VideoJam: Self-Balancing Architecture for Live Video Analytics

Youssouph Faye, Francescomaria Faticanti, Shubham Jain, Francesco Bronzino

VideoJam: Self-Balancing Architecture for Live Video Analytics

Youssouph Faye[1], Francescomaria Faticanti[2], Shubham Jain[3], Francesco Bronzino[2]
[1]Université Savoie Mont Blanc, LISTC
[2]ENS de Lyon, CNRS, Université Claude Bernard Lyon 1, LIP, UMR 5668
[3]Stony Brook University

***Abstract*—Edge-based live video analytics are a promising approach to reduce bandwidth overheads caused by the transmission of raw video streams to the cloud. However, the limited resources available on edge devices make it challenging to successfully process video streams in real-time. This gets further exacerbated when attempting to process video streams from mobile cameras. While mobile cameras are a desirable source of information, thanks to them being in the right place at the right time, they are inherently dynamic and unpredictable. To address these challenges, we propose `VideoJam`, a decentralized load balancing solution for live video analytics. `VideoJam` uses a set of load balancers to balance incoming video traffic across replicas without the need of centralized coordination. Exploiting the inherent load dynamicity generated by different video sources, `VideoJam` predicts the incoming load for each processing component and offloads excessive traffic to less-loaded neighbors. Further, `VideoJam` operates independently of deployed configurations and cameras present in the system, dynamically adapting to handle load changes and balance video traffic across available resources. Our evaluation shows that `VideoJam` can adapt to different mixes of mobile and fixed cameras, as well as quickly adapting to configuration changes occurring at runtime. Compared to state-of-the-art solutions, `VideoJam` achieves 2.91× lower response time, while reducing video data loss by more than 4.64× and generating lower bandwidth overheads.**

***Index Terms*—Video Analytics, Load balancing, Distributed System.**

I. Introduction

Video camera flows are a pervasive source of information. Cities are increasingly deploying closed-circuit cameras that are used for safety, security, and traffic control applications [1], [2], [3]. To process the incoming videos streams in real-time, live video analytics architectures have been proposed to support a multitude of applications across navigation, safety, and control [4], [5], [6], [7], [8]. However, the increasing amounts of video data produced by available cameras has forced the community to move away from centralized cloud-based architectures [9], [4].

Edge-based live video analytics are a promising approach to reduce bandwidth overheads caused by the transmission of raw video streams to centralized clouds [10]. Unfortunately, in contrast with the quasi-infinite resources of data-centers, edge devices are often co-located with existing network equipment and deploy limited computational resources. As a result, they can rapidly become overloaded by the incoming video frames, causing data loss and reduced accuracy [7], [11]. To address this challenge, various solutions have been proposed to distribute the workload across locations, ranging from vertically splitting the processing between edge devices and the central cloud, wherein excess traffic is offloaded to the cloud [12], [13] to horizontally across different edge clusters, exploiting the dynamicity of processing requests incurred at each location [11], [14]. However, all these architectures assume that video flows are generated from fixed cameras (*e.g.*, traffic cameras deployed at street corners) and that their workflows present predictable patterns [7].

With the rising penetration of mobile cameras, an opportunity emerges to include these cameras into the design of video analytics architectures. Mobile devices, such as cars or drones, come equipped with high quality camera sensors that have been demonstrated to be very effective for navigation, safety, and control applications [15], [3]. These cameras often have the unique advantage of being *in the right place at the right time*. However, the scenes that mobile cameras capture vary more rapidly than for fixed camera.

Owing to the high mobility and dynamic content of mobile cameras, we identify three key challenges in accommodating these cameras in video analytics architectures. First, the workload generated by mobile cameras is more dynamic and unpredictable, requiring constant adjustments to the processing infrastructure. Existing solutions such as Distream [6] recognize the need for adaptation to varying processing loads, but ultimately fall short of developing a solution that adjusts at the rate imposed by mobile cameras (we expose such problem in detail in Section V). Second, the continuously changing scenes captured by mobile cameras make customary processing pipelines ineffective. For example, a typical pipeline used to process a video feed incoming from a fixed camera might employ a lightweight background subtractor to isolate moving objects, reducing the need for deploying more expensive object detection modules [6]. Finally, as mobile cameras appear and disappear from the deployment, they generate constant changes in the deployment configuration and the number of sources to process. This introduces the need to deploy different processing pipelines for different cameras. Overall, existing video analytics architectures are ill equipped to handle such video traffic.

