Microservice-based Edge Device Architecture for Video Analytics

Si Young Jang
KAIST
Daejeon, South Korea
sy.jang@kaist.ac.kr

Boyan Kostadinov
KAIST
Daejeon, South Korea
boyanyk@kaist.ac.kr

Dongman Lee
KAIST
Daejeon, South Korea
dlee@kaist.ac.kr

ABSTRACT

With today's ubiquitous deployment of video cameras and other edge devices, progress in edge computing is happening at an incredible speed. Yet, one aspect of real-time video analytics at the edge that is still underdeveloped is the support for processing multi-tenant, multi-application scenarios with a limited set of resources. Existing systems either fail to provide the necessary performance, or rely too heavily on edge or cloud servers to handle the workload. This work proposes a new approach, inspired by both Function-as-a-Service and microservices architecture in order to efficiently place and execute video analytics pipelines on edge devices. The main contributions of this work are the ability to dynamically add and run new applications on already deployed systems, and the capability to horizontally distribute pipelines across other neighbouring edge devices. We prototype an implementation that we evaluate using multiple concurrent applications per device. Results show that our system provides more flexibility for on-the-fly reconfiguration than existing works do, with 20 % improvement in latency and 3.9 X increase in throughput.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → *Distributed artificial intelligence*; Computer vision tasks.

KEYWORDS

edge computing, video analytics, microservices, serverless, real-time

ACM Reference Format:
Si Young Jang, Boyan Kostadinov, and Dongman Lee. 2021. Microservice-based Edge Device Architecture for Video Analytics. In *The Sixth ACM/IEEE Symposium on Edge Computing (SEC '21), December 14–17, 2021, San Jose, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3453142.3491283

1 INTRODUCTION

In recent years, edge computing has emerged as a field of interest in the area of distributed systems. Borrowing concepts and ideas from the cloud, the edge is supposed to provide a similarly powerful computing platform but without the expense of additional latency. This promise makes for a perfect match with video analytics (VA) applications that have strict performance and latency requirements.

Edge devices, such as Jetson Nano [1], Jetson TX2 [2] and newer phones, have ever more increasing processing capabilities. Not only are their CPUs and GPUs getting faster, but the SoCs in use have also begun to incorporate special neural processing units (NPU), sometimes called Neural Engines (Apple) or AI Engines (Qualcomm), for accelerating deep learning tasks [3, 4]. This allows them to process more complex and deep ML models that were not possible just a few short years ago.

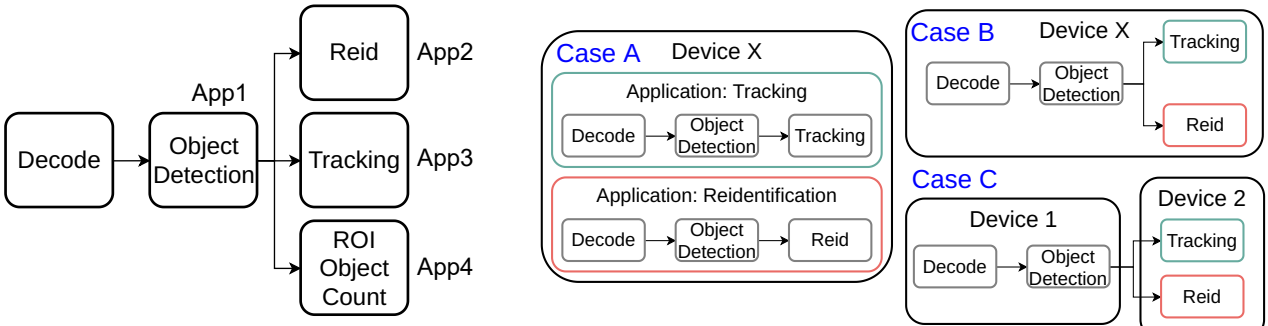
Video analytics, dubbed the killer application for edge computing [5], has seen a lot of attention from the research community, tackling the problem from various angles. Solutions range from optimising, compressing and quantizing the requisite machine learning models to run on constrained devices [6, 7, 8], through carefully-designed workload scheduling systems [9, 10], to complex edge-cloud distributed deployments [11, 12] that configure to achieve the necessary requirements of a given application scenario. In short, works can be separated into those who focus on optimising the application components and those who propose new systems or architectures to support the real-time video analytics use case.

All of these approaches provide certain benefits to the processing of VA. Yet, they generally only examine a relatively specific application in a mostly static environment. As the authors of [13] point out through their data analysis, albeit for a single application, the dynamics of a camera surveillance system should not be understated as they are ever changing. However, few works investigate support for multi-tenancy and multi-application on edge devices. It is not difficult to imagine a situation where multiple authorised parties (e.g. police, firefighters, city administrators) request for different types of real time VA services (e.g. general surveillance for public safety, tracking suspicious vehicle across multiple cameras, etc) to a privately maintained large-scale camera network in an urban environment. Some works [10, 12] do consider scenarios where multiple users might be analysing videos from the camera system simultaneously but only focus on adjusting the query scheduling and execution in a single application. In reality, supporting multiple applications for a multi-user environment is more challenging than managing their queries for a single application. To that end, it is necessary to have a generic architecture that can support VA use cases with high enough performance to meet service-level objective (SLO) targets.

In order to facilitate edge VA, prior works such as Distream [13] and EdgeEye [14] often leverage the help of an edge server to process a large portion of the analytics pipeline. Distream [13]

SEC '21, December 14–17, 2021, San Jose, CA, USA

ACM ISBN 978-1-4503-8390-5/21/12…$15.00
https://doi.org/10.1145/3453142.3491283



(a) Sample video analytic pipeline supporting 4 different application

(b) Overview of existing and proposed VA pipeline workflow

Figure 1: Illustration of sample VA application pipeline in DAG form (a) and how a pipeline looks like when deployed on devices (b)

assigns a small portion of the pipeline execution to the edge devices, creating a vertical distribution of work while balancing the request workload horizontally between the edge devices. However, support for multiple applications is lacking in their architecture, which attributes to a monolithic design. EdgeEye [14], on the other hand, designs two parts by having the edge servers do the computation, while the edge devices define and control the pipeline, and make requests to the remote services at the server side. While this approach can support multi-application, it underutilises the resources of neighboring edge devices, that is, horizontal resource provisioning of potential computing resource is not addressed. The authors of VideoPipe [15] propose horizontally distributed video pipelines that can support multiple applications on a single input. However, they lack support for processing more than one input, regardless of how many devices are involved, and they also exhibit slower than real-time performance.

With all this in mind, we propose a serverless microservice architecture that operates on top of a fleet of smart cameras (e.g. Nividia Jetson nano) for real-time VA that supports the following:

- Dynamic multi-tenant, multi-application use
- Collaborative pipeline distribution and execution horizontally across edge devices
- Sharing pipeline components across applications running on the same device for more efficient utilisation

To the best of our knowledge, this is the first work that uses a combination of serverless function and microservice concepts to provide real-time edge VA service. Previous works rely heavily on the server side for multi-tenancy/application support with serverless microservice architecture [16, 17], however, leaving resource on edge devices unexplored. We prototype a proof-of-concept (POC) system based on runtimes that handle both data and control plane operations, allowing multiple modules to concurrently run with minimal overhead. Results show 20 % improvement in latency and 3.9 x increase in throughput compared to existing works.

The rest of the paper is organised as follows: First, we describe the background of VA and the shortcomings of existing works. Then we discuss the intuition behind our approach, alongside some supporting preliminary experiments. We introduce the key considerations and proposed design in detail, as well as describe the experimental implementation. The implementation is then evaluated against competing approaches. Finally, we discuss the broader spectrum of related works and conclude the discussion of the architectural concept with potential ideas about future improvements.

2 BACKGROUND

In this section we introduce more details to the background of the edge VA field in terms of terminology and existing works, as well as some preliminary results.

