serving is that video analytics serving typically have relaxed latency requirements of hundreds of milliseconds, *i.e.,* analytics results on a frame do not need to be available in the current frame interval and are optimized for such latency-oblivious accuracy [14, 40], or are for retrospective analysis [6, 36]. Second, existing video analytics serving systems [14, 34, 35, 38, 40, 44, 49, 68, 73] focus on dynamically adjusting *client-side* offloading configurations to optimize the accuracy under network dynamics or server resource constraints; they do not control *server-side* configurations, *i.e.,* batch size, by coordinating requests on the server. A few video analytic systems incorporate local tracking, but with a focus on a single client [27, 42]; these works can be augmented with ARISE to support multiple clients efficiently.

Single-client AR offloading serving. Many DNN offloading systems have been proposed for a single AR client for tasks such as object detection [18, 19, 41], human pose estimation [41], and depth estimation [45]. Such systems assume a dedicated server is used for each AR client.

Collaborative edge-assisted AR. A few collaborative edge-assisted AR systems optimize AR offloading serving by an edge server [43, 53, 56, 70] by exploiting caching and serving cached results of previously offloaded frames. Such optimizations are applicable when AR clients encounter the same scene and are orthogonal to our design.

8 CONCLUSION

A cost-effective solution to deploying popular edge-assisted AR apps to support a large user base is to use MLaaS to serve offloaded AR inference requests. In this paper, we presented to our knowledge the first framework that addresses the AR inference serving problem. The framework employs an online accuracy estimator that estimates the accuracy for each AR client under various configurations and an online scheduler that proactively coordinate requests from the clients served by a server. Our evaluation using a large set of emulated AR clients and a 10-phone testbed show that ARISE supports 1.7x–6.9x more clients compared to various baselines while keeping the per-client accuracy drops with the client-specified SLA.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Chulhong Min for their helpful comments. This work is supported in part by NSF grants 2112778, 2113893, and 2312834.

REFERENCES

[1] 2020. TensorFlow Android Camera Demo. https://github.com/tensorflow/tensorflow/tree/48a2944c94b190434418d5a7c7f0df452c3aded5/tensorflow/examples/android

[2] 2022. Detectron2. https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md.

[3] NVIDIA Corporation 2024. *NVIDIA TensorRT*. NVIDIA Corporation. https://developer.nvidia.com/tensorrt

[4] NVIDIA Corporation 2024. *nvJPEG Libraries*. NVIDIA Corporation. https://developer.nvidia.com/nvjpeg

[5] Online. *AWS Wavelength - Deliver ultra-low latency applications for 5G devices*. https://aws.amazon.com/wavelength/

[6] Neil Agarwal and Ravi Netravali. 2023. Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics. In *Proc. of USENIX NSDI*.

[7] Adel Ahmadyan and Tingbo Hou. 2020. *Real-Time 3D Object Detection on Mobile Devices with MediaPipe*. Retrieved July 13, 2022 from https://ai.googleblog.com/2020/03/real-time-3d-object-detection-on-mobile.html

[8] Ibraheem Alhashim and Peter Wonka. 2018. High quality monocular depth estimation via transfer learning. *arXiv preprint arXiv:1812.11941* (2018).

[9] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *Proc. of SC*.

[10] Mario Almeida, Stefanos Laskaridis, Abhinav Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D. Lane. 2021. Smart at What Cost? Characterising Mobile Deep Neural Networks in the Wild. In *Proc. of ACM IMC* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 658–672. https://doi.org/10.1145/3487552.3487863

[11] Inc. Amazon Web Services. 2024. *Amazon Rekognition: Automate and lower the cost of your image recognition and video analysis with machine learning*. https://aws.amazon.com/rekognition/

[12] Kittipat Apicharttrisorn et al. 2019. Frugal Following: Power Thrifty Object Detection and Tracking for Mobile Augmented Reality. In *Proc. of ACM SenSys*.

[13] Ali J. Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. 2020. Edge-SLAM: Edge-Assisted Visual Simultaneous Localization and Mapping. (2020).

[14] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *Proc. of USENIX NSDI*.

[15] Jiawang Bian, Zhichao Li, Naiyan Wang, Huangying Zhan, Chunhua Shen, Ming-Ming Cheng, and Ian Reid. 2019. Unsupervised scale-consistent depth and ego-motion learning from monocular video. *Proc. of NeurIPS* 32 (2019).

[16] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint arXiv:2004.10934* (2020).

[17] Cesar Cadena, Yasir Latif, and Ian D. Reid. 2016. Measuring the performance of single image depth estimation methods. In *Proc. of IEEE/RSJ IROS*. 4150–4157. https://doi.org/10.1109/IROS.2016.7759611

[18] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, and Randy H. Katz. 2018. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. In *Proc. of ACM SenSys* (Shenzhen, China) *(SenSys '18)*. Association for Computing Machinery, New York, NY, USA, 292–304. https://doi.org/10.1145/3274783.3274834

[19] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proc. of ACM SenSys*.