In this paper, we tackle the challenge of designing a video analytics architecture capable of handling a mix of fixed and mobile video camera flows. We build on the recent idea [6] that

the dynamicity of the workload generated by video cameras can be exploited to balance the load across different edge clusters. However, we observe that: (1) While highly dynamic, the workload generated by mobile cameras is not completely unpredictable. In fact, the scenes captured by mobile cameras reveal sufficient patterns to make it possible to predict the amount of objects that will need to be processed in the next few seconds. (2) Enabling the coexistence of diverse video analytics pipelines requires dividing the processing problem into smaller components that function independently, even when pipelines later converge to the same set of modules. (3) The constant changes in the deployment configuration, such as the addition or removal of processing components, require an online approach to load balancing that can adapt to these changes without requiring a complete reboot of the processing pipelines.

We leverage these observations to build `VideoJam`, a load balancing solution for live video analytics. `VideoJam` deploys a set of load balancers co-located with every task in the analytics pipeline. Each load balancer monitors the incoming flow of frames or objects to process and periodically shares this information with its neighbors. Based on the information collected, the load balancers take independent decisions on how much traffic to process locally and whether to offload some of their workload to less-loaded neighbors. To make these decisions, `VideoJam` uses a lightweight machine learning model to predict the incoming workload for each processing component for the near future, as well as for its neighbors. Finally, the load balancers recover to eventual prediction errors via a congestion prevention signalling system. `VideoJam` operates independently of deployed configurations and dynamically adapts to handle eventual changes (*e.g.*, new camera arrivals or departures) without requiring hard reboots balancing incoming traffic accordingly.

We implement and evaluate `VideoJam` using a typical vehicular safety application, *i.e.*, vehicle plate detection, adapted to both fixed and mobile cameras. Our evaluation shows that `VideoJam` can achieve a 2.91× lower response time and reduce 4.64× video frames loss than a state-of-the-art approach [6], while also reducing network overheads. Further, we demonstrate the ability of `VideoJam` to dynamically adapt to changes in the deployment configuration, *i.e.*, change in number of function replicas or number of cameras in the system, without requiring any hard reboots. `VideoJam` adapts to the new configuration in less than 28 seconds (*i.e.*, +27% the failure time), to the incurred change. We release `VideoJam` as open source software for the community to use and extend.

II. Background and Motivation

In this section we present the characteristics of video analytics pipelines, providing motivation for integrating mobile video sources into the analytics architecture. We then present the key differences between fixed and mobile cameras, highlighting the challenges that we face when we integrate different types of cameras into such systems.

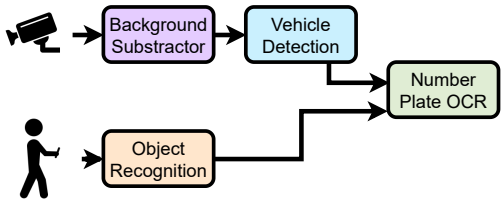


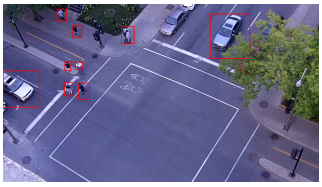
Fig. 1: The functions composing a vehicles' number plate detection application pipeline. Different sources require different processing functions.

A. Live Video Analytics

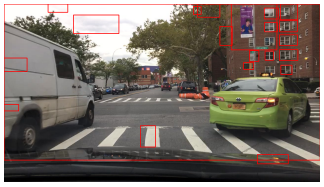
Live video analytics center around analyzing video camera streams in real-time, using algorithmic and computer vision techniques to extract valuable information. Incoming frames traverse a series of modules (or *functions*) that perform different tasks, such as object detection, classification, and tracking. These functions are combined into a pipeline, conventionally represented by a directed acyclic graph, where the output of one function is the input of the next and the final output depends on the analytics application deployed. Figure 1 shows an example of video analytics pipelines for a typical traffic control application: vehicles' number plate detection.