2.1 VA Pipeline

2.1.1 Sharing components in VA. The most common way to visualise a VA application is through a directed-acyclic graph (DAG) representing the execution pipeline [12, 11, 16, 17, 18]. The pipelines consist of the elements or modules of the given system.

An example pipeline is shown in Figure 1a. Each block represents a module from the pipeline, with arrows representing the direction of data flow. In this case, we begin with a *decode* module which decodes the incoming video stream into frames. Those frames are fed into an *object detection* ML model (e.g. Yolo[19] etc) that outputs bounding boxes, object class, and its confidence for all recognised objects in the frame (App 1).

The output of *object detection* module can be used as source of other pipeline components. For instance, the bounding box of detected people in a frame can be cropped and run another ML model (e.g. OSNet for reidentification) to distinguish the target person in the frame (App 2). Tracking people in a frame (App 3) also requires bounding box results from *object detection* module in order to build spatial traces of the moving object in the view. Finally, monitoring and counting person in the region-of-interest (App 4) also require location of the detected person for analysis.

To fulfil the requirement of multiple application support on the system level, we first describe two existing approaches in Figure 1b. Case A represents a monolithic system model where each application is managed without prior knowledge of the sharable components in each application. Therefore, it duplicates work due to unnecessary pipeline component such as Decoder and Object Detection. Case B represents a system model used in many stream processing frameworks such as GStreamer [20] and Deepstream

Paper	Dynamic app support	High Accuracy	Efficient Resource Use	Module Reuse
Distream [13]	No	Yes	Duplicated modules	Server-side
EdgeEye [14]	Partial	Yes	No, cameras don't process	Server-side
VideoPipe [15]	Partial	With server	No, one module at a time	Server and device
This work	Yes	With server	Yes - pipeline sharing, concurrency	Server and device

Table 1: Overview of related works in the context of multi-tenant, multi-application VA

[21]. However, it is only applicable when the entire application is known beforehand, so the developers can design the pipeline once and deploy. In reality, applications and environments change frequently, so more flexibility is required - but these frameworks do not provide a way to either seamlessly update them in real time or distribute them across multiple heterogeneous edge devices.

2.1.2 In-time video processing. Generally, the most important requirement of a VA system is the ability to deliver frames under a given service-level objective (SLO) that is often defined in terms of execution latency in milliseconds. The SLO would commonly be dependent on the video input rate. For instance, if surveillance cameras are recording at 15 FPS, then that means that every component of a pipeline should generally be able to process one frame in under $1/15 = 0.067s$ ($67ms$). Failure to do so would likely mean that the pipeline either experiences a bottleneck of frames being queued up at the given pipeline element, resulting in lag or perhaps dropping frames at the very start of the pipeline, which could decrease accuracy in applications such as tracking.

Consider a scenario where an object detection model can only support processing up to 10 FPS (processing 1 frame takes 0.1 seconds). If the system processes all the frames sequentially, it takes $0.033s$ longer to process a frame. This builds up a delay of 10 minutes for every elapsed hour of video. On the other hand, if the system drops frames, then it means the system is only processing 2/3 of the incoming video, which could mean missing important events in the scene.

Therefore, satisfying SLO per application requires the system to ensure that not only the whole pipeline but also each module in the pipeline does not generate lag.

2.1.3 Vertical balancing at the edge. Application component placement logic can be configured based on various factors such as hardware capability, resource demands and application context, which all should satisfy application requirements [22, 23]. Typical placement of VA pipelines and workloads span vertically (hierarchically) between edge device - edge server, where lightweight ML operations at the edge device discard less relevant frames before reaching the edge server [24, 25, 26]. However, balancing vertically only allows the system to adjust per edge device - server regardless of how much resource is available across edge devices.

Table 1 shows the most relevant related works which distribute VA pipelines at the edge, and compare them to our work. The comparison categories are selected based on their importance in multi-tenant, multi-application VA.

Distream [13] proposes a collaborative and intelligent workload distribution across edge devices by workloads prediction. However, this system is dependent on the contents of the video, requiring prior knowledge and training in new environments. Additionally, it does not describe support for deploying and using multiple applications at the same time.

EdgeEye [14] and VideoPipe [15] apply a modularisation approach to the application DAG. They split the elements into small pieces that together build up the VA pipeline, namely, execution and control. The execution elements hold the application logic and perform the computational work, such as decoding, object detection, and other elements. Control elements, on the other hand, define the flow of the pipeline and are responsible for moving the requisite data to the next element in the DAG. In EdgeEye [14], the control modules are positioned on the edge devices, while the execution ones are situated on the edge server. In VideoPipe [15], using a server is not necessary as a control element and its corresponding execution counterpart are co-located on the same device, albeit in different services. This leads to decreased model accuracy, as it is not possible to run highly accurate models without an edge server, like EdgeEye [14] and Distream [13] does. Yet, due to their reliance on a server, they do not make efficient use of the edge devices' resources. While VideoPipe can share components across application pipelines, they only support a single video input and the pipelines are executed synchronously - only one component in the entire pipeline can be active at a time, even if other devices are idling. Overall, dynamic application support is poorly represented in existing approaches.

2.2 Preliminary experiments

We dive into a deeper discussion of our key idea of distributing pipeline elements horizontally across edge devices and perform a preliminary experiment to support our intuition.

One device can have a part of the pipeline while another contains the remainder, as shown in Figure 1b Case C. When we have multiple components running on the same device, as seen in Figure 1b Cases A and B, it causes contention when they are used at the same time because of the inherent need for context switching, leading to limited performance of the components. Instead, if one device can "focus" on a single resource-demanding component, it should be able to achieve that component's maximum performance.

A preliminary verification was conducted using one object detection model (based on ResNet10 [27]) and one person reidentification model (Osnet x1.0 [28]). Each model was deployed using the TensorRT [29] runtime on a Triton Inference Server [30] on a Jetson Nano [1] using a client library for the server to evaluate the maximum performance. Each model was run both independently and also simultaneously with the other model, and the results are shown

in Table 2. In terms of inferences per second (IPS), we can see in the IPS column that when running both models simultaneously the performance of the OSNet model drops to 52% of its maximum while ResNet drops to 48%. Additionally, the latency numbers show that running two models at once can be the difference between the application lagging and normal operation. If we assume a 15 frames per second (FPS) video, that means that every frame needs to be processed within 67ms which is true in both cases in singular mode but in simultaneous execution OSNet would not be able to keep up.

Model	Execution Mode	IPS (Higher is better)	Latency (ms) (Lower is better)
OSNet [28]	Singular	**264.6**	**48.98**
	Simultaneous	137.4	92
ResNet [27]	Singular	**90.2**	**22.1**
	Simultaneous	44	66.58

Table 2: Model performance when executed alone vs simultaneous execution. Bolded text is best-case results.

3 SYSTEM DESIGN

This section discusses the overall system design. It discusses the anticipated technical challenges and considerations, as well as the proposed architecture.

3.1 Design Considerations

3.1.1 Modularity & Extensibility. With the goal of supporting flexible, dynamic pipelines, we consider the modularity and extensibility based on two characteristics - the size of microservices and the support for a range of deep learning models.

Microservice size: Establishing the correct size of the microservices is one of the most important aspects of the planning phase. While in theory any size would work, in practice this might have huge implications for performance. Finding this size might seem trivial at first, if we just consider the heavy, AI-based components. However, there are multiple intermittent preprocessing and post-processing functions that need to be performed to be able to provide an input to a module. For instance, a DNN model in an object detection module requires input to be resized (e.g. 272x480 for ResNet10), colour space changed, transposed and normalised (e.g. changing value range from 0-255 to 0-1) beforehand. This is similar in post-processing functions such as Non-maximum Suppression (NMS) which filters out unlikely predictions from the model inference.

The question that arises is if every one of these functionalities should be its own microservice? The trade-offs are clear here. If the system components are split into the minimum functions, then network latency due to network calls can be more expensive than the actual computation latency of the component. On the other hand, if the components become too big, then we sacrifice a lot of flexibility in terms of the potential configurations, as well as the distribution of the components.

