OVIDA: Orchestrator for Video Analytics on Disaggregated Architecture

Manavjeet Singh[1], Sri Pramodh Rachuri[1], Bryan Bo Cao[1], Abhinav Sharma[1], Venkata Bhumireddy[1], Francesco Bronzino[2], Samir R. Das[1], Anshul Gandhi[1], Shubham Jain[1]
[1] *Stony Brook University, USA*, [2]*École Normale Supérieure de Lyon, France*

***Abstract*—Millions of video cameras are deployed globally across major cities for learning-based video analytic (VA) applications, such as object detection. Video streams from the cameras are either sent over the wide-area network to be processed by the cloud or are (at least partially) processed in a local edge workstation, incurring significant latency and elevated financial costs. In this paper, to minimize reliance on the cloud and overcome the unavailability of high-compute workstations on edge, we investigate the use of heterogeneous and distributed embedded devices as edge nodes shared by multiple cameras to fully serve the video processing needs of a VA application (without requiring cloud support).**

We present OVIDA, an edge-only orchestrator to deploy VA application(s) on a distributed edge environment to maximize accuracy. Given the resource-constrained nature of edge nodes, OVIDA disaggregates the VA application pipeline into multiple modules. OVIDA's core functionality and contributions are: (i) optimizing the placement and replication of the VA application modules across the edge nodes to maximize the throughput, and in turn, accuracy; and (ii) an adaptive model selection algorithm for VA modules based on accuracy-throughput trade-off to maximize accuracy in response to varying load conditions. To further improve performance, OVIDA employs a central-queue–based design (instead of the usual push-based design), which also obviates the need for complex load balancing algorithms. We implement OVIDA on top of Kubernetes and evaluate its performance for three VA applications, supported over a heterogeneous edge cluster under varying network conditions. When compared against several baselines in our evaluation, we achieve throughput and accuracy gains of at least 51% and 28%.

***Index Terms*—Video Analytics, Edge-only Deployment, Disaggregation, Module Placement, Model Selection**

I. Introduction

A significant rise in the deployment of cameras to support diverse applications in surveillance, health-care, transportation, and safety has led to the wide adoption of video analytics systems [1]. These systems use advances in computer vision and machine learning, fueled by the growth in edge and cloud computing paradigms. The global video analytics market is estimated to grow from $5 billion in 2020 to $21 billion by 2027, at a CAGR of 22.70% [2]. This level of growth necessitates efficient systems that can perform live Video Analytics (VA).

Traditionally, cloud computing has been favored due to the intensive computational requirements posed by VA applications using Deep Neural Networks (DNNs). However, this computatonal capability comes at a cost of significant latency, elevated financial costs, and privacy issues [3]–[5]. For example, there are 15,576 CCTV cameras in London Underground train stations; enabling live streams of these to the cloud would require significant capital and operational expenditures [6]. In contrast, the edge computing paradigm offers reduced latency and enhanced privacy by bringing data processing closer to the source [4], [7], albeit with weaker compute nodes. Edge computing is also beneficial for remote cameras that may not have a high bandwidth connection to the cloud and in settings where using the cloud is not economical [3].

Unfortunately, edge devices, such as Jetson Nanos, Jetson Orins [8] and Raspberry Pis [9], are significantly less capable than cloud servers in terms of processing power, memory size, and connectivity. Since DNNs have high memory and GPU requirements, a single edge node may not have sufficient resources to execute an entire VA application with satisfactory performance; in the VA context, performance is typically measured in terms of accuracy or throughput. In response, recent works have suggested breaking (i.e., disaggregating) the VA pipeline into its individual constituent functions (referred to as 'modules' in this paper) and deploying them as microservices (or 'services') on multiple edge nodes [10]–[13].

However, disaggregating VA pipelines across the edge poses four key challenges. (i) *Reduced data locality*: The input and output data sizes of VA modules in a pipeline can vary significantly, resulting in severe data transfer delays over diverse edge networks if modules are not placed appropriately. (ii) *Heterogeneity*: Modules may have different resource requirements (e.g., memory, GPU), and can have variable performance on heterogeneous edge nodes. (iii) *Underutilization of resources*: Due to disaggregation, each module has to spend a significant amount of processing time in I/O operations, which results in the underutilization of GPU resources. (iv) *Large model space*: Often, there are multiple models that can be employed to perform a single VA task on a specific compute node. These models, however, can vary in resource consumption, service rates, and content-specific accuracy. Due to their differences, the models might react differently to real-world dynamism in arrival rate and frame content, making selecting the appropriate model an important challenge.

These challenges necessitate intelligent resource allocation, placement policies, and efficient model selection for a distributed edge. Unfortunately, existing solutions are either not optimized for multi-function pipelines [14], [15], assume that there will only be one neural network-based VA-application

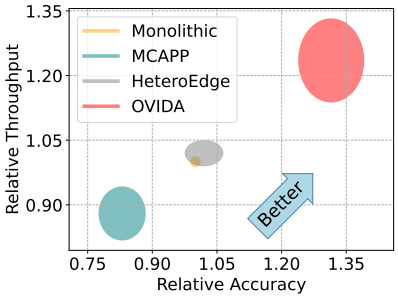
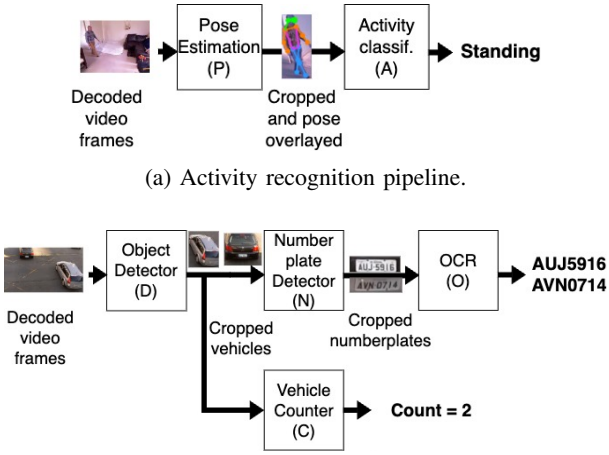


Fig. 1: A summary of accuracy and throughput performance results, comparing OVIDA with other placement policies, normalized against Monolithic. The center of each ellipse represents the median, while the radius in each direction indicates the standard deviation.



(a) Activity recognition pipeline.

(b) Number plate recognition and vehicle counting pipelines; object detector is a common function for these two pipelines.

Fig. 2: Illustration of the three VA pipelines implemented and evaluated in this paper.

service [11], or are not able to optimally use the resources on an edge node [16]. Several works offer techniques to select the optimal model based on the given constraints [14], [17]–[20]. Most of them depend on the inference of the "golden configuration" – the best possible configuration. Such an approach requires all or most models to be loaded into the memory simultaneously, which is infeasible in embedded devices such as smart cameras due to memory limitations.

This paper presents the design and implementation of a distributed framework, OVIDA (**O**rchestrator for **VI**deo analytics on **D**isaggregated **A**rchitecture), for efficiently deploying disaggregated VA pipelines over network-connected heterogeneous edge nodes to maximize the throughput and accuracy of the VA application. OVIDA also features a lightweight pseudo-central queue-based mechanism that minimizes queuing delays, reduces per-frame latency, and obviates the need for complex load balancing in heterogeneous settings. OVIDA is enabled with a lightweight solution to dynamically choose and switch models on edge devices in response to changes in arrival rate and frame content.

OVIDA presents a complete deployment solution for distributed VA on the edge. We disaggregate the VA pipelines into stateless modules and make the following contributions:

- *Replication and Placement:* OVIDA performs bottleneck function detection and horizontally scales bottleneck modules by deploying multiple instances. These instances are placed across edge nodes to maximize the end-to-end throughput (proportional to accuracy) of the VA pipeline, while optimizing resource utilization at each node.
- *Adaptive model selection:* OVIDA continuously estimates the best DNN model based on load–accuracy trade-off, leading to the best possible accuracy, without requiring to search through the entire space every time.
- *Load balancing:* We use insights from queuing theory to implement a lightweight central queue-based mechanism that minimizes queuing delays (leading to a lower per-frame latency) and obviates the need for complex load balancing.