Historically, video analytics research has focused its attention on the placement problem occurring at deployment time. Deploying a video analytics application involves taking an orchestration decision of where to place the instances of the application's pipeline. Placement decisions are taken based on the available resources in the compute infrastructure and the workload generated by available cameras. Early work on video analytics focused on how to efficiently transmit video traffic to centralized clouds for processing [5], [8], [16]. Yet, while centralized datacenters offer unbounded compute resources, transporting the increasing amounts of video streams to these locations can result in network bottlenecks, requiring either to preprocess video frames on premises [13], [12] or to reduce the quality of the video transported [17], potentially affecting the performance of deployed applications.

To combat these challenges, recent work has focused on deploying video analytics pipelines at the edge of the network [6], [7], [14]. These approaches aim to take advantage of compute resources deployed close to video cameras and bypass the transmission to remote locations. However, edge compute devices are often co-located with the existing network equipment and deploy limited computational resources. As a result, they can rapidly become overloaded by incoming video frames causing data loss and reduced accuracy. To address this challenge, different solutions have been proposed to distribute the workload across locations. For example, Distream [6] exploits the inherent load dynamics present in video flows to split the processing pipeline between two hierarchical locations, *i.e.*, the camera and the edge compute machine. Chameleon [7] instead optimizes the pipelines configuration based on temporal and spatial predictions on the contents of camera sources.

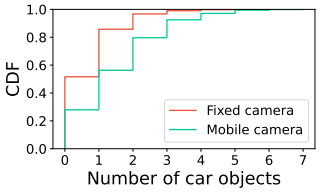


(a) Fixed camera with background subtractor.

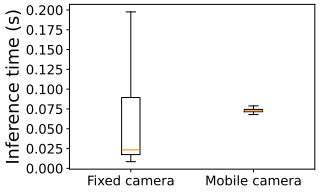


(b) Mobile camera with background subtractor.

Fig. 2: A comparative example on the performance of background substractor functions on fixed and mobile cameras.



(a) Number of detected vehicles.



(b) Inference time.

Fig. 3: Vehicle detection performance for fixed (background substractor + detection) vs mobile (YOLOv5) cameras.

Yet, all these architectures assume that video flows are generated from fixed cameras (*e.g.*, traffic cameras deployed at street corners) and that their workflows present predictable patterns allowing reconfiguration decisions to be taken at longer timescales, for example when traffic conditions change during the day due to commute patterns. However, mobile cameras have become pervasive in the last decade. Mobile devices, from smartphones to cars and drones all come equipped with high quality camera sensors. These cameras often have the unique advantage of being in the right place at the right time, offering the potential to enhance existing architectures and improve application performance. Unfortunately, video feeds generated by mobile cameras are fundamentally different from fixed camera ones, posing unique challenges into the path for their integration.

B. Challenges in Incorporating Mobile Cameras

Mobile cameras bring the advantage of providing a unique perspective on the captured scenes. They can be in the right place at the right time, capturing scenes that would be otherwise invisible from fixed cameras. However, this benefit comes at the implicit cost of having to handle fundamentally different dynamics. Mobile cameras are constantly moving, capturing new scenes; they can appear and disappear from a deployment; and the scenes they capture can vary more rapidly than for fixed cameras. For these reasons, the differences between fixed and mobile cameras can greatly impact the deployment choices of a video analytics pipeline architecture that aims to process their video feeds. With the goal of designing a video analytics architecture that can handle both fixed and mobile cameras, we identify three core challenges that arise from the coexistence of these cameras in the same deployment.

Challenge #1: Heterogeneous performance profiles. The majority of existing video analytics solutions [7] assume the homogeneity in the performance of the processing components deployed. Distream [6] relaxes this assumption by considering the presence of heterogeneous processing devices and accounts for this disparity in implementing load balancing policies within its architecture. However, mobile cameras are inherently in constant movement, quickly capturing new scenes. This makes the processing pipelines that are effective for fixed cameras ineffective for mobile cameras.