In general, in order for a microservice size to be applicable, the following assertion should hold true:

$$T_m > T_{nb} + T_{na}$$

In this inequality, T_m stands for the per-input execution time of the microservice, T_{nb} stands for the network transmission time **to** the microservice (before) and T_{na} stands for the network transmission time **from** the microservice (after).

Deep Learning Models Selection: Naturally, when speaking of executing deep learning model at the edge, there are some significant limitations. Due to the limited computational power, it is not possible to run large, state-of-the-art models as they require a substantial amount of GPU power. Many of them are too big to even be able to load in the memory of an edge device (e.g., EfficientDet-D7 is 3819MB [31]), while others can load but have very low throughput (e.g., Yolo [19] and EfficientDet-D2 [31]). Nonetheless, the goal of the system is not to provide unparalleled accuracy but rather demonstrate how edge devices can independently handle complex workloads. To this end, smaller, edge-optimised models are necessary for showcasing this concept. As edge devices are expected to grow more powerful in the future, the system would easily be able to accommodate larger, more accurate models too. Apart from the improved processing capability, advancements in the deep learning models consumes less resources while providing higher accuracy. For instance, the MobileNet model architecture from 2017 matching or exceeding the accuracy of 2014 state-of-the-art models such as VGG and GoogleNet [32], yet using just a fraction of the resources.

Supporting changing application requirements also means handling a wide variety of models. To that end, it is important to select an inference backend that handles is not only performant but also has to be compatible with different model definitions. Also key is the ability to update or add a model during runtime, ideally with minimal service interruption.

3.1.2 Performance. In order to meet the SLO and other application requirements, low latency is a must in a VA system. Additionally, the overhead of a complex, distributed architecture should also be examined. We explore those issues from both a networking and a computational perspective.

Networking: One of common concerns when evaluating microservice against monolith architectures is the additional overhead that is inevitable due to the necessity to use networking to communicate between different services. For non-realtime applications, that is not too big of a concern as a communication delays of 0.1 or 0.2 seconds are not all that noticeable in those situations. However, in a real-time VA application, this could be the difference between being able to match the SLO and having lag in the system. We discuss two important aspects when considering the network overhead: the messaging protocol and the message serialisation options.

Messaging Pattern & Protocol: Choosing a suitable messaging protocol is an important aspect of optimising the performance of the system. It is vital to keep the communication overhead to a minimum and avoid unnecessary data transfers. While using broker-based messaging patterns in the cloud is common, they are not well suited in our environment due to the unnecessary data hops they introduce. Thus, we focus our attention on direct messaging protocols, such as HTTP, ZeroMQ, gRPC.

Message Serialisation: In a real-time VA system, we often need to move relatively large amounts of data between services, as images in some intermittent states would be represented as just regular

multi-dimensional arrays. Then, there would be also a tradeoff between compressing the arrays, sending, and then decompressing compared to directly transmitting them in their raw state. Common API formats, such as JSON and XML are very inefficient for large amounts of data, as they represent everything as strings. Thus, formats that support binary data instead are better suited to represent the matrices we are working with. Examples include msgpack [33] and protocol buffers [34].

Computation: Processing real-time video feeds through a large distributes system without lagging is a key requirement. As mentioned in Section 3.1.1, no individual function call should exceed the input rate of the module it is part of. To do that, asynchronous processing is required. For instance, we have a video decoding module that receives a video frame every $T_{input} = 33ms$. If that service consists of the sequence, alongside their times: video decoding T_{vd}, frame resizing T_{fr}, matrix transposing T_{mt}, and normalisation T_n, each one of these functions is not very time consuming on their own. Then, there are two ways to execute this sequence:

(1) Synchronously - In synchronous execution, we execute each function one at a time and at any time we are running at most one of them. This means that the total decode module time would be:

$$T_{decode} = T_{vd} + T_{fr} + T_{mt} + T_n$$

Then, if $T_{decode} > T_{input}$, the component, and the entire system, will lag.

(2) Asynchronously - In asynchronous execution, on the other hand, a function moves on to the next available input as soon as it is done processing its current input. This means that we are effectively executing in parallel, and no module has to wait for everything else to complete before a function runs. The total decode module time in this case would be defined by the longest running function:

$$T_{decode} = max\{T_{vd}, T_{fr}, T_{mt}, T_n\}$$

Naturally, it would not be necessary to have each and every function executed in parallel - if two consecutive functions combined have a shorter execution time than T_{input}, then they can be executed synchronously, thus reducing the number of threads and/or co-routines.

Similar to in-module execution, looking at the larger pipeline, we should also avoid blocking network calls. For instance, once we have sent a preprocessed frame from video decoding to object detection, we should move on to the next input rather than wait until the pipeline completes its execution. Note that this is different from what VideoPipe [15] does as instead they only ever have a single frame in the pipeline at a time.

3.2 Proposed Architecture

In this section, we are going to describe the design of our architecture. First we provide an overview (Section 3.2.1), then we cover the runtime (Section 3.2.2) and modules (Section 3.2.5).

3.2.1 Overview. Borrowing from the idea of microservices, our proposed architecture attempts to segment the VA components into their own independent modules. However, apart from purely distributing each component in its own module, we go a step further and take inspiration from the concept of FaaS runtimes to be able to interact between modules on the same device without network cost [35]. Thus, we can have multiple modules per device and per "microservice".

We use the following terminology to describe the parts of our system, which we can see in Fig 2. Each device in the environment is represented as a node in the cluster. Every node has Docker [36] running as the container runtime. Then, we only have a single container hosting an application. We call these applications "runtime". Runtimes take care of running and controlling modules. Finally, modules are created within a runtime and consist of functions, which are our smallest building blocks.

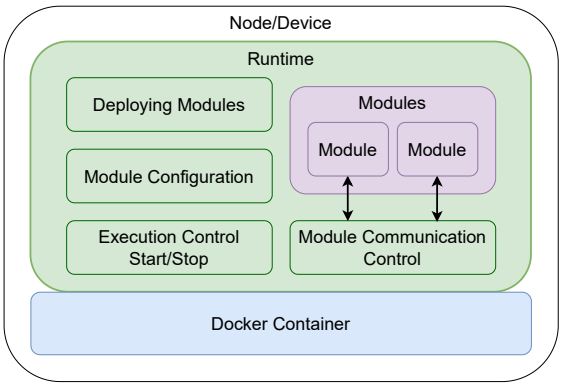


Figure 2: Overall view of a node and a runtime

3.2.2 Runtime. In a traditional microservice architecture, if there were two modules to be deployed and executed on the same node, each one of them would be running in its own container process. This separation, however, means that every time data is sent between two modules, it would have to go through networking, even if the modules are co-located on the same node. While the isolation is important for security, it is not a concern to run both in the same container in the closed-off environment which our target environment is. This is why we only run a single runtime per container so that it functions similarly to how a FaaS runtime would - we can deploy multiple modules on a single runtime. Runtimes serve the purpose of both a control and data plane through the following functionalities: deploying/configuring modules, execution control of pipelines, module communication control and hosting the modules. A representation of these can be seen in Figure 2. The colours differentiate between what is controlled by the runtime (green) and the modules (purple).

3.2.3 Control Plane. The need for a control plane stems from the fact that we have a shared runtime between modules. Due to that fact, we can not use the Kubernetes platform to specify pipeline deployments and configurations. However, due to using a custom control plane, we have more customisation options while still benefitting from Kubernetes *features* such as the general orchestration of our runtimes.

Table 3 shows a summary of the main API endpoints supported by the control plane. These endpoints are what a user can interact with to control the execution and deployment of a pipeline. Most

Function	Description
`deploy(request)`	Deploy a set of modules
`update(module, settings)`	Update an existing module
`start(pipeline_id)`	Start execution of a pipeline
`metrics()`	Gather collected metrics
`sink(module, message)`	Call next module in pipeline

Table 3: Summary of the functionality handled by the control plane

importantly, the `update` endpoint allows us to modify pipelines while they are running - for instance adding a new application to an object detection module that is already processing. The last entry, `sink()` is the the runtime function that modules call to move data forward in the pipeline.