We implement OVIDA over Kubernetes [21] using Docker [22] to containerize each module instance. We demonstrate the performance of OVIDA for three VA application pipelines, supported over a heterogeneous edge cluster under varying network conditions. We also compare OVIDA's performance against several baselines, including those from recent works [11], [16]. Based on experimental evaluations, we show that OVIDA's placement achieves up to 41% improvement in accuracy compared to the next-best placement policy. Figure 1 shows the accuracy and throughput of OVIDA compared to other baseline policies across various deployment scenarios (see Section VII-A for experimental details). Combining OVIDA's placement and dynamic model selection further increases the accuracy gains by 15%.

II. Background and Motivation

This section provides essential background on video analytics, edge computing, and the availability of diverse models for VA applications. It also motivates the need to maximize resource utilization of edge nodes and implement input-specific dynamic model switching to enhance accuracy.

Video Analytics (VA) refers to analyzing video content using computer vision techniques. A VA application typically consists of several vision functions, chained together to form a pipeline, where the output of one function is the input to the next. The VA pipelines used in this paper for the evalution study are Activity Recognition (AcRg), Number Plate Recognition (NPR), and Vehicle Counting (VC), as illustrated in Figure 2. Note that NPR and VC have a common function – the object detector. The functions in VA pipelines may differ significantly in resources required (e.g., CPU, GPU, and memory) and their input/output data sizes.

Edge Computing aims to reduce the bandwidth and latency requirements when compared to the cloud by using computing resources near the data source [5]. VA applications are excellent candidates for edge computing, as it eliminates the need to send large video content over the network to remote

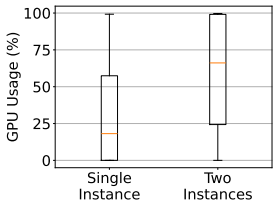


Fig. 3: Improved GPU utilization obtained by placing multiple bottleneck modules on an edge device.

cloud servers. Moreover, processing data on hardware that is already owned and paid for by stakeholders such as cities, service providers, or the users, gives an economic and privacy incentive to use edge computing. However, edge-only systems do have some drawbacks when compared to a cloud data center; they have limited resource availability (e.g., memory and compute) and are typically heterogeneous in terms of computing and connectivity.

Since edge nodes are underpowered and have limited resources, it is essential to maximize their utilization. We conduct an experiment to investigate the opportunity to host multiple functions on a single edge node. Our key insight is to leverage the time spent on I/O operations by each function. As can be seen in Figure 3, the GPU is significantly underutilized when only a single instance is supported. This is due to high I/O overhead in a disaggregated pipeline—frame transfers over the network and unavoidable hardware limitations such as copying an image to GPU memory before it can be processed. However, when two module instances (or replicas) of the bottleneck service are placed together, they work in parallel and share the GPU, resulting in much higher ($\sim 3.6\times$) GPU utilization due to interleaving. Further, deploying two instances of the bottleneck module on the same device under the same load nearly *doubles* the achieved throughput. Thus, multiplexing DNN-based modules over a GPU-enabled embedded edge device can improve resource utilization and overall pipeline throughput and accuracy. Prior works, however, do not exploit this feature and only deploy a single pipeline module per application on an edge device [16].

Diverse model implementations. A given VA task can be serviced at varying levels of performance by different model implementations. In particular, the computer vision community offers a variety of DNN models with different levels of accuracy, inference times, and resource consumption for performing the same tasks. For example, models like YOLO [23], which are used for detection, pose estimation, classification and other applications are available in multiple sizes (e.g., nano, small, and medium) and differ in their accuracy and resource consumption because of differences in their number of model parameters. At low load, a simpler and faster model (such as a nano variant) might perform worse than a more powerful model (such as a medium variant). However, as load increases, for example, due to additional video streams, the medium model variant will drop more frames due to slower inference times. Subsequently, it may end up performing worse than the nano model, which may not have to drop any frames due to its lightweight needs.

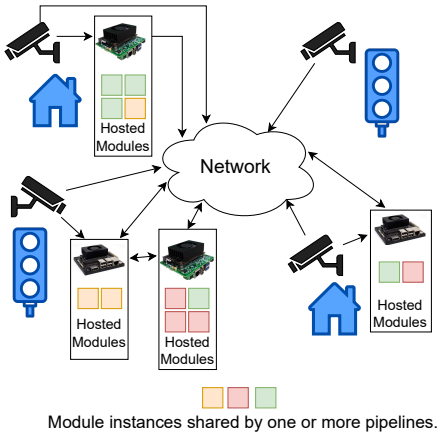


Fig. 4: Overview of distributed edge devices featuring multiple VA pipelines. Each colored box represents a VA module hosted on the edge device. The modules are replicated as needed.

Prior works have proposed techniques to select the best possible model to perform a VA task in response to changes in input [18] and required quality of service [14], [17] (see Section VIII for a discussion of related works). All of these require all models to be pre-loaded into the memory for a periodic comparison. Unfortunately, this is not possible in memory-constrained embedded devices.

III. System Design and Implementation

This section discusses the fundamental system design decisions behind OVIDA, including the disaggregation of the VA pipeline (Section III-A) and the central queue-like architecture (Section III-B). We discuss the key contributions of OVIDA— module instance placement and model selection—in Sections IV and V, respectively.

A. Microservice-based distributed pipeline

The functions in a VA pipeline can either all be placed on a single device in a monolithic fashion or across multiple devices in a distributed fashion. A monolithic deployment has the following major drawbacks, especially when deployed on the edge: (i) Each edge node may lack sufficient memory to host all required function models (note that our choice of pipelines and edge nodes still allows us to evaluate monolithic deployment in Section VII); (ii) New frames start processing only after the current input has been fully processed through the pipeline; (iii) The architecture cannot scale effectively with the bottleneck function of the pipeline.

To avoid the limitations of a monolithic deployment on the edge, we design a *distributed pipeline* that splits a VA application into multiple functions, where each function works independently. OVIDA packages each VA pipeline function as a microservice (or service), and each service may include one or more replicas of containerized functions called 'modules' (see Figure 5b). Module instances (or replica) can be distributed across network-connected nodes as needed, avoiding resource

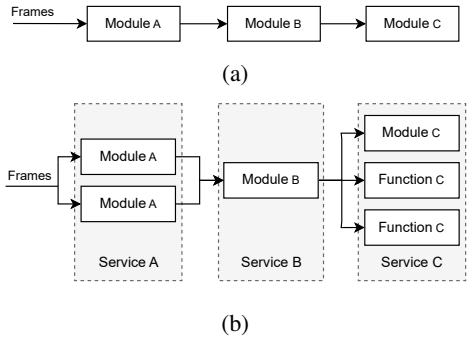


Fig. 5: Example illustrations of (a) a pipeline with 3 modules and with one instance each, and (b) the same pipeline with 3 modules but with multiple instances for modules A and C.

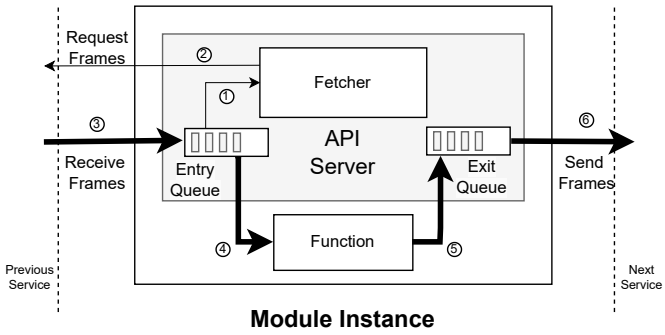


Fig. 6: Module instance in pseudo-central queue design.

limitations (e.g., memory availability, illustrated in Figure 4). The distributed design also provides another key benefit—*scalability*. Since edge devices are resource-constrained, the throughput of a specific module may be limited; by horizontally scaling a module using multiple instances, we alleviate bottlenecks. In contrast, a monolithic deployment requires replicating the entire pipeline, not just the bottlenecked modules. However, a drawback of disaggregating the pipeline is the *additional I/O costs of transferring frames between the modules*. With careful module placement (Section IV), our evaluation in Section VII shows that this cost can be minimized, and is significantly outweighed by its benefits.

Leveraging the horizontally scaled instances of a module requires careful resource allocation among them to avoid processing and queueing delays. Each module instance can have varying throughput based on the compute power of the edge node, the other modules sharing the node, and the current input load. Thus the resource availability at an edge node can change dynamically and abruptly. We next discuss how OVIDA's architecture design is inherently tailored to handle such dynamic scenarios.