To exemplify the difference between fixed and mobile cameras, we consider a vehicles' number plate detection application. The goal of this application consists of detecting vehicles from a camera frame using a vehicle detection module and extract their number plate via an Optical Character Recognition (OCR) module. To lighten the load of the processing pipeline, the first step involves isolating objects in the frame to limit vehicle detection executions solely on cropped images [6]. This is typically done using a background subtractor module that compares the current frame with a background model and outputs a mask of the foreground objects [6]. Figure 2a shows the output of a background subtractor module applied to a fixed camera frame. We observe that the background subtractor is able to isolate the moving objects in the frame while still objects (*e.g.*, parked cars) are not detected as they are still in the frame. Unfortunately, while the application of a background subtractor is effective for fixed camera streams, the same pipeline becomes ineffective for mobile cameras. By nature, mobile cameras are constantly moving, capturing new scenes. As a consequence, when applied to a moving subject, the subtractor model is not capable of adapting to new scenes, causing erroneous detections or, in the worst case, detecting the entire frame as foreground (as shown in Figure 2b).

To compensate for the degraded performance of the background substractor module, existing solutions tailored for mobile cameras [18], [19] replace the early stages of the pipeline with an object recognition module (*e.g.*, YOLO [20]) that is capable of detecting and classifying objects in the frame in a single operation. However, this comes at a price, as detection models are more resource-intensive, especially for scenarios where no object is in the frame. Figure 3 shows the performance difference between the two approaches on two selected videos, a fixed and mobile one. We observe that, while the number of detected vehicles is relatively similar across videos, the characteristics of the inference times for the two approaches highly varies depending on the processed frame. In fact, while YOLO performs consistently due to its single pass nature, the fixed camera pipeline's performance varies based on the number vehicles present in the frame. Ultimately, this leads to the conclusion that the two approaches should not be treated as interchangeable and that the choice of the pipeline to deploy should be tailored to the video source type.

Challenge #2: Highly variable workloads. Previous work has highlighted how video camera feeds differ in the amount of objects of interest they capture depending on their deployment location (*e.g.*, a building entrance vs. an emergency exit) and

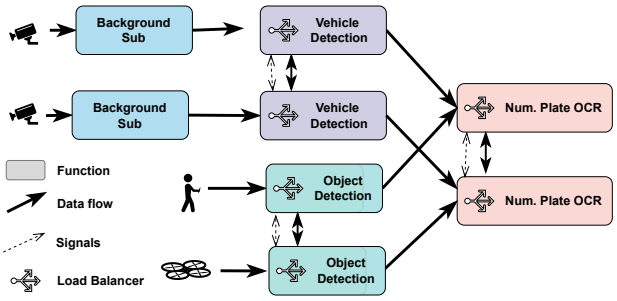


Fig. 4: VideoJam architecture.

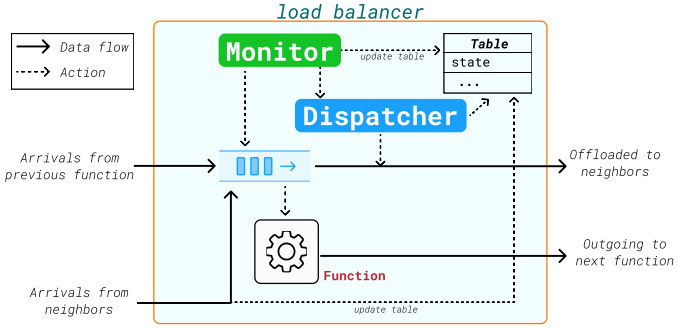


Fig. 5: VideoJam's load balancer.

the time of capture (*e.g.*, at night vs during the day) [21], [22]. These differences generate variability in the workload that the video analytics pipeline needs to process and have been exploited to design more efficient processing pipelines [7], [6]. Distream [6] proposed to exploit this variability to dynamically balance the workloads across processing clusters, taking advantage of periods of lower usage from some nodes in the architecture to support overloaded ones. Their solution achieves this through the use of two main design elements: (1) A cross-device workload balancer that takes the cross-camera workload correlations and the heterogeneous compute capabilities of smart cameras and edge clusters to optimize cross-camera workload balancing via an optimization problem; and (2) a workload adaption controller which triggers the cross-camera workload balancer when cross-camera workload imbalance is detected.