In listing 1 we can see an example of how a deployment request looks like. It is a dictionary consisting of modules as keys, while the values represent the specific arguments to the module as well as the common arguments to the runtime. In this instance, we create a pipeline of just two elements, decoder and inference, with decoder having inference as one of its output modules. Note how the `outputs` field is a list - this is to support complex pipeline definitions where a module can have both multiple inputs and multiple outputs, rather than just a one-to-one mapping, which allows multi-tenant use-cases.

```
{
"Decoder_1": {
    "decoder_args": {...},
    "runtime_args": {
        ...,
        "outputs": ["Inference_1"]
    }
},
"Inference_1": {
    "inference_args": {...},
    "runtime_args": {...}
}}
```

Listing 1: Example request for creating two modules in a runtime. Some details were omitted for brevity.

3.2.4 Data Plane: Cross-module communication is another functionality that is handled by the runtime, simplifying and optimising the connection between modules. In Figure 3, we have the breakdown of a two-device, triple module deployment, with the first device handling two modules and the second having one. We design each module to have an *input* and an *output* queue to guarantee that no data is being dropped. During normal operation, these queues should generally be almost empty unless bottlenecked by the module. Furthermore, queues can be utilised for multiple requests together for batch processing.

The queues also act as a communication medium between the modules. The runtime reads items from the output queue of a module, which consist of two fields: `target_module` and `message`. The runtime checks if the target module is co-located in the same runtime. If yes, it proceeds to put the message into the target module's input queue. Otherwise, the message is serialised and forwarded to the runtime responsible for the target module through the use of a network call.

3.2.5 Modules and Functions. Modules are designed in such a way that the functions within a module are only the ones absolutely necessary for the module's purpose. They generally consist of only a few functions and input/output queues, as can be seen in Figure 3. Other feature such as sending object to the next module is done in the control plan. This allows the module developer to focus only on the specific implementation, without concern for the rest of the structure or flow. Because of these abstractions, we are able to satisfy our design goal of having an extensible system.

4 PROTOTYPE IMPLEMENTATION

This section provides implementation specific details for the PoC prototype of the proposed architecture. We describe the runtime and the available modules.

4.1 Runtime

For our runtime, we want the device cluster to be easily administrable, so it is structured around a Kubernetes cluster running Docker containers. Each node has just one Docker container running our Python-based runtime. The runtime uses a high-performance web server as base - gunicorn [37] with meinheld [38] worker processes. This guarantees high performance and low response times.

Both the control (Section 3.2.3) and data plane (Section 3.2.4) are running on top of this gunicorn server. The server controls user interaction with the application - requesting new services, modifying existing ones, or getting metrics. It also handles service-to-service messaging.

On top of the HTTP messaging, we use msgpack [33] as an easy-to-use format for efficient serialisation. Following shows our rationale behind our choice:

- It provides easy native support for python dictionaries due to JSON-like schema.
- There is no need to worry about handling the metadata of the arrays in terms of shape or datatype.

Guaranteeing optimal performance means making sure that no code has to wait for another function to finish execution. To this end, the network communication component of the data plane is implemented through the use of multiple threads - one for each message preparation (msgpack encoding) and message transmission. Similarly, when a message is received, it is decoded in a separate thread to not impact the rest of the application.

Within the runtime, when a new service request is received, the service is spawned as a new python process. To avoid communication overheads, the microservice python processes and the main server process communicate through the use of queues mentioned in Section 3.2.4, as illustrated in 3.

The runtimes collect various metrics regarding the performance of microservices. Most importantly, they monitor the queue sizes, which can be used to identify potential bottlenecks by observing if

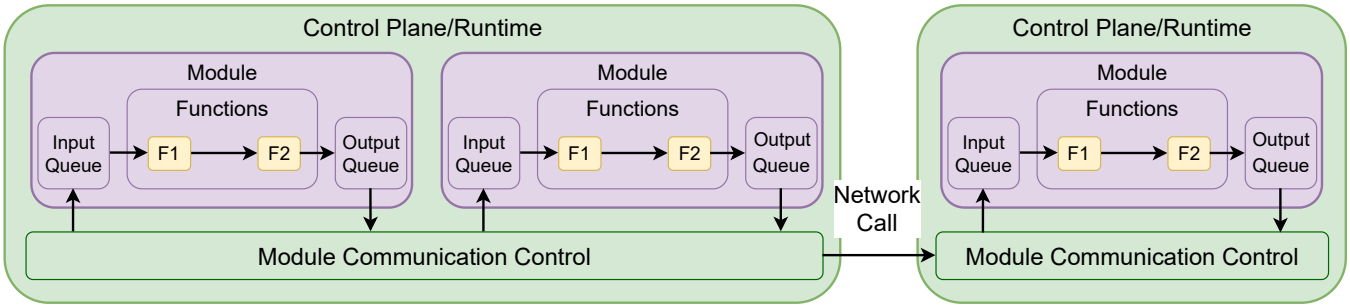


Figure 3: Module and communication control view

any queue starts to fill up. In addition, due to the common networking layer, the runtime instruments both incoming and outgoing messages with additional metadata so that both microservice execution duration and network request times can be kept track of. Through tracking frame identifiers, the runtimes across a pipeline also record the total pipeline latency for a specific application. These metrics are then exposed through the control plane.

4.2 Modules

For this prototype system, we have implemented several modules in order to support multiple application scenarios. These modules are as follows:

4.2.1 Decoder. The decoder module supports livestream video formats (RTSP) as well as decoding a saved video file. It takes a video as an input and returns the decoded frame as an output. In addition, it can optionally resize, transpose, and normalise the output frame. Supports both software-based decoding through OpenCV and hardware decoding through the use of the GStreamer in OpenCV.

4.2.2 Inference. As deep learning is a key component in many VA applications, support for a range of models and applications is vital, as discussed in 3.1.1. Thus, we use the Nvidia Triton Inference Server [30] as it supports a range of backends, and fits our flexibility design. Moreover, the server has a wide range of features regarding dynamic batch size inference and the queuing of inference requests if necessary, providing a very high level of performance, and more importantly, higher than the performance of using a deep learning libraries for inference.

We use the provided Python library to create our own inference module that fits within our runtime design. The inference module abstracts the details of using the server, so that using the module for a custom model becomes just a matter of pre- and post-processing the data. To minimise performance impacts, instead of using traditional network calls to the server, we instead use the provided option to directly register data in the GPU shared memory, avoiding sending tensors across a network interface. We implement two models: Object Detector and Reidentification.

Object Detector: The object detection module supports running any object detection model. It takes as an input the frame, already resized and preprocessed, and returns the raw output from the model. The reason for that is that as one of the heavy modules, we want to reduce its size as much as possible, so we are keeping it to just the inference part. The specific model we are using is a ResNet10 based DetectNet model by Nvidia [27]. The module also includes other auxilliary functions that it needs, such as postprocessing and cropping of the results.

Reidentification (reid): The reidentification module takes in an a list of cropped images as input (referred as gallery), and a set of template target images to compare against as as part of its configuration. Whenever the gallery is updated, the images are run through the reid model to extract the features for the gallery. Every input is compared against this feature-matrix using cosine similarity. If the top result is above a given threshold, we mark that as a successful reid match. The specific model in use is one for Person Reidentification - OSNetx1.0 [28].

4.2.3 Tracker. For the tracking module, we use the `motrackers` library [39] which performs tracking by detection. The module takes as an input a list of bounding boxes (from postprocessing module) and an optional frame, then runs the tracking algorithm on the bounding boxes. The output is the resulting tracks, or the visualised result onto the input frame. The supported trackers are CentroidTracker, Centroid Kalman Filter Tracker, SORT [40], and IOUTracker [41].