B. Pseudo-central queue design

In general, distributed applications employ a "push" based architecture, where input data, such as frames or objects, are pushed from the source to the servers (or module instances, as in our case); such push-based architectures have been employed in recent VA works, such as Distream [12], VideoEdge [24], and Hetero-Edge [11], among others. This architecture works well if there is only one instance (as shown in Figure 5a) or if each of the multiple instances of a module has the same processing rate. However, this is not true in heterogeneous edge-based VA systems where different instances may have different resource capacities and availability (due to CPU and GPU sharing), and thus different processing rates. Consequently, an instance with a lower processing rate must either maintain a possibly ever-increasing queue or drop some frames. On the other hand, an instance with a higher processing rate might idle.

To address the issue of heterogeneous processing rates, OVIDA employs a *central queue implementation*, which is pull-based instead of push-based. Under this implementation, an instance pulls frames to process only when it is ready to process them, i.e., once it has completed processing the previous frames. Queueing literature informs us that a central queue-like policy (single queue, multiple servers, such as $M/M/k$) provides significantly lower frame processing time than a round-robin, push-based policy [25, Chapter 14.4]. This is because, with a push-based policy, incoming jobs (or frames) can get stuck behind a slow job at an instance; this is not the case with a pull-based central queue.

An idealized central queue concept assumes that there is no communication delay between the central queue and the module instances. In real edge deployments, however, network link delays are not negligible. To minimize these delays in practice, OVIDA makes use of buffers that we call entry and exit queues, as explained below.

Design details: OVIDA amortizes data transfer delays by pulling multiple frames (equal to the buffer size of the entry queue) from the central queue to an idle instance. We refer to this architecture with the buffers as a *pseudo-central* queue. Figure 6 shows the major components of OVIDA's pseudo-central queue design.

- *Entry and Exit Queue* are the shared data structures that serve as buffers among API Server, Fetcher, and Function.
- *API Server* runs an HTTP server and provides API for receiving frames and handling frame requests. The received frames are stored in the entry queue ③. The requested frames are sent from the exit queue ⑥.
- *Fetcher* runs on a separate thread inside the API Server. If the entry queue is not filled ① and has n empty buffer spaces, it requests n frames to be sent from a module instance of the previous service (using the latter's Server Process API) ②. The Fetcher keeps requesting frames from instances of the previous service (see Figure 5b) in a round-robin fashion until the entry queue is full.
- *Function* runs the main task of processing a frame. It pops the unprocessed frame from the entry queue ④, processes it, and pushes it into the exit queue ⑤. If the exit queue is filled up to a set limit (generally shallow), the worker process will stop popping from the entry queue and cease the Fetcher. For the first module in the pipleline, if the entry queue is full, incoming frames from the frame decoder are

dropped until there is space available in the entry queue.

We evaluate the impact of the pseudo-central queue on a distributed and heterogeneous deployment of a Number Plate Recognition pipeline (more in Section VII) and find that the pseudo-central queue design achieves as much as 4× lower latency compared to push-based under high loads. As such, we employ the pseudo-central queue in our implementation of OVIDA (and all baselines, see Section VI-C).

IV. Placement and Replication

In general, the throughput of a VA pipeline can be increased by: (1) using faster edge nodes, (2) optimizing the utilization of the available edge nodes, or (3) employing a faster variant of the DNN used in a module. Unlike the cloud, upgrading to a faster edge node on the fly (option 1 above) is not practical due to the limited availability of edge devices. In this work we explore the other two options. We first discuss option 2 in this section. This is aimed at improving the utilization of available resources via careful model placement and replication. In Section V, we will follow up with option 3—dynamic switching between faster-but-less-accurate and slower-but-more-accurate versions of DNNs.

Recall that our objective in this paper is to *maximize the accuracy* of the pipelines deployed on distributed, diverse edge devices. Higher throughput leads to increased accuracy by reducing the number of dropped frames [19], [26]. Frame drops primarily come from an imbalance of processing rates at different stages of the pipeline, giving rise to excessive queuing at one or more stages. This can be alleviated by a 'bottleneck removal' approach. We iteratively replicate and strategically place the *bottleneck modules*. Bottleneck modules are those modules generating the highest delay of all modules in the pipeline, where the delay of a module is the sum of the network transfer delay plus the processing time for that module. While we do not directly optimize latency, it is achieved indirectly by identifying and alleviating the bottleneck. To prevent queue buildup, the queue size is kept very small, thereby capping maximum queuing delays. The proposed strategy must be executed on already resource-constrained edge devices, and therefore, it must have a low overhead. So, we look for low-overhead, fast heuristics that work well in practice.

Identifying the bottleneck module(s) is important before replicating it and determining optimal placement. A key challenge in a dynamic setting is that the *bottleneck may shift* with load or with a change in deployment. For example, consider the pipeline shown in Figure 5a. For simplicity, assume each new instance of a module has equal throughput. If the inference time per request for each module is in the ratio $2 : 1 : 3$ (corresponding to Module A, B, and C), then Module C is the bottleneck. Adding a new instance of Module C and distributing the load equally changes the ratio to $2 : 1 : 1.5$, shifting the bottleneck to Module A. There are also other challenges. The bottleneck depends also on the actual placement of module instances, even if the number of module instances remains unchanged. For example, the inference time ratios in the above example may depend on the actual devices the modules are located on and their resource availability, which in turn depends on hardware configurations and workloads. Overall, instead of trying to achieve a global optimal over all possible replications and placements which is an np-complete problem [27], we use a simple iterative approach based on bottleneck analysis that we will describe in the following subsections.

TABLE I: Notations

E	The set of all available edge devices
$\tau_{m,e}$	Mean processing (service) time of module m on edge device e
λ_k	Fraction of aggregated (input) arrival rate of all frames over all pipelines that is attributed to pipeline k
$d_{m,e}$	Mean network delay experienced by module instance m on edge device e
$f_{k,m,e}$	Fraction of frames of pipeline k processed by module m on edge device e

A. Bottleneck Analysis

We define our problem as minimizing the 'cost' of the bottleneck module while keeping GPU and memory usage for all modules within predetermined bounds to avoid resource contention. These bounds are set by available edge resources.

The cost of a module instance is the sum of network delay d for data transfer from the previous module and the inference time τ of the module instance, given the current placement choices being considered. This cost is scaled by the fraction of frames sent to the module instance computed over all frames arriving at the entire system over all active pipelines (similar to the use of 'visit ratios' in classical bottleneck analysis.) This scaling factor depends on respective arrival rates of different VA pipelines, the number of pipelines sharing a specific module and the number of instances of the module used. The notion of cost above thus signifies the 'service demand' on the module.

Similar to other works [14], [19], we perform offline profiling to determine the inference time for each module for each edge device. We similarly use profiling to estimate the amount of data transferred between modules to estimate the network transfer delay. We use a simple estimate of the network bandwidth to compute the delay. For infrequent changes, such as adding a new camera with a different stream resolution, offline profiling and subsequent placement can be redone. For more real-time changes in arrival rate and frame content, we propose dynamic model switching (Section V) that is not captured in the solution approach we discuss here.

B. Iterative Approach

The necessity to solve replication and placement problems jointly while being mindful of heterogeneity makes this problem challenging. Besides, using compute-intensive optimization methods such as mixed integer linear programming (MILP) is not feasible in resource-constrained edge devices, encouraging the need for lightweight heuristics.

OVIDA uses an iterative approach for bottleneck identification and removal. An initial placement is iteratively improved

by first identifying the bottleneck module and then replicating that module and placing the new module instance, assuming the needed edge resources are available to accommodate the new instance. This in turn changes the costs of some or all modules as the service demand changes due to changes in visit ratios, and speficially reduces the bottleneck cost. The iterations continue until a *saturation point* is reached, i.e., no improvement in the bottleneck cost is possible or no further resources are available to replicate and place new instances. Note that the algorithm reduces the bottleneck cost monotonically, thereby guaranteeing convergence.

Algorithm 1 Initialization

```
for s ∈ S do
    cost ← ∞
    node ← None
    m ← new module instance of service s.
    for e ∈ E do
        lowestcost ← d[m][e] + τ[m][e]
        if lowestcost < cost then
            cost ← lowestcost
            node ← e
        end if
    end for
    Place e ← m
end for
```

Initialization: To initialize the pipeline, OVIDA iterates through all services in the order of their position in the VA pipeline, placing one module instance per service. Iteration in the order of positon in the pipeline is critical because the networking cost calculation for each module depends on the placement of modules in the previous services. Each module is placed on the edge node with the lowest cost for that specific module. This initialization algorithm is illustrated in Algorithm 1.