Unfortunately, this approach assumes that workloads have predictable profiles, either due to the cameras' relative locations, their time of capture [7], or, more generally, by training a prediction model based on previous patterns [6]. However, the introduction of mobile cameras generates a new level of variability in the workload that is not easily predictable. First, mobile cameras constantly vary their point of observation and might capture different scenes at different instances in time. Second, the inherent moving nature of mobile cameras causes them to appear or disappear from the deployment, generating sudden changes in the workload that the video analytics pipeline needs to process. Overall, relying on long term prediction models to infer incoming load is not sufficient, or even potentially counter-productive, to correctly balance the load in the presence of mobile cameras.

Challenge #3: Varying configurations. Early work in video analytics focused on the problem of optimizing the placement in the infrastructure of the functions that belong to the processing pipelines. The placement decisions behind this optimization are conventionally driven by the available resources in the compute infrastructure, *i.e.*, the number of available servers or GPUs, and the workload generated by available cameras, *i.e.*, the number of camera flows to process [5], [8], [16]. Due to the overhead incurred, changes in the deployment configuration, such as the addition or removal of processing components, occur a longer time scales due long term pattern shifts (*e.g.*, day vs night scenes) [7], [6]. However, the presence of mobile cameras introduces a new level of dynamism in the deployment that has yet to be accounted for. Mobile cameras can appear or disappear from the deployment, and the processing pipeline needs to be able to adapt to these changes without requiring a complete reboot of the processing pipelines. This introduces the need for an online approach to load balancing that can adapt to changes in the deployment configuration, such as the addition or removal of processing components, without requiring any hard reboots and quickly adapting, in less than a few seconds, to the incurred changes.

To tackle the identified challenges, we present in the following section VideoJam, a live video analytics solution aimed at supporting the coexistence of fixed and mobile cameras within the same processing pipeline.

III. VideoJam Architecture

This section presents the general design of VideoJam, a self-balancing architecture for live video analytics. VideoJam is designed around three design guidelines:

1) **Per-function type-based load balancing.** To cope with the heterogeneous performance profiles and highly variable workloads, we design VideoJam to implement a decentralized set of load balancers on a per-function type basis.
2) **Short-term load forecasting.** To enhance frames offloading across functions but avoid errors from unreliable long-term trends, we design VideoJam to solely rely on short-term forecasting of incoming workload trends.
3) **Robustness to configuration changes.** Finally, to work independently from variations in configurations changes and camera arrivals and departures, we design VideoJam to solely rely on observed performance patterns (*e.g.*, incoming rate) rather than any pre-compute knowledge of deployment configuration.

In the rest of the section, we first present the global overview of the VideoJam architecture, then describe in details the components of the system.

A. System Overview

The core VideoJam's architecture leans on a set of load balancers, one associated with each and every replica the

processing functions deployed among the set of available edge servers, as shown in Figure 4. In VideoJam, each load balancer (shown in Figure 5) communicates with the rest of load balancers installed on the functions of the same type (*e.g.*, background subtractor, object recognition, *etc.*), also called *neighbors*. The load balancer wraps a function replica and mainly consists of two components: the *Monitor* and the *Dispatcher*. Through these two modules the load balancer takes decisions about incoming frames from the previous functions in the pipeline and from the neighbors to compute the (1) the output to the next function in the pipeline, and (2) a load balancing policy, *i.e.*, the amount of frames to be offloaded towards its neighbor. Such policy is calculated on the basis of a state table containing the information on the states of each neighbor, presented in rows. This information includes the estimates of load for the neighbors derived from predictions generated by a forecasting model.

B. Message Exchange

In VideoJam's system of load balancers, each component computes its state and load balancing policy based on local information, *i.e.*, the incoming load from the previous function in the pipeline, and information collected from the neighbors acquired through a set of message interactions listed below. These mechanisms are designed to ensure that the load balancers converge to a common policy and compensate for potential errors in load forecasting.