4.2.4 Region-of-Interest (RoI) Monitor. This module keeps track of the objects of a specific class appearing in a predefined RoI on screen. Takes a list of bounding boxes as an input, and a RoI definition as part of its configuration. The output is a simple log of the recorded detections within the RoI.

5 EVALUATION

In this section, the benefits of the proposed architecture are demonstrated. The following approaches are evaluated:

- Device-only monolith - This baseline refers to systems such as Distream [13] when we focus only to the device side of their architecture, applied to a multi-application scenario.
- VideoPipe [15] - VideoPipe was chosen as one of the two baselines as it is the approach closest to ours in terms of modularisation. We emulate the VideoPipe features by making two small adjustments to our implementation. First, we have a runtime per module to emulate how VideoPipe runs each module in a separate JavaScript context. Then, as VideoPipe uses synchronous processing without threads, we remove those from our system. At that point the architectures are identical, albeit with a different underlying tech stack.

- Ours (Section 4)

5.1 Experimental Setup

This section describes the environment that the experiments were conducted in, as well as the scenarios that are executed and tested.

5.1.1 Environment.

Hardware: The testbed environment where the proposed architecture was deployed consists of Jetson Nano [1] devices to serve as the edge environment. Each Jetson Nano has a Tegra System-on-a-Chip, that has a quad-core ARM Cortex-A57 CPU running at 1.5GHz, a Nvidia Maxwell-based GPU running at 921MHz with 128 CUDA cores, 4GB RAM that is shared between the CPU and the GPU, and 2GB swap file. The devices are running at their performance mode using up to 10W of power. They are all connected to the same network switch using gigabit ethernet.

Software: Each Jetson is running the Nvidia-provided operating system image, with versions 4.4.1 for Jetpack and 32.4.4 for L4T (Linux for Tegra). The devices are in a Kubernetes [42] cluster using the lightweight Rancher k3s [43] distribution. The k3s version is 1.19.7 and the Docker [36] version is 19.03.6. Docker is set as the default runtime for the cluster nodes as other runtimes (such as containerd) do not support attaching GPU to a container on the Jetson platform.

5.1.2 Scenarios. The following pipelines were used for evaluating the proposed approach. Every pipeline begins with video decoding, followed by object detection and postprocessing. We distinguish the applications by the last elements in their DAGs. We briefly describe details of the pipeline and combination choices.

- Target tracking pipeline (1 application)
- RoI monitoring pipeline (1 application)
- Person reidentification pipeline (1 application)
- Tracking + reidentification pipeline (2 applications) : Two types of application pipeline requested to the system. We've chose tracking and reidentification as the two combination because these two combination are the most resource demanding tasks (as shown in Figure 4).
- Tracking + reidentification + RoI monitoring pipeline (3 applications) : This combination shows the maximum number of application types requested to the system, which is three.

For workload distributed, we distinguish between experiments using the terms shown in Figure 1. Case A refers to the monolithic existing works, with Case B and Case C representing the two types of distribution of our proposed approach. VideoPipe refers to the approach proposed in [15], however, implemented in our system as described at the start of this section 5. For evaluation purposes, we always use the same video for the experiments, part of the Nvidia DeepStream sample applications [21]. The video is recorded from a side of the road and contains few pedestrians and multiple vehicles throughout the whole video. The input resolution is set at 1280p×720p and the framerate is 15 FPS with a duration of 6m24s. For consistent evaluation of our proposed system, we play the dataset repeatedly for each scenario. As we are recording end-to-end per-frame latencies for each approach, it is trivial to calculate how a livestream video would change the outcome. As the plots can have some noisy data, we use the 95th percentile of results and apply a moving average smoothing of 15 frames to the data for every experiment.

For the horizontally distributed approaches - Case C and VideoPipe, the experiments were run by splitting the pipelines between two Jetson devices. We run the experiment in single video mode in order to accurately reproduce the experiments conducted by the VideoPipe authors.

5.1.3 Metrics. The experiments are going to be evaluated based on the following metrics.

- End-to-end latency (s) - The end-to-end latency represents the time in seconds that a frame needs since it is first decoded from the input video stream until reaching the output of a specific pipeline. It is vital that the latency is consistently low, as otherwise increasing latency would mean the application is lagging.
- Frames per second (FPS) - The frames per second indicate how many frames can be processed by a given approach per second, showing the throughput of the scheme. We use this metric when comparing the distributed approaches - Case C vs VideoPipe. This is the case as latency is not enough to truly highlight the difference between the two, due to VideoPipe dropping frames they cannot process.
- Overhead (s) - The overhead of the distributed approaches, ours and VideoPipe, are incurring due to networking. This shows how much additional latency is added on top of the regular computation, as a microservice-approach would always have some associated networking costs compared to monoliths.

As we run experiments for multiple applications, these metrics also serve to show that our approach can facilitate the requisite flexibility for more complex scenarios, without sacrificing performance, as discussed in our design consideration section in Section 3.1. Naturally, as all metrics are tied to performance, they also address the performance considerations. While not directly measured in this work, we expect that these performance metrics also have linear correlation to energy consumption as well.

5.2 Experimental Results

5.2.1 Performance. In this section, we discuss the performance of the proposed approach. We split the evaluation depending on its execution type - single device or distributed, as well as based on the number of applications in the given experiment. Most of the results in this section are depicted as line plots of the end-to-end latency per frame, throughout the experiment to be able to observe the consistency and potential lag of the different approaches. All latency plots use the same data preprocessing: filtering to 95 percentile of results, applying a moving average smoothing to a window size of 150 points, and only showing the slowest application of the experiment, as it has the most impact on performance.

Single Device: In this section, we explore the latency performance of a single device deployment. Here, we are comparing Case A vs Case B.

Single Application: We begin with the first experiments just providing baseline performance numbers for the single application use cases. Due to the simple nature of these applications, we do not evaluate either distributed approach on them. Further, as Case

A and Case B are effectively the same approach when it comes to single application, we show combined results for both. We can see in Figure 4 that all 4 cases are does not lag, with low latency. Reid appears to be the slowest at 0.2 seconds latency per frame, while RoI and IoU taking just under 0.1 seconds, with marginal difference between them. SORT, as a more computationally expensive tracking algorithm, exhibits slightly higher latency but still faster than Reid.

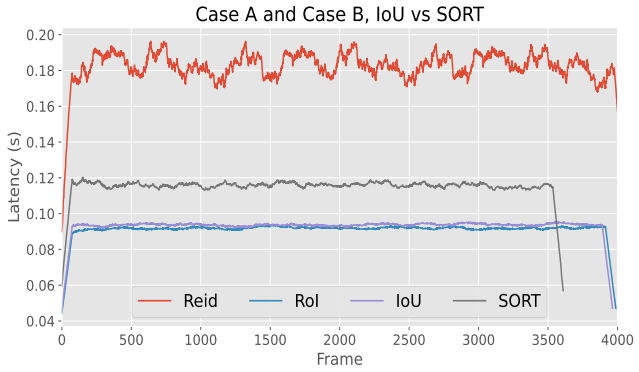


Figure 4: Single application latency, Cases A and B

Dual Application: In the dual application test, we evaluate Case A and Case B approaches using 2 dual application pipelines. The applications are Reid and Tracking, with Tracking using SORT as it performs slower than the IouTracker. Results are shown in Figure 5

For Case A, most of its applications are behaving similarly to the single application scenario. The Reid app exhibits stable latency but with performance is about 5 to 10% worse than in the single application case, as we have more to process. The main difference is in the tracking application that has extremely high latency and reaches more than 55 seconds of delay. Given the rising trajectory, it is clear that the application is lagging. This is the first sign of the limitations of naively putting multiple applications on the same device, without sharing parts of the pipeline.