Bottleneck detection and interative replication and placement: Algorithm 2 describes the iterative approach mentioned before. For notational convenience we assume that the same module is not replicated multiple times on the same edge device.[1] A data structure in the form of a cost matrix is maintained — $cost(m, e)$ that presents the cost when a module instance m for service s is placed on edge device e. The visit ratios are also maintained in a similar form $visit_ratio(m, e)$, presenting the fraction of the aggregated arrival frame rates seen by this instance. Note that $visit_ratio(m, e) = \sum_k \lambda_k f_{k,m,e}$, where λ_k is the fraction of total arrival frame rates attributed to the VA pipeline k that uses this instance and $f_{k,m,e}$ is the fraction of the frames of the pipeline k processed by this instance. $f_{k,m,e}$ is determined by the algorithm.

$$cost(m, e) = visit_ratio(m, e) \times (d_{m,e} + \tau_{m,e}).$$

[1]This is only to simplify presentation, the algorithm implemented and evaluated can handle any possibility.

Algorithm 2 Placement and replication

```
(m, e) = arg max cost[][] /* Identify the bottleneck */
e_c = φ; lowestcost = ∞
for e' ∈ E do
    Copy cost[][] to cost'[][]
    Copy visit_ratio[][] to visit_ratio'[][]
    if enough remaining capacity on e' to accommodate m then
        /* Check whether the bottleneck can be removed*/
        Update visit_ratio'[m][e] for all (m, e) for which cost'[m][e] is defined, assuming a new instance of m is now mapped on e'.
        Recalculate cost'[][] based on updated visit_ratio'[][]
        if cost'[m][e] < cost[m][e] and max cost'[][] < max cost[][] then
            /* This choice removes the bottleneck*/
            /* Remember the choice */
            if max cost'[][] < lowestcost then
                e_c = e'; lowestcost = max cost'[][]
            end if
        end if
    end if
end for
```

Algorithm 2 presents one single iteration. In each iteration, the largest element (m, e) of the cost matrix represents the current bottleneck. An additional instance of the bottleneck module m is created. A check is performed to see whether this new instance, when placed in edge node $e' \in E$, alleviates the bottleneck, assuming that such placement is at all doable (i.e., e has enough remaining capacity). For each check, the visit ratios are (temporarily) updated by recalculating $f_{k,m,e}$, for all e where an instance of the module m is already placed (including the new e'). For simplicity, $f_{k,m,e}$ is kept in inverse proportion to the delay $(d_{m,e} + \tau_{m,e})$. After these checks are performed, the best option is selected (e_c in the Algorithm). The best option is the one that reduces the cost of the bottleneck and any new bottleneck (to be treated again in the next iteration) has a lower cost than before.

Once the best option is determined, the visit ratios and cost matrix are updated (not directly shown in the Algorithm). At this point, the estimated GPU and memory usage of the chosen edge node e' are adjusted based on the offline profiled GPU and memory usage. Thus, the remaining capacity of e' is available for the next iteration.

The iterations continue until there is no further decrease in the bottleneck cost or a new module instance cannot be created/placed due to resource saturation. This heuristic approach results in higher throughput and accuracy compared to previously known approaches, as demonstrated in Section VII. The replication and placement algorithm runs when a VA application or edge device arrives or departs. However, in real scenarios, such events are rare, and the computation overhead is amortized over the system's total runtime.

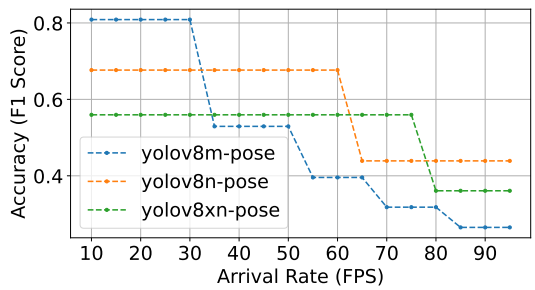


Fig. 7: Load vs. accuracy relationship for three variants of YOLO model for pose estimation on Orin NX at 25W power profile.

V. Adaptive Model Selection

To simplify the placement process, it is assumed that each module instance will use the most accurate (and often slowest) DNN model available. Ironically, this can have a negative impact on accuracy under certain conditions. For example, if the arrival rate increases, the modules with slower but more accurate models will not be able to keep up with the load, and will eventually drop more frames, resulting in an accuracy reduction. In such cases, accuracy can be improved if each module dynamically adapts to input load variations and selects the best model based on the load conditions.

Understanding the trade-off between accuracy loss due to (i) frame drops and (ii) using faster, less accurate models is essential. Figure 7 gives an example of this trade-off between three variants of the Yolov8-pose model [23]: medium (m), nano (n), and extra-nano (xn) (more on this later in Section VI). These models differ in terms of accuracy and inference times. A higher arrival rate results in more frames being dropped for the slower models, causing faster but less accurate models to perform better when the load is higher.[2] To exploit this trade-off at runtime, a dynamic approach is needed to switch between different models to achieve the best possible performance at all times.

Prior studies have introduced methods for model selection that are optimized for either accuracy or latency. Unfortunately, these approaches have two significant drawbacks that make them unsuitable for edge deployments consisting solely of embedded devices. First, they require all or multiple models to be pre-loaded into the memory [14], [17]–[20]. Given the limited memory availability and the need for edge devices to support multiple modules, this is impractical. Second, model selection methods such as integer linear programming [14], [19] or custom-trained DNNs [18] are computationally intensive, which further limits their feasibility for being executed on edge devices. As we discussed in Section IV, for an edge-only deployment, it is imperative to use low-overhead approaches.

We develop a low-overhead technique for model selection that adapts to changes in arrival rate and frame content. The technique requires advance offline profiling for online estimations. We profile inference times and their impact on pipeline accuracy for all models using different types of frame content that may be possible and easily estimated (for example, different lighting conditions). With this data, we estimate the relationship between arrival rate and accuracy. We calculate the drop rate ($max(\frac{\lambda-\mu}{\mu}, 0)$), based on which a fraction of frames are dropped at equal intervals, and the resulting accuracy is then recorded. Here, λ and μ are arrival and service rates, respectively. Then, when the pipeline is operational with live traffic, each module periodically assesses its arrival rate and checks for any changes in frame content. Changes in frame content, such as day, night, and camera out-of-focus conditions, can be detected using low-overhead classical computer vision methods [28], [29]. Arrival rate and frame content information can be plugged into the offline-estimated load-accuracy relationship to estimate the best model for the current conditions. If a model other than the current one is predicted to have better accuracy, it is loaded in the background. OVIDA switches to this new model when the model is loaded and ready, with very minimal disruption to the pipeline processing. Note that memory limitations can be managed during the loading of the new model by employing memory swap, albeit with a temporary drop in performance.

In our current implementation, the model selection is performed independently for each module, if a choice of models is available. While coordination across the entire pipeline may improve performance further, the overheads incurred for such a joint optimization may not be justifiable.

VI. Methodology and Experimental Setup

A. Experimental Testbed

Our heterogeneous compute testbed consists of five edge compute devices; two 16 GB NVIDIA Orin NXes (orin nx), two 8 GB NVIDIA Orin Nanos (orin nano), and one 4 GB NVIDIA Jetson Nano (jetson nano) [8]. The orin nanos and orin nxes acted as edge nodes and the jetson nano was used for streaming. To further increase heterogeneity, orin nxes were set to 10-watt and 25-watt power profiles each, while orin nanos were set to a 15-watt power profile. All devices were connected over a switched LAN with a peak throughput of 1 Gbps and a link delay of less than a millisecond. We emulated various limited bandwidth network conditions using tc packet filtering [30]. We used Kubernetes (K8s) [21] to connect and manage the cluster of edge devices through a virtual network. Each function was containerized using Docker [22]; K8s manages the deployment and networking of these containers. We considered varying network topologies, load conditions, and deployment types for evaluations.