Information Request. Load balancers submit a request for neighbor state information when their local data is missing or obsolete. In this case, a load balancer sends, together with its table, an information request to its neighbor, asking for information about its status. Based on the response (described below), the receiver then updates its table according to the table received, substituting the rows containing obsolete information.

State Update. A state update is always sent by a load balancer in one of two scenarios: (1) at bootstrap to announce its presence to neighbors; (2) after receiving an information request from one of its neighbors. State responses contain a summarization of the local system state, including current and forecasted loads, as well as expected incoming loads from neighbors. State updates are also sent when a prediction error is detected after receiving an offloaded workload from one of the neighbors. Such an error quantifies the difference between the expected load (according to the predictions) and the actual load received from the neighbor.

Congestion Risk Signal. A congestion risk signal is sent by a load balancer to one of its neighbors only when it detects, after updating its table and according to the last computed policy, that a given neighbor is transmitting offloaded traffic to itself, while this was not expected given previous exchanges, *i.e.*, when the neighbor is not present in the list of neighbors from which it should receive offloaded video data.

Data Offloading. Beyond the actual offloaded workload, the data transferred from a load balancer to one of its neighbors may also contain the sender's table. It is worth noting that a load balancer cannot receive workloads from the neighborhood while it is offloading to another neighbor. Hence, the receiver first checks whether it is offloading to one or more neighbors according to its computed policy. In this case, it recomputes the load balancing policy and checks whether, according to the new policy, it is supposed to receive some data. If not, it sends a congestion risk signal back to the sender. Otherwise, it receives the workload, computes the prediction error and sends a status update if the error is above a certain threshold.

C. System state computation

In VideoJam, the system state is computed periodically and represents the current state of the load balancer based on the incoming load from the previous function in the pipeline, the processing rate, the queue size, and the offload rate to its neighbors. Here we formally describe the computation of the system state.

Operational Time Windows. Each load balancer associated to function c, operates over a time windows W_j^k, which consists of $k \in \mathbb{N}^+$ consecutive time-slots and is defined as

$$W_j = W_j^k = \{w_i\}_{i=1}^k,$$

where w_i is the i^{th} time-slot of the time window. All the time-slots within the time window have the same duration of Δ seconds. The next time window is denoted as W_{j+1}.

Each load balancer $S_{c,n}$ is characterized by its state, and keeps track of the states of all its neighbors (identified by $\mathcal{S}_c^*$) in a table. We further indicate with S_c the set of all load balancers associated to function c deployed among the edge nodes.

State. At each time window $W_j^k = \{w_i\}_{i=1}^k$, the state of load balancer $S_{c,n}$ is defined by (1) its processing rate $\mu_{c,n}$, (2) the historical incoming load $\{\lambda_{w_i}^{c,n}\}_{w_i \in W_j^h}$ of $h \geq k$ previous time-slots, (3) its current queue size $\varphi_{W_j}^{c,n}$, and (4) the offload δ_n^c, *i.e.*, the total workload offloaded towards its neighbors. The table of each load balancer contains an estimation of all its neighbors' states. Such an estimation is based on the last information the load balancer has received from the neighborhood and it is updated every r time windows. When this time expired, *i.e.*, $r = 0$, the load balancer sends an information request to the neighborhood. As shown in Figure 5, the interaction of the two main components, *i.e.*, the Monitor and the Dispatcher, of each load balancer, determines the final offloading policy. We describe in detail how the dispatcher calculates the offloading policy in the next section.

The Monitor is in charge of monitoring the state and performance of the load balancer over time. More precisely, it supervises the incoming load from the prior function in the pipeline, and the processing rate. At each time-slot w_i of a given time window, the monitor measures the amount of workload $\lambda_{w_i}^{c,n}$ coming from the previous function in the pipeline. At the end of each time window, the monitor updates the historical incoming load of the state of the load balancer by shifting its $h-k$ values to the left and replacing the last k

values with the load received in the window W_j. After such an update, the monitor triggers the Dispatcher for the computation of the load balancing policy.