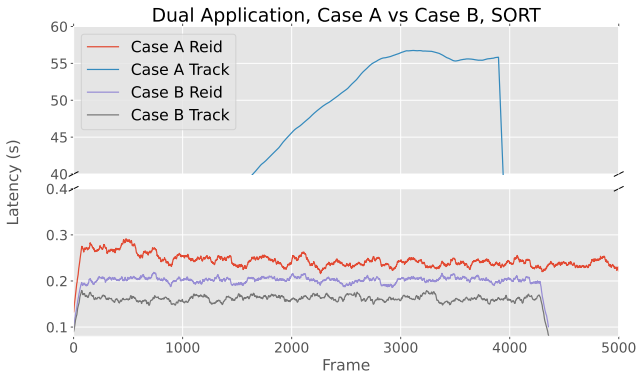


Figure 5: Dual application latency, Cases A and B

On the other hand, for Case B, we observe quite a different story for the Case A approach. All of the results are consistent, with no sharply rising latencies. Reid is marginally slower than its single-app counterpart from Figure 4. Importantly, the tracking app is also stable and is not lagging. This is due to the ability to share computation between pipelines, and more specifically, avoiding the duplicate work.

Triple Application: In this experiment, we run the triple application pipeline, consisting of tracking, RoI monitoring, and person reidentification, once again showing results for the SORT tracker as the slower option. In Figure 6 we can see the results of running this experiment using Case A. Note that this time we use a logarithmic scale for the y axis of the monolith experiment. This is due to all of the applications experiencing spikes in latency. The culprit for this, similarly to the previous experiments, is the computational cost of running the SORT algorithm but also the repeated execution of Object Detection for every pipeline. As the three applications are running simultaneously, having one of them occupy the system resources means that there is not enough resources left for the other applications to run correctly. This is why we see a form of waterfall-effect where the SORT tracking application causes both Reid and RoI to also lag. Once again, the increase in delay is only until the queues fill up, at which point the method starts dropping frames and reaching a sort of equilibrium at the cost of those dropped frames.

Once again, Case B behaves very differently, as can be seen in Figure 6. It suffers no lag despite the triple application workload. If anything, the results that it provides are closer to running the same case with dual application workload (Fig 5) than Case A with triple applications. Once more, the Reid and Tracking applications are on the slower side but still well within reason. In fact, the differences between dual and triple application are minimal, which serves to show both the scalability and flexibility of this approach.

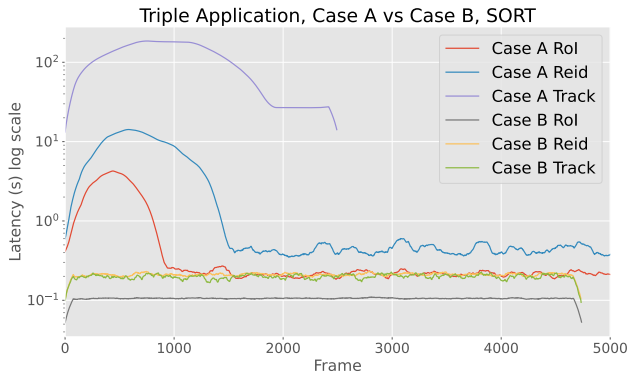


Figure 6: Triple application latency, Cases A and B

Distributed pipeline: In this section we evaluate Case C against VideoPipe. As a single application pipeline does not have enough components to benefit from distribution, we only explore dual and triple application cases. The pipelines are distributed across two devices, running on a single input video to replicate VideoPipe's evaluation. Once again, we focus only on the slowest application in a pipeline, as it is most telling of the approach's performance. In this case, this is the Reid app. We also focus on only tracking using SORT, as when we have two devices to process a single video, we

can afford to use the more accurate option. We can see the results in Figure 7.

Taking a look at the results of VideoPipe, we see two main trends. Firstly, the latency of both dual and triple application cases seems to be stable throughout the experiment, with no lag. The average latencies seem to be hovering around 0.25 seconds, which is slightly above the values for dual and triple application using single approach - the difference comes from the additional overhead from networking calls. The second trend, expectedly, is that the triple application scenario is slightly slower overall than the dual application one. This is only natural given that there is more workload for the devices to handle.

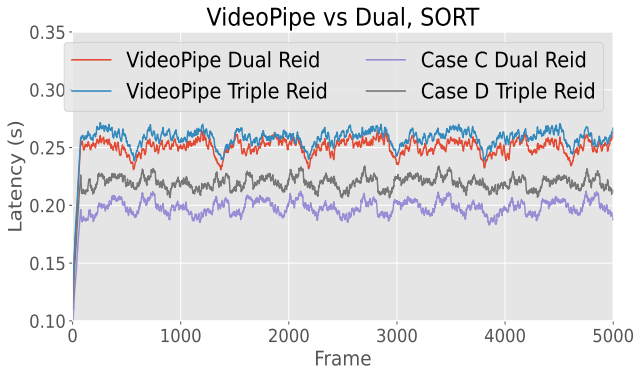


Figure 7: Distributed application latency, Cases C and VideoPipe

Moving on to Case C, at first glance it seems quite similar. There is no apparent lag, and the triple application is also slightly slower then dual application. The main difference is that the entirety of the results are shifted down in latency by about 0.05 seconds. This roughly 20% difference comes from the avoidance of network calls when two modules are located on the same device.

However, these charts do not show the full story. Due to the difference in execution - VideoPipe being synchronous, while our approach is asynchronous - the latency values have different meanings. Yes, the end-to-end latency is the same but that is not the only important metric to consider in a VA application. Where the difference between the two approaches is much more stark is the throughput. This can be seen in Table 4. With the synchronous processing of VideoPipe, only one module, on one device is actively processing at a time. Thus, any frame that arrives while another frame is still running through the pipeline is dropped at the start. While this means that the approach can never lag per se, it does so at the cost of potentially dropping a significant amount of information. The latency in that case is directly correlated to the throughput of the system and we can see that in the table, with the FPS being between 3.7 and 3.82 for dual and triple application, respectively. This is equivalent to losing approximately 75% of the incoming frames.

On the other hand, due to the non-block execution of our approach, our latency serves as an indication as to whether the application is lagging or not. However, its time is not directly correlated to the throughput of the system. As we can see in Table 4, for both

Approach	Dual Reid FPS	Triple Reid FPS
VideoPipe	3.82	3.70
Ours	15	15

Table 4: Throughput of distributed approaches

dual and triple application case, we have a throughput of 15 FPS, regardless of the differences in latency. This is a consequence of asynchrony - we could have latencies of multiple seconds, but, as long as they are consistent and not showing lag, the throughput would remain as high as the input video rate. Ultimately, this means we have both lower latency and higher throughput through purely architectural decisions.

5.2.2 Overhead. In this experiment, we consider the final one of the design considerations - architectural overhead. As we are distributing a set of modules across devices, we will incur transfer costs. Those costs are expressed in three steps: serialise a message, send it over the network, deserialise the message. For the purposes of quantifying and visualising the overhead, we consider the following experiment. We take a single application person reidentification that we split across two devices. For input, we are using a single video to exclude web server load as a possible factor and instead focus only on the concepts at hand, namely our approach compared to VideoPipe. We evaluate by measuring the cumulative overhead of the executed application, as an input is moving across the pipeline. The cumulative overhead at a given element is calculated as follows:

Given n modules, indexed as $(0..n)$, the values $(TO_0..TO_n)$ represent the time to move an input from the previous to the current module, i.e., the module's overhead. The overhead of module n is determined as follows:

$$TO_n = T_s + T_N + T_d$$

where T_s stands for the serialisation time, T_N for network send time, and T_d for deserialisaiton time. Thus, the cumulative overhead at element n is:

$$TCO_n = \sum_{m=0}^{n} TO_m$$

The results are shown in Figure 8. The bars represent the cumulative latency at each stage in four-element pipeline. The middle vertical line depicts where the data has to move from the first device to the second one. In each group of 2 columns, the left one shows the overhead of our approach while the right one shows VideoPipe. The stacked bars also show the breakdown of overhead as composed of network time and serialisation/deserialisation. There are two main things to observe. As VideoPipe places each module in its own runtime, they have to use network requests even between two modules that are running on the same device. This means that there is an increase in overhead between any two modules, with a slightly larger increase when the network call has to be done to another device (Postprocess to Crop). On the other hand, our use of a singular runtime for modules co-located on the same device means we can completely avoid unnecessary network calls, as described in 3.2.4. This is why we have only the minimal overhead of the runtime when staying in the same device, and the regular network

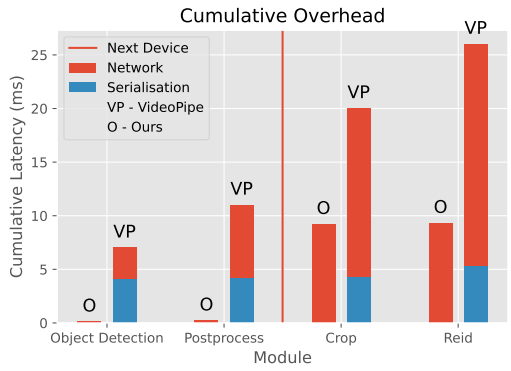


Figure 8: Latency overhead of our approach compared to VideoPipe. Horizontal line depicts the boundary between devices 1 and 2.