Network Topologies: We used three different network topologies: local area (LAN), wide area (WAN), and a combination of the two (Extended LAN). For LAN, we do not limit the network in any way. For WAN, download and upload speeds range from 25–244 and 8–195 Mbps, respectively. Similarly for Extended LAN, download and upload speeds range from 25–1000 and 8–1000 Mbps, respectively. Upload and

[2]While very high frame rates have been used for this experiment, on edge hardware with lower capability, lower and more realistic frame rates can achieve the cross-over points. Also, sometimes the same module is shared by multiple pipelines, making the effective frame rate high.

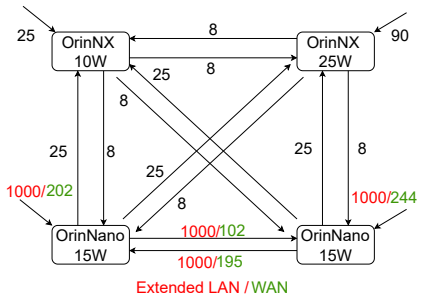


Fig. 8: Illustration of the WAN and Extended LAN network topologies we experimented with.

download speeds for WAN and Extended LAN are illustrated in figure 8. Differences between the two are highlighted in red and green, while identical links are shown in black. The selection of link delays and bandwidth was motivated by the Ookla speed test report for fixed networks [31].

B. VA Application Pipelines

To evaluate our contributions, we implemented three VA pipelines: (i) Number Plate Recognition (NPR), (ii) Vehicle Counting (VC), and (iii) Activity Recognition, (AcRg).

The NPR pipeline consisting of three unique DNN based services: object detector (D), number plate detector (N), and optical character recognition (O). The object detector detects vehicles in frames. It forwards the cropped images of the detected vehicles to the number plate detector module(s), which then detects number plates in the cropped images. The detected number plates are further cropped and forwarded to the optical character recognition module(s), which recognizes the characters in the number plate images and saves the results. The DNN models used are YOLOv5-medium [32] for the D modules and YOLOv4-tiny [33] for the N and O modules. We use the unique number plates detected as the accuracy metric for NPR pipeline. Due to the absence of a ground truth in the dataset employed, we consider the set of number plates detected when no frames are dropped ("golden configuration") as the ground truth.

The VC pipeline consists of DNN-based object detector (D) and a counter (C). The object detector service is the same as the one used by the NPR pipeline. The counter counts and reports the number of vehicles detected by the object detector. We use the number of cars detected as the accuracy metric.

The AcRg pipeline, inspired by the work of Huang et al. [34], consists of two DNN-based services: pose-estimator (P) and activity recognition (A). Pose-estimator detects people in the image and generates cropped images with pose overlaid on them. These cropped images are sent to the activity recognition module(s), which classifies the activity in the cropped images. The pose-estimator uses yolov8-pose variants (medium (*m*), nano (*n*), extra nano (*xn*)) [23], trained on the coco dataset [35]. The medium and nano variants are off-the-shelf models whereas the extra nano variant was trained ourselves with a lower depth and width parameters compared to the nano variant. We fine-tuned two more variants of yolov8m-pose (*md* and *mb*) for pose estimation under specific conditions: (i) low light, and (ii) out-of-focus frames. This was achieved by reducing the gamma value and adding Gaussian blur to the training images from the coco dataset for each respective variant. The activity recognition uses a yolov8-cls nano [23] model fine-tuned on the training set of the fall dataset [36]. Since we have access to the ground truth, we use F1-score as the accuracy metric for the AcRg pipeline.

Figure 2 illustrates the three pipelines discussed above. All models used were converted to TensorRT [37] format, which improves inference throughput without affecting accuracy.

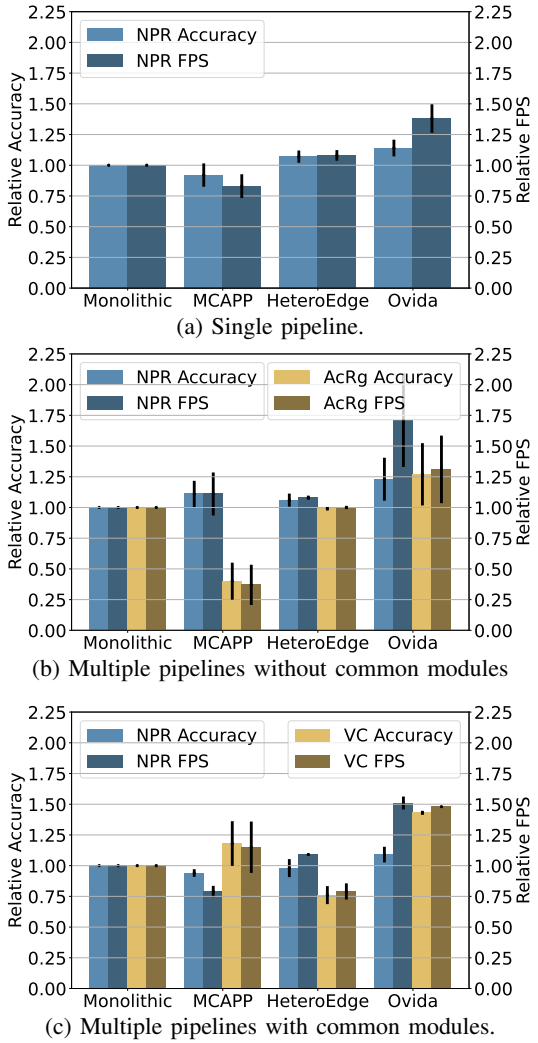
Datasets: We used the VehicleRear dataset [38] which provides videos captured at 30FPS with 1920x1080 resolution to evaluate the NPR and VC pipelines. To evaluate the AcRg pipeline, we used the fall dataset [36] and compiled a single video from all the images in the dataset.

Load Conditions: The load for the VA pipelines was varied by adjusting the number of simultaneous streams processed by the system, ranging from 1 to 3. Increasing the number of streams beyond 3 resulted in impractically low accuracy due to the limited performance of the edge devices.

C. Baselines

We compare OVIDA against the following baselines:

- *Monolithic* is the baseline deployment, where there is no disaggregation of VA pipeline, wherein the entire pipeline is implemented as a single code block on a single compute device. Unlike a disaggregated deployment, there is no parallelism and an input frame is processed completely through the pipeline before a new frame can start processing. Each stream has its own monolithic module and there is no sharing of frames among them.
- *MCAPP* is a solution framework for the multi-component application placement problem proposed by Bahreini et al. [16]. The placement is done in two steps: (i) Matching the modules and edge nodes using the well known Hungarian matching algorithm [39]. The cost for each combination of edge node and a module is the sum of service cost and the communication cost between the user and the edge node. (ii) Considering inter-component communication costs to make swaps in module-node mappings such that the overall cost of the pipeline is reduced. The number of instances of each module deployed corresponds to the number of streams handled by the deployment. While MCAPP only allows a single module per edge device, we enhance its performance by allowing the sharing of an edge device via modules processing separate video streams.
- *HeteroEdge* is a placement policy based on Hetero-Edge's fast heuristic [11] that places the modules on edge devices with GPU usage less than a predefined threshold. Although the original heuristic does not support multiple DNN-based modules, we made minor tweaks to allow that. The policy traverses over the module instances in pipeline order for placement (for example, D-N-O order in NPR pipeline). Similar to MCAPP, the number of module instances corresponds to the number of streams handled by the deployment.



(a) Single pipeline.

(b) Multiple pipelines without common modules

(c) Multiple pipelines with common modules.

Fig. 9: Performance comparison between OVIDA and other policies under different deployment scenarios. All metrics are normalized with respect to Monolithic.

- *OVIDA* employs the placement policy from Section IV.

To ensure a fair comparison of placement policies, we use central-queue–based load balancing (see Section III) for all baselines in our evaluation.

VII. Evaluation

We evaluate the effectiveness of OVIDA by measuring performance and resource consumption via our testbed deployment. We first evaluate the two main components: OVIDA's placement heuristic and adaptive model selection, followed by an end-to-end evaluation that combines the two.

A. Placement Evaluation

Figure 9 shows the median achieved throughput (in FPS) and VA application accuracy under various conditions (see Section VI) for three deployment types: (i) single pipeline (NPR pipeline), (ii) multiple pipelines without a common module (NPR and AcRg pipelines), and (iii) multiple pipelines with common modules (NPR and VC pipelines). The plots are normalized using *Monolithic* as the baseline, with all reported metrics divided by the baseline values. In the case of multiple pipelines with common modules, *OVIDA* has the ability to share common modules, whereas other policies do not.