D. Load Balancer Algorithm

Here we describe in details the algorithm executed by each load balancer to compute its local offloading policy. The Dispatcher is in charge of computing the load balancing policy of `VideoJam` through the computation of the queue size and the prediction of future incoming workload for the load balancer. For a given function c, the policy is determined in such a way that the load is fairly distributed among all the load balancers in $\mathcal{S}_c$. In this manner, the load is processed at approximately the same time, as shown in [23]. Regarding the prediction of future incoming workload, given the limited capabilities of computational resources at the edge, state-of-the-art methods, such as Long Short-Term Memory (LSTM) [24], result to be computationally intensive and, hence, prohibitive for the described scenario [25]. Furthermore, when it comes to video analytics, learning the specific distribution of the load results to be a non-trivial task due to concept-drift problems [26], especially when dealing with mobile cameras. For these reasons, the dispatcher relies on lightweight ML models to predict the incoming workload. In particular, we have developed a lightweight convolution-based neural network model based on a few convolution layers for predictions [27]. This model presents fast inference time and high accuracy metrics. The complete pseudocode of the dispatcher is described in Algorithm 1.

Determine the Queue Size. Within a given time window W_j, each load balancer determines the queue size $\varphi^{c,n}_{W_{j+1}}$, which is the load at the beginning of the next window W_{j+1}, by counting the amount of load currently waiting for processing (line 2 of Algorithm 1). The queue size of each neighbor $S_{c,m} \in \mathcal{S}^*_c$ is estimated by adding (1) the difference between the incoming load and the processing rate, and (2) the total amount of workload exchanged with the neighborhood, to the previous expected load (lines 3–7 of Algorithm 1). More formally,

$$\tilde{\varphi}^{c,m}_{W_{j+1}} = \tilde{\varphi}^{c,m}_{W_j} + \sum_{w_i \in W_j} (\lambda^{c,m}_{w_i} - \mu_{c,m}) + \delta^c_m, \qquad (1)$$

where, $\delta^c_m = \sum_{p \in \mathcal{N}} \delta^c_{m|p}$, $\delta^c_{m|p}$ is the offload between $S_{c,m}$ and $S_{c,p}$, where $\delta^c_{m|p} < 0$ if data is offloaded from $S_{c,m}$ to $S_{c,p}$, and $\delta^c_{m|p} > 0$ if $S_{c,p}$ is sending data to $S_{c,m}$.

Predict Future Incoming Workload. The load balancer forecasts its load and the incoming load of its neighbors for the next time window. For such predictions we rely on a convolution-based neural network model used to predict short-term workload [28] (more details on the model are available in Section IV). The input of the predictive model is the historical incoming load W_h (*i.e.*, $\{\lambda^{c,n}_{w_i}\}$ consisting of $h \geq k$ previous time-slots) for all load balancers. While the output is the incoming load for the next window W_{j+1}. So when the predicted workload deviates from the actual workload during the monitoring phase, the Monitor can detect it and react to make adjustments at any time.

Offloading Policy Computation. Within a given time window W_j, each load balancer $S_{c,n}$ determines the offloading policy, *i.e.*, $\delta^c_{n|m}$ for each $S_{c,m} \in \mathcal{S}_c$ for the next time-window W_{j+1}. Such a policy is computed through the following steps:

(1) First, the estimation of the global load, $\tilde{\phi}^{c,n}$, is computed according to

$$\tilde{\phi}_{c,n} = \tilde{\varphi}^{c,n}_{W_{j+1}} + \sum_{w_i \in W_{j+1}} \lambda^{c,n}_{w_i}, \qquad (2)$$

i.e., the estimated queue size at the beginning of the next time window plus the expected incoming load. The queue size estimate is performed for all its neighbors in $\mathcal{S}^*_c$, whereas it can simply be obtained from the load balancer queue.

(2) Then, based on the estimation of the global load, each load balancer $S_{c,n}$ estimates the actual load that every load balancer in $\mathcal{S}_c$ should handle:

$$\bar{\phi}_{c,n} = \frac{\sum_{n \in \mathcal{N}} \tilde{\phi}_{c,n} * \mu_{c,n}}{\sum_{n \in \mathcal{N}} \mu_{c,n}}, \qquad (3)$$

where $\tilde{\phi}_{c,n}$ is the global load that should be processed by $S_{c,n}$ with a process rate of $\mu_{c,n}$, and $\sum_{n \in \mathcal{N}} \mu_{c,n}$ is the total processing capacity of all the load balancers associated to function c.