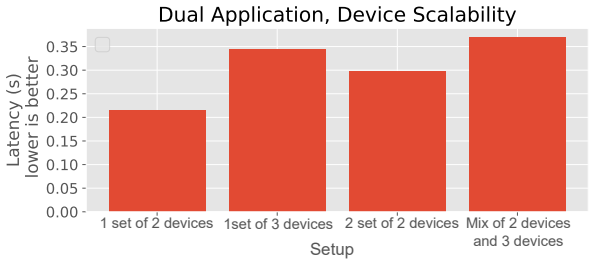


Figure 9: Scalability experiment, ranging the number of devices from 2 to 5, using a dual application scenario.

call when proceeding to a different device. While the exhibited overheads, even by VideoPipe, are only reaching up to 25ms, it should be noted that this is best-case scenario. With the addition of more modules or more inputs, these values would only grow, further degrading the performance.

5.3 Discussions

5.3.1 Flexibility. One of the focal points of our approach is of course the flexibility. By showing the minimal performance impact of adding more applications, we do numerically address one aspect of the flexibility of our architecture. Nonetheless, some aspects are not possible to represented through evaluation - for instance, the range of possible configurations and supported deployments. Our design choices lend themselves to a wider gamut of options for pipeline distribution. In fact, it allows us to not only emulate the architecture of both VideoPipe and EdgeEye by just setting a different configuration upon pipeline deployment but also do that on-the-fly, without stopping or restarting the running applications. How VideoPipe can be represented through our approach has already been discussed in 5. Whereas VideoPipe co-locates their modules and services on the same device, EdgeEye has the modules on the edge devices while the services are on an edge server. Given that our architecture has a clear distinction between control and data plane, it is trivial to separate a pipeline in a manner that mimics EdgeEye. This ability to implement a range of existing architectures is one of the main strengths of our approach. While the existing works are also modular, their design decisions in terms of networking and synchronous processing mean they are not as flexible as our approach.

5.3.2 Scalability. While flexibility is the main purpose of this work, we also explore the potential scalability of our approach when applying the distributed processing to multiple devices. To that end, we run a dual application scenario consisting of Reid and Tracking applications using 2 to 5 devices. As each device represents a camera, this means that we have an equivalent number of input videos. What we see in Figure 9 is the average latency of the slowest application in each case. We can see that by increasing the number of devices, the latency also grows - this is mostly due to the associated additional network cost, similarly to Figure 8.

The reason behind that is the inherent difficultly of placing the components in a way that is most efficient without prior knowledge of the performance of allocated module on a device. In this work, we manually distribute the modules on multiple devices. For instance, in the triple device case, we have each device running one object detection, tracking, and reid on the three inputs it has. This is nearly at the limit of the detection and reid models performance. For our 4 device situation, a distribution of 2 sets of 2 device combination alleviates the performance bottleneck. Yet, moving up to 5 effectively means using a combination of 2 and 3 device partitioning, thus the slowest results being very close to that.

These results tell us that the architecture allows for scalability but would significantly benefit from a more sophisticated service allocation approach, perhaps handled by a centralised control plane. Apart from allocation, this control plane could also serve as a supervisory monitoring and adjustment of the running applications to meet their requirements.

For our future work, we plan to devise a module allocation scheme which profiles and matches the edge resources and the service request, similar to the work done in [22, 23] so that the system accommodates support for application-level flexibility as well. Furthermore, we intend to leverage various joint learning techniques [44] to train configuration policy learned from multiple edge environments.

6 RELATED WORK

6.1 Collaborative VA

While our work mainly focuses on the architectural design to support VA at edge, research community has put effort to prune out large volume of unessential video frames for the application that it serves.

Recent work leverage smart filtering techniques to drop *uninsteresting* video frames early on from the edge device so that these frames are not feed to the edge server for further processing. The insight is that, in majority of the video frames, objects-of-interest is not present. Therefore large amount of CPU/GPU resource can potentially be wasted if all frames where to be inspected by a resource hungry DNN based object detection model. For instance,

Vigil [45] leverages a lightweight object detector for person counting to decide more essential frames from the camera cluster. Similarly, authors of FilterForward [46] designs micro filters to filter frames with object's feature vectors on given input frame. Wang et al., [47] propose adaptive sampling technique for AR device to dynamically adjusts the sampling rate of video depending on the type of application it serves. Glimpse [48] and Reducto [25] propose filtering mechanism based on how much frame there are between consecutive frames, as no severe content change may happen in few milliseconds. These aforementioned techniques are orthogonal to our work, but when adopted in our architecture, optimisation techniques may also reduce bandwidth costs.

6.2 Serverless at Edge

As we draw inspiration from serverless computing, it should be noted that works using serverless computing propose a variety of interesting approaches for edge computing. The authors of [49] apply the serverless concept to an edge computing network. The works in [50, 51] propose systems that focus around scalability and abstracting the underlying platform's complexity away from the user. However, due to the reliance and focus on serverless functions, those platforms are designed for applications with short, bursty workloads, instead of the continuous heavy workload that is presented by real-time video analytics.

Other works are instead focused around the container runtimes and optimising those for the lower performance edge devices. In [52], the authors trial a system that allows for better sharing of libraries and resources between containers, with the goal to reduce the size of the containers and improve their startup times. Similarly, [53] proposes a new, edge-first FaaS platform that takes the environment's requirements into consideration. Instead of containers, they apply WebAssembly as the execution platform and find preliminary results to be promising in terms of start-up latency.

There are some works in serverless that do specifically target continuous or real-time applications as well. On the cloud level, [16] is a serverless functions framework based on AWS Lambda. It proposes a similar approach to design functions as reusable components between a variety of video processing pipelines. However, it lacks deep learning model support and is designed for offline processing where it can benefit from parallel processing but would be unable to support real-time videos.

At the edge, an approach similar to ours is considered by [54] as they apply microservices to a cluster of Raspberry Pi edge nodes to distribute a GStreamer-based video processing pipeline. Yet, their solution is not designed to support either real-time processing or deep learning models. The authors of [55] attempt to support real-time applications through the OpenFaaS serverless framework by extending the scheduling to support more complex definitions with regard to what bursts an application should support. Their work, however, is only evaluated on powerful edge servers and does not support the sharing of components across pipelines.

7 CONCLUSION

In this paper we describe a two-tier solution in order to address the important issue of supporting multi-tenant, multi-application real-time VA use cases on PoC testbed which consists of a fleet of smart cameras. Both aspects of the solution are inspired from serverless concepts successfully applied in the cloud - serverless functions and microservices architecture. First, we define a way to co-locate video pipeline components on the same device, allowing them to share existing modules in order to reduce unnecessary calculations. Through our experiments, we show that this helps to both significantly improve performance and reduce resource consumption and contention at device-level. Second, we extend the locally shared runtimes idea to also support distributed execution of video pipelines across other edge devices. We compare this approach to a well-known existing horizontal-splitting approach and find that despite architectural similarities our approach significantly outperforms the existing work. Ultimately, we believe this paper achieves its goal of showing the promise of microservices in a real-time scenario of processing edge VA.