We see that *MCAPP* performs poorly, achieving almost the lowest FPS and accuracy compared to other policies in all three deployments. Deploying two pipelines without common modules (Figure 9b) results in a 60% lower accuracy for the AcRg pipeline and a minimal improvement of 10% for the NPR pipeline compared to *Monolithic*. This is to be expected as *MCAPP* only places one module instance per stream on an edge device, ***thereby incurring communication cost every time a frame is transferred between module instances.***

HeteroEdge performs slightly better than *Monolithic* for single pipeline and multiple pipelines without common module deployments. Unlike *MCAPP*, *HeteroEdge* is able to place multiple module instances on a single device (until GPU or memory saturation), resulting in lower inter-module communication delays. ***Due to the parallelism of a disaggregated pipeline, HeteroEdge outperforms Monolithic*** by 7% in terms of accuracy in the best case (Figure 9a). ***The unbalance caused by saturating a device before moving on to the next one, overshadows the gains of parallalism and results in performance worse than Monolithic in case of pipelines with common modules (Figure 9c).***

OVIDA outperforms *Monolithic*, *MCAPP*, and *HeteroEdge* in all deployment cases. In the case of single pipeline deployment (Figure 9a), *OVIDA* achieves 14% higher accuracy and 38% higher throughput compared to *Monolithic*, and 7% higher accuracy and 27% higher throughput than *HeteroEdge*, the next best placement policy. For multiple pipelines without common modules (Figure 9b), *OVIDA* achieves 14% accuracy and 58% throughput gain for the NPR pipeline, and 27% accuracy and 31% throughput gain for the AcRg pipeline compared to the next best policy, *HeteroEdge*. Finally, for multiple pipelines with common modules (Figure 9c), *OVIDA* achieves a 13% accuracy and 50% throughput gain for NPR pipeline and 41% accuracy and 48% throughput gain for VC pipeline compared to *Monolithic*. Sharing common modules provides an additional accuracy boost of 4% for the NPR pipeline and 20% for the VC pipeline, compared to *Monolithic*. Here, VC and NPR pipelines' common service, object detector, was the bottleneck. Increasing the number of object detectors significantly boosts VC's performance, as it is the only resource-intensive module in the pipeline.

OVIDA outperforms the baseline approaches because it alleviates the bottleneck by selectively scaling the module instances. Further, it takes into account the heterogeneous communication delays and service times among edge nodes while performing the placement. We also find that *OVIDA* achieves high GPU utilization (but comparable to HeteroEdge), as shown in Figure 10, which shows the median GPU utilization for all policies across all deployments. By more efficiently placing modules, *OVIDA* makes better use of GPU resources.

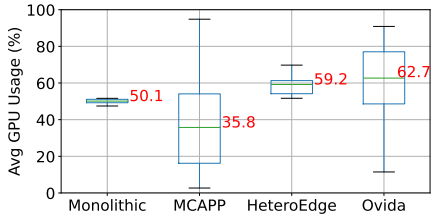


Fig. 10: GPU usage across all devices for the evaluation shown in Figure 9.

B. Adaptive Model Selection

We now demonstrate how OVIDA enhances application accuracy when load changes unpredictably by dynamically selecting the most suitable model. In the VA context, load can change as a result of *change in arrival rate* (for example, due to change in the number of input streams) or *change in frame content* (for example, change in lighting conditions). In case of change in arrival rate, re-placement can be performed in response to each change; however, this will cause some application downtime as modules are redeployed under the new placement. Nonetheless, re-placement cannot address a change in frame content as placement techniques are oblivious to the real-time content of incoming frames.

To enable a low-overhead solution that can also address changes in frame content, we consider dynamic model selection whereby the underlying model implementation for a module can be switched at runtime. Dynamic model selection can lead to better accuracy in the presence of changing workload conditions, as discussed in Section II. We evaluate OVIDA's model selection under two frame transformations: (i) Darkended frames, representing night or low-light indoor locations; and (ii) Blurred frames, representing the case where a camera is out of focus or has been tilted/panned/zoomed to point to a different location. Such transformations can be easily detected using classical computer vision techniques [28], [29], and can thus trigger a search for a better model to improve accuracy under the transformed frame conditions.

For our model selection evaluation, we use the AcRg pipeline because the pose-estimation module effectively demonstrates the benefits of dynamic switching. Similar to our placement evaluations, we initially deploy all module instances with the best available model, which can then be switched later. OVIDA checks for changes in arrival rate and frame content every ten seconds. If a switch is warranted, the new model is loaded in the background. Once loaded, the input workload is redirected to the new model, and the old model is unloaded from memory. All models we used take ∼25 seconds to load on orin nano and ∼19 seconds to load on orin nx; note that the workload is being actively serviced by the older model while the newer model is loaded.

Figure 11a shows how OVIDA reacts to a change in the arrival rate. The figure shows the achieved accuracy under different policies as a function of time. Static policy using different models is represented as dotted colored lines. Black solid line shows OVIDA's adaptive model selection (denoted as AMS). The green shaded region represents the duration for which only a single stream is assigned to the deployment (arrival rate of 30 FPS). In this case, the *m* (medium) model's has a low drop rate, and thus performs better than *n* (nano) and *xn* (extra-nano). The red shaded region represents the time when three streams are assigned to the deployment (arrival rate of 90 FPS). Due to the *m* model being slow and dropping more frames, *n* performs better in this region. The vertical blue lines in Figures 11a and 11c indicate the timestamps when the model switches occurred. Multiple blue lines near a load change point denote the model switch for each of three module instances of pose-estimation. As can be seen from the figure, OVIDA's dynamic switch is able to react to the changes in arrival rate and switch to the better model accordingly. In terms of median accuracy, ***dynamic model switching achieves a 15% improvement over the default placement using only the* m *model and a 10% improvement over the* n *model, the best static choice in this case (see Figure 11b)***.

Figure 11c shows OVIDA in action when the frame content changes. The green, red, and yellow regions represent in-focus images with good illumination, dark images, and out-of-focus images, respectively. Throughout this experiment, two streams were assigned to the deployment (arrival rate of 60 FPS). Similar to Figure 11a, vertical blue lines represent switch points. OVIDA starts by switching to *n* model on orin nanos and the *m* model on orin nx to adjust to the arrival rate of 60 FPS. In dark and out-of-focus parts of the stream, OVIDA dynamically selects the appropriate model (*md* for dark and *mb* for blur). In terms of median accuracy, ***dynamic model switching in response to changes in frame content achieves a 14% improvement over the default placement using the* m *model and a 10% improvement over the* mb *model, the best static choice in this case (see Figure 11d)***.

We thus conclude that OVIDA reacts appropriately to workload changes by dynamically selecting the best model. Note that the actual accuracy and gains are a function of the application, models, frame content, and datasets.

C. End-to-end Evaluation of OVIDA

We now evaluate the benefits of combining OVIDA's placement and adaptive model selection against other placement schemes. Similar to Section VII-B, we used the pose-estimation pipeline. The input load traces are the same as those employed in Figures 11a and 11c.