[20] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *Proc. of USENIX ATC*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proc. of USENIX NSDI*.

[22] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. 2016. R-FCN: Object detection via region-based fully convolutional networks. In *Proc. of NeurIPS*. 379–387.

[23] Pranab Dash, Z Jonny Kong, Y Charlie Hu, Chris Turner, Dell Wolfensparger, Mun Gi Choi, Abhinav Kshitij, and Viviane E McLandrich. 2023. How to Pipeline Frame Transfer and Server Inference in Edge-assisted AR to Optimize AR Task Accuracy?. In *Proc. of ACM EdgeSys*. 36–41.

[24] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proc. of ACM SoCC*.

[25] A. Dosovitskiy et al. 2017. CARLA: An open urban driving simulator. In *Proc. of CoRL*.

[26] Kuntai Du et al. 2020. Server-driven video streaming for deep learning inference. In *Proc. of ACM SIGCOMM*.

[27] Chengsi Gao, Ying Wang, Weiwei Chen, and Lei Zhang. 2021. An Intelligent Video Processing Architecture for Edge-cloud Video Streaming. In *Proc. of ACM/IEEE DAC*. 415–420. https://doi.org/10.1109/DAC18074.2021.9586328

[28] Moinak Ghoshal et al. 2022. Can 5G mmWave support Multi-User AR?. In *Proc. of PAM*.

[29] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. of IEEE CVPR*. 580–587.

[30] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proc. of IEEE/CVF ICCV*. 2961–2969.

[31] Florian Heinrich, Luisa Schwenderling, Fabian Joeres, Kai Lawonn, and Christian Hansen. 2020. Comparison of Augmented Reality Display Techniques to Support Medical Needle Insertion. *Proc. of IEEE TVCG* 26, 12 (2020), 3568–3575.

[32] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proc. of IEEE/CVF ICCV*. 1314–1324.

[33] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A Cost-Effective Deep Learning Inference System. In *Proc. of ACM SoCC*.

[34] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: Scalable Adaptation of Video Analytics. In *Proc. of ACM SIGCOMM*.

[35] Shiqi Jiang, Zhiqi Lin, Yuanchun Li, Yuanchao Shu, and Yunxin Liu. 2021. Flexible High-Resolution Object Detection on Edge Devices with Tunable Latency. In *Proc. of ACM MobiCom* (New Orleans, Louisiana) *(MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3447993.3483274

[36] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. Noscope: Optimizing Neural Network Queries over Video at Scale. *Proc. of VLDB* (2017).

[37] Liu Ke, Udit Gupta, Mark Hempsteadis, Carole-Jean Wu, Hsien-Hsin S. Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. In *Proc. of IEEE HPCA*.

[38] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. 2023. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *Proc. of USENIX NSDI*.

[39] Z. Jonny Kong, Qiang Xu, Jiayi Meng, and Y. Charlie Hu. 2023. AccuMO: Accuracy-Centric Multitask Offloading in Edge-Assisted Mobile Augmented Reality. In *Proc. of ACM MobiCom*.

[40] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proc. of ACM SIGCOMM*. 359–376.

[41] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *Proc. of ACM MobiCom*.

[42] Ruoyang Liu, Lu Zhang, Jingyu Wang, Huazhong Yang, and Yongpan Liu. 2021. PETRI: Reducing Bandwidth Requirement in Smart Surveillance by Edge-Cloud Collaborative Adaptive Frame Clustering and Pipelined Bidirectional Tracking. In *Proc. of ACM/IEEE DAC*.

[43] Zida Liu, Guohao Lan, Jovan Stojkovic, Yunfan Zhang, Carlee Joe-Wong, and Maria Gorlatova. 2020. CollabAR: Edge-assisted Collaborative Image Recognition for Mobile Augmented Reality. In *Proc. of ACM/IEEE IPSN*.

[44] Jiachen Mao et al. 2019. MobiEye: An Efficient Cloud-Based Video Detection System for Real-Time Mobile Applications. In *Proc. of ACM/IEEE DAC*.

[45] Jiayi Meng, Zhaoning Kong, Qiang Xu, and Y Charlie Hu. 2021. Do Larger (More Accurate) Deep Neural Network Models Help in Edge-assisted Augmented Reality?. In *Proc. of ACM SIGCOMM NAI*.

[46] Jiayi Meng, Z. Jonny Kong, Y Charlie Hu, Mun Gi Choi, and Dhananjay Lal. 2022. Do We Need Sophisticated System Design for Edge-assisted Augented Reality?. In *Proc. of ACM EdgeSys*.

[47] Microsoft. 2024. *Azure AI Vision*. https://azure.microsoft.com/en-us/products/ai-services/ai-vision/

[48] Anton Milan et al. 2016. MOT16: A Benchmark for Multi-Object Tracking. *arXiv preprint arXiv:1603.00831* (2016).