ACKNOWLEDGEMENT

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01126, Self-learning based Autonomic IoT Edge Computing)

REFERENCES

[1] 2015. Jetson nano developer kit. (2015). https://developer.nvidia.com/embedded/jetson-nano-developer-kit.
[2] 2017. Jetson tx2 module. (2017). https://developer.nvidia.com/embedded/jetson-tx2.
[3] 2021. Kirin 990 5G. (2021). https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-990-5G.
[4] 2021. Kirin 9000. (2021). https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000.
[5] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-time video analytics: the killer app for edge computing. *computer*, 50, 10, 58–67.
[6] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: an approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 123–136.
[7] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*.
[8] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.
[9] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 253–266.
[10] 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (NSDI'17). USENIX Association, USA, 377–392.
[11] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. 2017. Lavea: latency-aware video analytics on edge computing platform, 1–13.
[12] Chien Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. VideoEdge: Processing camera streams using hierarchical clusters. In *Proceedings - 2018 3rd ACM/IEEE Symposium on Edge Computing, SEC 2018*. Institute of Electrical and Electronics Engineers Inc., (December 2018), 115–131. ISBN: 9781538694459. DOI: 10.1109/SEC.2018.00016.
[13] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. 2020. Distream: Scaling live video analytics with workload-adaptive distributed edge intelligence. In *SenSys 2020 - Proceedings of the 2020 18th ACM Conference on Embedded Networked Sensor Systems*. Association for Computing Machinery, Inc, (November 2020), 409–421. ISBN: 9781450375900. DOI: 10.1145/3384419.3430721.
[14] Peng Liu, Bozhao Qi, and Suman Banerjee. 2018. EdgeEye - An edge service framework for real-time intelligent video analytics. In *EdgeSys 2018 - Proceedings of the 1st ACM International Workshop on Edge Systems, Analytics and*

Networking, Part of MobiSys 2018. Association for Computing Machinery, Inc, (June 2018), 1–6. ISBN: 9781450358378. DOI: 10.1145/3213344.3213345.

[15] Mohammad Salehe, Zhiming Hu, Seyed Hossein Mortazavi, Tim Capes, and Iqbal Mohomed. 2019. VideoPipe: Building video stream processing pipelines at the edge. In *Middleware Industry 2019 - Proceedings of the 2019 20th International Middleware Conference Industrial Track, Part of Middleware 2019*. Association for Computing Machinery, Inc, (December 2019), 43–49. ISBN: 9781450370417. DOI: 10.1145/3366626.3368131.

[16] Lixiang Ao, Geoffrey M. Voelker, Liz Izhikevich, and George Porter. 2018. Sprocket: A serverless video processing framework. In *SoCC 2018 - Proceedings of the 2018 ACM Symposium on Cloud Computing*. Association for Computing Machinery, Inc, (October 2018), 263–274. ISBN: 9781450360111. DOI: 10.1145/3267809.3267815.

[17] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines, (February 2021). http://arxiv.org/abs/2102.01887.

[18] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. 2019. Hetero-edge: orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1270–1278.

[19] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 779–788.

[20] 2001. Gstreamer framework. (2001). https://gstreamer.freedesktop.org/.

[21] 2021. NVIDIA DeepStream SDK | NVIDIA Developer. (2021). https://developer.nvidia.com/deepstream-sdk.

[22] Jayson G Boubin, Naveen TR Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. 2019. Managing edge resources for fully autonomous aerial systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 74–87.

[23] Xingyu Zhou, Robert Canady, Shunxing Bao, and Aniruddha Gokhale. 2020. Cost-effective hardware accelerator recommendation for edge computing. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[24] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. 2018. Deepdecision: a mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1421–1429.

[25] 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 *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 359–376.

[26] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. 2018. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 159–173.

[27] 2021. DetectNet: Deep Neural Network for Object Detection in DIGITS | NVIDIA Developer Blog. (2021). https://developer.nvidia.com/blog/detectnet-deep-neural-network-object-detection-digits/.

[28] Kaiyang Zhou, Yongxin Yang, Andrea Cavallaro, and Tao Xiang. 2019. Omni-scale feature learning for person re-identification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 3702–3712.

[29] 2021. NVIDIA TensorRT | NVIDIA Developer. (2021). https://developer.nvidia.com/tensorrt.

[30] 2021. Triton Inference Server. (2021). https://developer.nvidia.com/nvidia-triton-inference-server.

[31] Mingxing Tan, Ruoming Pang, and Quoc V Le. 2020. Efficientdet: scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 10781–10790.

[32] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[33] Sadayuki Furuhashi. 2008. MessagePack. (2008). https://msgpack.org/.

[34] 2001. Protocol Buffers. (2001). https://developers.google.com/protocol-buffers.

[35] Christopher Stewart and Kai Shen. 2005. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 71–84.

[36] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014, 239, 2.

[37] 2010. Gunicorn - Python WSGI HTTP Server for UNIX. (2010). https://gunicorn.org/.

[38] 2011. Meinheld. (2011). https://meinheld.org/.

[39] Aditya M Deshpande. 2020. Multi-object trackers in Python. (2020). https://adipandas.github.io/multi-object-tracker/.

[40] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. 2016. Simple online and realtime tracking. In *2016 IEEE international conference on image processing (ICIP)*. IEEE, 3464–3468.

[41] Erik Bochinski, Volker Eiselein, and Thomas Sikora. 2017. High-Speed tracking-by-detection without using image information. In *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2017*. Institute of Electrical and Electronics Engineers Inc., (October 2017). ISBN: 9781538629390. DOI: 10.1109/AVSS.2017.8078516.

[42] 2014. Kubernetes. (2014). https://kubernetes.io/.

[43] Rancher. 2021. K3s: Lightweight Kubernetes. (2021). https://k3s.io/.

[44] Shalisha Witherspoon, Dean Steuer, Graham Bent, and Nirmit Desai. 2020. Seec: semantic vector federation across edge computing environments. *arXiv preprint arXiv:2008.13298*.

[45] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 426–438.

[46] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya R Dulloor. 2019. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*.

[47] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2019. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 152–165.

[48] Tiffany Yu-han Chen. 2015. Glimpse : Continuous , Real-Time Object Recognition on Mobile Devices Categories and Subject Descriptors. *SenSys '15 Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, 155–168. DOI: 10.1145/2809695.2809711.

[49] Alex Glikson, Stefan Nastic, and Schahram Dustdar. 2017. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, 1–1.

[50] Luciano Baresi and Danilo Filgueira Mendonca. 2019. Towards a serverless platform for edge computing. *Proceedings - 2019 IEEE International Conference on Fog Computing, ICFC 2019*, 1–10. DOI: 10.1109/ICFC.2019.00008.

[51] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. 2019. Towards a Serverless Platform for Edge AI. In *HotEdge*.

[52] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. 2020. Toward lighter containers for the edge. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[53] Tobias Pfandzelter and David Bermbach. 2020. TinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *Proceedings - 2020 IEEE International Conference on Fog Computing, ICFC 2020*. ISBN: 9781728110868. DOI: 10.1109/ICFC49376.2020.00011.

[54] Olivier Barais, Johann Bourcier, Yerom David Bromberg, and Christophe Dion. 2016. Towards microservices architecture to transcode videos in the large at low costs. In *2016 International Conference on Telecommunications and Multimedia, TEMU 2016*. Institute of Electrical and Electronics Engineers Inc., (August 2016), 69–74. ISBN: 9781467384094. DOI: 10.1109/TEMU.2016.7551918.

[55] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-time Serverless: Enabling application performance guarantees. In *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*. ISBN: 9781450370387. DOI: 10.1145/3366623.3368133.