Figure 12 shows the relative FPS and accuracy achieved by different policies for the above-mentioned load traces. The 'Monolithic', 'MCAPP', 'HeteroEdge', and 'OVIDA Place' policies shown only perform the initial placement using the fixed *m* model. OVIDA Place with fixed *m* model achieved throughput and accuracy gain of 17% and 11% over Monolithic. Our end-to-end system, OVIDA with placement and adaptive model selection ('OVIDA Place+AMS') achieves significantly better performance, with throughput and accuracy gains of 51% and 28%, respectively, over Monolithic. ***Compared to OVIDA's placement-only policy, OVIDA's placement+AMS thus obtains additional throughput and accuracy***

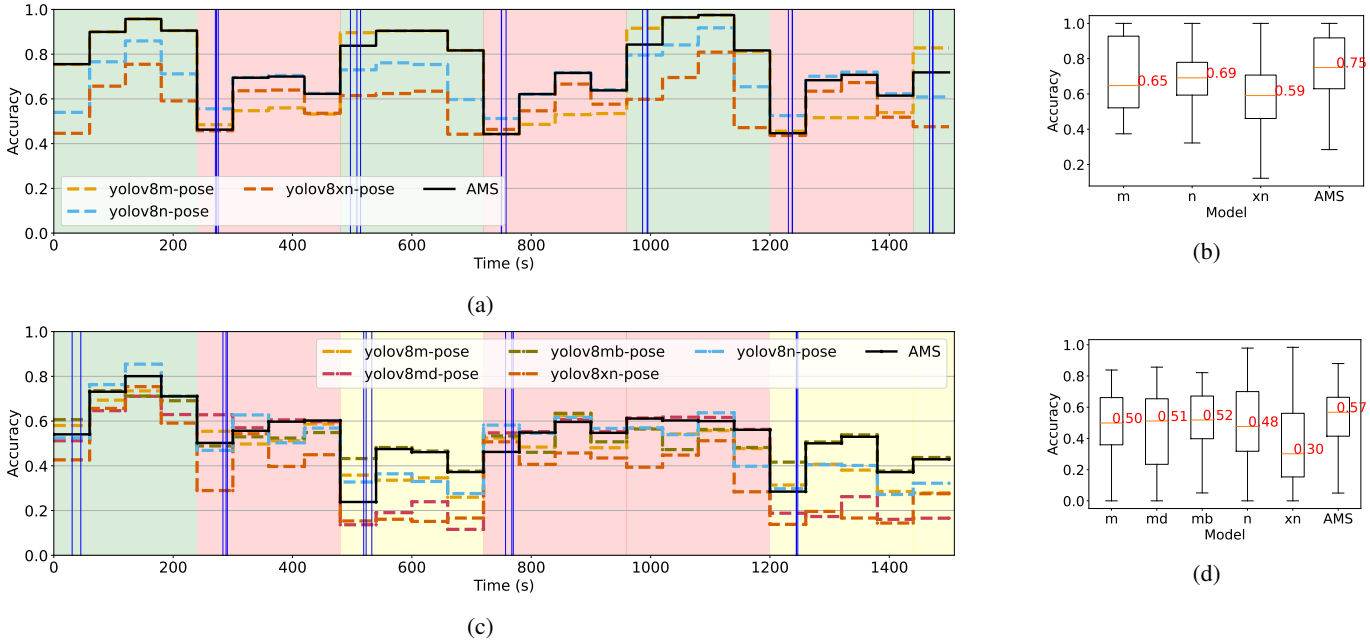


Fig. 11: (a) Selecting the appropriate models in response to changes in arrival rate. (b) Distribution of accuracy for AcRg pipeline in (a), sampled every 30s. (c) Selecting the appropriate models in response to changes in frame content. (d) Distribution of accuracy for AcRg in (c), sampled every 30s. Note that 'AMS' here refers to OVIDA with Adaptive Model Selection.

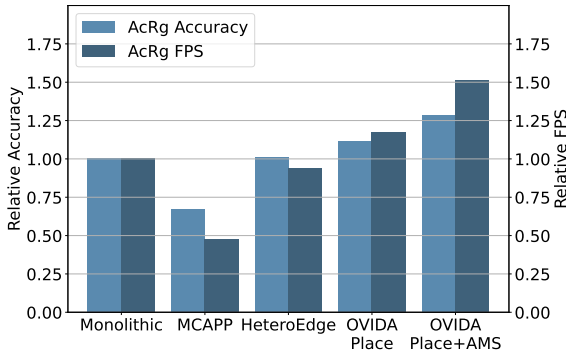


Fig. 12: End-to-end performance improvement afforded by OVIDA with placement and model selection for the pose estimation pipeline.

gains of 29% and 15%, respectively, highlighting the benefits of combining model placement and dynamic model selection.

VIII. Related Work

Table II presents a summary of the important features that set OVIDA apart from existing work. In this section, we first discuss prior works that focus on VA placement and then those that tackle the VA model selection problem.

Module Placement: The placement problem in the VA context refers to scheduling module instances on available edge devices to maximize performance (throughput and accuracy). Earlier works have explored different system designs to address this problem. (i) *Edge-cloud collaboration:* Edge devices are only either used for partial processing or for filtering out frames before sending them to the cloud for serving the remaining VA operations [5], [13], [40]. (ii) *Hierarchical processing:* Video streams go through multiple layers of processing infrastructure such as smart cameras, nearby workstation cluster, and the cloud, each of them hosting some VA function(s) [24]. (iii) *Local edge:* Completely processing video streams at the edge using the devices in the same local area network [11], [12], [41]–[43]. While many prior works have explored several distinct combinations of system designs, apart from Bahreini et al. [16] none have considered the possibility of a ***distributed edge node system hosting a multi-component VA pipeline***. Further, prior works often make certain limiting assumptions, such as assuming (i) the availability of powerful workstations or server-class machines with discrete GPUs to execute neural network-based VA functions, and (ii) all edge nodes being present in a local network connected with high-speed links.

Among the above works, we will now discuss those most closely related to ours. Bahreini et al. [16] propose a generic heuristic for placing multi-component pipelines on distributed edge nodes. However, the heuristic is limited to placing *only one* module per pipeline on a node, which can greatly limit performance (as we show in Sections II and VII-A).

Hetero-Edge [11] presents a system that works with heterogeneous edge nodes and places the bottleneck module on a single GPU, if that GPU's usage is less than a threshold, before moving to other alternative devices. If no GPU is available, the other functions are parallelized over the CPU of the same or a different node. This heuristic assumes only one module

TABLE II: Feature comparison with most relevant works.

Features	MCAPP [16]	Distream [12]	HeteroEdge [11]	OVIDA (Ours)
Distributed edge nodes	✓	✗	✗	✓
Hosts pipelines with multiple DNN based modules	✓	✓	✗	✓
Adaptive load balancing	✗	✓	✗	✓
Bottleneck detection and scaling	✗	✗	✗	✓
Does not require a workstation	✓	✗	✓	✓
Adaptive model switching	✗	✗	✗	✓

instance for every service and further assumes that there is only one module in the pipeline that requires GPU access. These assumptions do not hold for many VA tasks (Section VII-A).

Distream [12] proposes (i) dynamic load balancing among smart cameras, and (ii) finding an optimal point to slice the pipeline such that the first half is placed on a smart camera and the second on a powerful GPU-enabled workstation that is assumed to be present in the same local area network. However, Distream is quite rigid in its system design in that it requires a powerful workstation and a high-speed connection among all edge devices, which may not always be available in an edge-only scenario.

Oakestra [15], a general-purpose orchestrator for edge deployments, is a scheduler to place applications on suitable edge nodes based on an extensive set of factors including bandwidth, memory, GPU and CPU requirements, and network latency from the source. However, unlike OVIDA, Oakestra focuses on placing an entire, independent application on a single node rather than appropriately distributing components of an application pipeline that interact with each other.

Model Selection: Existing works have proposed strategies for dynamic model selection with distinct system designs for embedded devices [20], [44] and workstations on edge [14], [17]–[19]. We next discuss the closely related works that focus on model selection on heterogeneous distributed edge computing deployments.

INFaaS [14] allows developers to register multiple models per application; the submitted models are then hosted on available edge devices. A user's query to INFaaS contains the image to process, the VA application to use, the maximum allowed latency, and the minimum required accuracy. INFaas then selects the best possible model-device combination to satisfy the query. However, INFaaS was designed and evaluated only on high-performing edge devices with enough memory to load multiple models simultaneously.

Proteus [19] uses a similar system design to INFaaS, except that it optimizes model-device selection for maximizing accuracy as opposed to maintaining minimum accuracy and maximum latency bounds. However, it also requires loading multiple model variants on an edge device. Further, Proteus models the optimization as a mixed integer linear programming problem, whose solution is computationally intensive. Proteus' high memory and computing demands make it infeasible for edge deployments lacking powerful workstations.

LiteReconfig [44] proposes an approach to select an inference model based on light features (frame resolution, number of objects detected) and heavy features (histogram of color, histogram of oriented gradients, and feature extraction using a DNN among others) to maximize accuracy for a minimum latency constraint. It utilizes a custom-trained DNN model to estimate the additional latency cost of using the heavy features and determines whether to employ them. Based on this estimated latency cost, it selects the inference model to maintain both accuracy and latency within specified bounds. RAVAS [18] proposes a low-overhead technique to share the GPU among different models processing multiple video streams. RAVAS uses reinforcement learning while receiving periodic accuracy feedback from the "golden configuration"—the best configuration of models. Both LiteReconfig and RAVAS require all the available models to be pre-loaded into memory for quick switching and comparison, which is not feasible in edge deployments due to memory constraints. In contrast, OVIDA only keeps one model in memory and makes a swap when required.