[49] Sibendu Paul, Kunal Rao, Giuseppe Coviello, Murugan Sankaradas, Oliver Po, Y. Charlie Hu, , and Srimat T. Chakradhar. 2022. Enhancing Video Analytics Accuracy via Real-time Automated Camera Parameter Tuning. In *Proc. of ACM SenSys*.

[50] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. 2019. Swift Machine Learning Model Serving Scheduling: A Region Based Reinforcement Learning Approach. In *Proc. of SC*.

[51] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proc. of IEEE CVPR*. 7263–7271.

[52] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[53] Pei Ren et al. 2020. Edge AR X5: An Edge-Assisted Multi-User Collaborative Framework for Mobile Web Augmented Reality in 5G and Beyond. *IEEE Transactions on Cloud Computing* (2020), 1–1.

[54] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. 2015. MLaaS: Machine Learning as a Service. In *Proc. of IEEE ICMLA*.
[55] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proc. of USENIX ATC*.
[56] Pravin Shankar, Tamer Nadeem, Justinian Rosca, and Liviu Iftode. 2008. CARS: Context-Aware Rate Selection for Vehicular Networks. In *Proc. of IEEE ICNP*.
[57] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proc. of USENIX SOSP*.
[58] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proc. of USENIX NSDI*.
[59] Diana Wofk, Fangchang Ma, Tien-Ju Yang, Sertac Karaman, and Vivienne Sze. 2019. FastDepth: Fast Monocular Depth Estimation on Embedded Systems. In *Proc. of ICRA*.
[60] Ran Xu, Jayoung Lee, Pengcheng Wang, Saurabh Bagchi, Yin Li, and Somali Chaterji. 2022. LiteReconfig: Cost and Content Aware Reconfiguration of Video Object Detection Systems for Mobile GPUs. In *Proc. of EuroSys* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 334–351. https://doi.org/10.1145/3492321.3519577
[61] Ran Xu, Chen-lin Zhang, Pengcheng Wang, Jayoung Lee, Subrata Mitra, Somali Chaterji, Yin Li, and Saurabh Bagchi. 2020. ApproxDet: Content and Contention-Aware Approximate Object Detection for Mobiles. In *Proc. of ACM SenSys* (Virtual Event, Japan) *(SenSys '20)*. Association for Computing Machinery, New York, NY, USA, 449–462. https://doi.org/10.1145/3384419.3431159
[62] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. 2016. SERF: Efficient Scheduling for Fast Deep Neural Network Serving via Judicious Parallelism. In *Proc. of SC*.
[63] Chun-Han Yao, Chen Fang, Xiaohui Shen, Yangyue Wan, and Ming-Hsuan Yang. 2020. Video Object Detection via Object-Level Temporal Aggregation. In *Proc. of ECCV*, Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm (Eds.). Springer International Publishing, Cham, 160–177.
[64] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. 2020. NEMO: Enabling Neural-Enhanced Video Streaming on Commodity Mobile Devices. In *Proc. of ACM MobiCom* (London, United Kingdom) *(MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, Article 28, 14 pages. https://doi.org/10.1145/3372224.3419185
[65] Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications. In *Proc. of ACM MobiCom*.
[66] M. Yong. 2019. *Object Detection and Tracking using MediaPipe.* https://developers.googleblog.com/2019/12/object-detection-and-tracking-using-mediapipe.html
[67] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. of USENIX ATC*.
[68] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *Proc. of USENIX NSDI*.
[69] Jinrui Zhang, Deyu Zhang, Xiaohui Xu, Fucheng Jia, Yunxin Liu, Xuanzhe Liu, Ju Ren, and Yaoxue Zhang. 2020. MobiPose: Real-Time Multi-Person Pose Estimation on Mobile Devices. In *Proc. of ACM SenSys* (Virtual Event, Japan) *(SenSys '20)*. Association for Computing Machinery, New York, NY, USA, 136–149. https://doi.org/10.1145/3384419.3430726
[70] Wenxiao Zhang, Bo Han, and Pan Hui. 2022. SEAR: Scaling Experiences in Multi-user Augmented Reality. *Proc. of IEEE TVCG* 28, 5 (2022), 1982–1992.
[71] Yunfan Zhang, Tim Scargill, Ashutosh Vaishnav, Gopika Premsankar, Mario Di Francesco, and Maria Gorlatova. 2022. InDepth: Real-Time Depth Inpainting for Mobile Augmented Reality. In *Proc. ACM IMWUT*.
[72] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. 2022. ByteTrack: Multi-object Tracking by Associating Every Detection Box. In *Proc. of ECCV*.
[73] Zhang, Wuyang and He, Zhezhi and Liu, Luyang and Jia, Zhenhua and Liu, Yunxin and Gruteser, Marco and Raychaudhuri, Dipankar and Zhang, Yanyong. 2021. ELF: Accelerate High-resolution Mobile Deep Vision with Content-aware Parallel Offloading. In *Proc. of ACM MobiCom*.