IX. Conclusion

In this paper, we presented the design and evaluation of OVIDA. To our knowledge, OVIDA is the first fully disaggregated video analytics design that has been evaluated on a testbed of multiple embedded devices for three VA applications and three network topologies, under varying load conditions. Our experiments show the OVIDA's placement policy can outperform all baseline techniques in terms of median accuracy. Compared to the next-best policy, OVIDA achieves accuracy gains of up to 7%, 27%, and 41% for single pipeline, and multiple pipelines with and without common modules, respectively, under various network and load conditions. We also present an adaptive model selection technique that selects the most appropriate model for a VA task and seamlessly switches to it as needed, in response to changes in load. OVIDA's adaptive model selection can provide additional accuracy gains of 15%. We believe that OVIDA's design can easily be scaled to efficiently support large VA systems on a network of distributed edge nodes.

Acknowledgements

This work was supported by NSF grants CCF-2324859, CNS-2214980, CNS-2106434, CNS-1909356, CNS-1750109, CNS-2055520, CNS-2106594, and a Stony Brook University OVPR Seed Grant. This work was also supported in part by the ANR Project N^o ANR-21-CE25-0013 (PARFAIT). We would like to express our heartfelt thanks to Goutham Surya Arukonda and Omkar Manjrekar for their contributions in implementing the VA pipelines used in this paper.

References

[1] Fortune Business Insights. CCTV Camera Market Size, Growth — Global Report. [Online]. Available: https://www.fortunebusinessinsights.com/cctv-camera-market-107115

[2] "Video Analytics Market," accessed: 2024-7-4. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/intelligent-video-analytics-market-778.html

[3] Y. Zhang, J.-H. Liu, C.-Y. Wang, and H.-Y. Wei, "Decomposable intelligence on cloud-edge iot framework for live video analytics," *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 8860–8873, 2020.

[4] Z. Hu, N. Ye, C. Phillips, T. Capes, and I. Mohomed, "mmFilter: Language-guided video analytics at the edge," in *Proceedings of the 21st International Middleware Conference Industrial Track*, ser. Middleware '20, 2020, p. 1–7.

[5] H. Sun, Y. Yu, K. Sha, and B. Lou, "mVideo: Edge computing based mobile video processing systems," *IEEE Access*, vol. 8, pp. 11 615–11 623, 2020.

[6] "CCTV on London Underground | Freedom of Information," 2021. [Online]. Available: https://tfl.gov.uk/corporate/transparency/freedom-of-information/foi-request-detail?referenceId=FOI-0030-2122

[7] M. Ghasemi, S. Kleisarchaki, T. Calmant, L. Gürgen, J. Ghaderi, Z. Kostic, and G. Zussman, "Real-time camera analytics for enhancing traffic intersection safety," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, p. 630–631.

[8] "NVIDIA Jetson," accessed: 2024-10-18. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

[9] "Raspberry Pi," accessed: 2024-7-3. [Online]. Available: https://www.raspberrypi.com/documentation/

[10] S. P. Rachuri, F. Bronzino, and S. Jain, "Decentralized modular architecture for live video analytics at the edge," in *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, ser. HotEdgeVideo '21, 2021, p. 13–18.

[11] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "HeteroEdge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, Apr. 2019, pp. 1270–1278.

[12] X. Zeng, B. Fang, H. Shen, and M. Zhang, "Distream: scaling live video analytics with workload-adaptive distributed edge intelligence," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 409–421.

[13] M. Zhang, F. Wang, Y. Zhu, J. Liu, and Z. Wang, "Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines," in *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021, pp. 80–93.

[14] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411.

[15] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott, "Oakestra: A lightweight hierarchical orchestration framework for edge computing," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 215–231.

[16] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–11.

[17] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 253–266.

[18] A. Rahmanian, A. Ali-Eldin, S. K. Tesfatsion, B. Skubic, H. Gustafsson, P. Shenoy, and E. Elmroth, "RAVAS: Interference-Aware model selection and resource allocation for live edge video analytics," in *2023 IEEE/ACM Symposium on Edge Computing (SEC)*, 2023, pp. 27–39.

[19] S. Ahmad, H. Guan, B. D. Friedman, T. Williams, R. K. Sitaraman, and T. Woo, "Proteus: A high-throughput inference-serving system with accuracy scaling," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024.

[20] V. S. Marco, B. Taylor, Z. Wang, and Y. Elkhatib, "Optimizing deep learning inference on embedded systems through adaptive model selection," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 1, pp. 1–28, 2020.

[21] "Production-Grade container orchestration," accessed: 2023-7-26. [Online]. Available: https://kubernetes.io/

[22] "Docker: Accelerated, containerized application development," accessed: 2023-7-26. [Online]. Available: https://www.docker.com/

[23] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics yolov8," 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[24] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "VideoEdge: Processing camera streams using hierarchical clusters," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 115–131.

[25] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[26] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali, "Gemel: Model merging for memory-efficient, real-time video analytics at the edge," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 973–994.

[27] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, pp. 393–406.

[28] I. Luchko, "Digital camera autodetect," accessed: 2024-6-22. [Online]. Available: https://github.com/luchko/digital-camera-day-or-night

[29] J. L. Pech-Pacheco, G. Cristobal, J. Chamorro-Martinez, and J. Fernandez-Valdivia, "Diatom autofocusing in brightfield microscopy: a comparative study," in *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, vol. 3, 2002, pp. 314–317.

[30] "tc(8) - linux manual page," https://man7.org/linux/man-pages/man8/tc.8.html, accessed: 2024-7-1.

[31] "United states's mobile and broadband internet speeds," https://www.speedtest.net/global-index/united-states?fixed, accessed: 2023-7-29.

[32] G. Jocher, "YOLOv5 by Ultralytics," May 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[33] A. Bochkovskiy, C. Wang, and H. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *CoRR*, vol. abs/2004.10934, 2020. [Online]. Available: https://arxiv.org/abs/2004.10934

[34] Z. Huang, Y. Liu, Y. Fang, and B. K. P. Horn, "Video-based fall detection for seniors with human pose estimation," in *2018 4th International Conference on Universal Village (UV)*, 2018, pp. 1–4.

[35] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common objects in context," 2014.

[36] K. Adhikari, H. Bouchachia, and H. Nait-Charif, "Activity recognition for indoor fall detection using convolutional neural network," in *2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA)*. IEEE, May 2017, pp. 81–84.

[37] "NVIDIA TensorRT," accessed: 2023-7-29. [Online]. Available: https://developer.nvidia.com/tensorrt

[38] I. O. De Oliveira, R. Laroca, D. Menotti, K. V. O. Fonseca, and R. Minetto, "Vehicle-Rear: A new dataset to explore feature fusion for vehicle identification using convolutional neural networks," *IEEE Access*, vol. 9, pp. 101 065–101 077, 2021.

[39] H. W. Kuhn, "The hungarian method for the assignment problem," *Nav. Res. Logist. Q.*, vol. 2, no. 1-2, pp. 83–97, Mar. 1955.

[40] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. Andersen, M. Kaminsky, and S. R. Dulloor, "Scaling video analytics on constrained edge nodes," *MLSys*, vol. abs/1905.13536, pp. 406–417, 2019.

[41] P. Liu, B. Qi, and S. Banerjee, "EdgeEye: An edge service framework for real-time intelligent video analytics," in *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, 2018, pp. 1–6.

[42] Q.-M. Nguyen, L.-A. Phan, and T. Kim, "Load-Balancing of Kubernetes-Based edge computing infrastructure using resource adaptive proxy," *Sensors*, vol. 22, no. 8, 2022.

[43] B. B. Cao, A. Sharma, M. Singh, A. Gandhi, S. Das, and S. Jain, "Representation similarity: A better guidance of dnn layer sharing for edge computing without training," *arXiv preprint arXiv:2410.11233*, 2024.

[44] R. Xu, J. Lee, P. Wang, S. Bagchi, Y. Li, and S. Chaterji, "LiteReconfig: cost and content aware reconfiguration of video object detection systems for mobile GPUs," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 334–351.