(3) The load balancer computes the *unbalanced* load for all the load balancers associated to function c (line 12 in Algorithm 1) as

$$\theta_{c,n} = \bar{\phi}_c - \tilde{\phi}_{c,n}. \qquad (4)$$

Positive values for $\theta_{c,n}$ indicate that the function associated to the load balancer will be underutilized. In this case, it should receive workload from neighbors $S_{c,m}$ with negative values of $\theta_{c,m}$. Negative values for $\theta_{c,n}$, on the other hand, point out that the function will be overloaded requiring to offload some workload to the neighbors.

(4) Finally, the amount of workload to be offloaded towards each neighbor $S_{c,m}$, *i.e.*, the offloading policy $\delta^c_{n|m}$, is finally computed based on $\theta_{c,n}$ as described from line 14 to 27 of Algorithm 1. Until all the θ's are set to 0, the dispatcher takes the most loaded $S_{c,n}$ to balance to the least leaded $S_{c,m}$ in S_c. The load is transferred from $S_{c,n}$ to $S_{c,m}$ if there is enough room for the load, and the unbalanced load of $S_{c,n}$ is set to 0. Otherwise, $S_{c,n}$ transfers to $S_{c,m}$ the maximum load that can be received $\theta_{c,m}$ and the unbalanced load of the receiver is set to 0.

IV. Implementation and Deployment Configuration

In this section we present the details on how we implement `VideoJam`, *i.e.*, the set of parameters of the architecture and the video analytics components. Further, we present our evaluation setup, including the evaluation metrics, the datasets we have select for evaluating the system, and the baselines we compare `VideoJam` with.

Algorithm 1: Dispatcher algorithm procedure

Function *dispatcher*(φ, λ, μ, δ, $\mathcal{S}_c$)
  /* 1. Determine the queue size */
  $\varphi^{c,n}_{W_{j+1}} = getQsize()$
  /* 1. Estimation load for neighbors, i.e., $\mathcal{S}^*_c$ */
  for $S_{c,m} \in \mathcal{S}^*_c$ **do**
    $\tilde{\varphi}^{c,m}_{W_{j+1}} = \tilde{\varphi}^{c,m}_{W_j} + \sum_{w_i \in W_j} (\lambda^{c,m}_{w_i} - \mu_{c,m}) + \delta^c_m$;
  end
  for $S_{c,n} \in \mathcal{S}_c$ **do**
    /* 2. Forecast the future incoming load, i.e., $\lambda^{c,n}_{w_i}, w_i \in W_{j+1}$ */
    $\lambda^{c,n}_{w_i} = model(\{\lambda^{c,n}_{w_i}\}_{w_i \in W^h})$;
    /* Estimation of global load */
    $\tilde{\phi}_{c,n} = \tilde{\varphi}^{c,n}_{W_{j+1}} + \sum_{w_i \in W_{j+1}} \lambda^{c,n}_{w_i}$;
  end
  /* Compute the unbalanced load */
  for $S_{c,n} \in \mathcal{S}_c$ **do**
    /* balanced load $\bar{\phi}_{c,n}$ */
    $\bar{\phi}_{c,n} = \frac{\sum_{n \in \mathcal{N}} \tilde{\phi}_{c,n} * \mu_{c,n}}{\sum_{n \in \mathcal{N}} \mu_{c,n}}$;
    /* unbalanced load */
    $\theta_{c,n} = \bar{\phi}_c - \tilde{\phi}_{c,n}$
  end
  /* 3. Compute the offloading policy */
  while *any*($\theta_{c,n} < 0$) *and any*($\theta_{c,n} > 0$) **do**
    /* the most overloaded */
    $n = argmin(\theta_c)$;
    /* the less overloaded */
    $m = argmax(\theta_c)$;
    $q = |\theta_{c,n}|$;
    if $q < \theta_{c,m}$ **then**
      /* load from $S_{c,n}$ to $S_{c,m}$ */
      $\delta^c_{n|m} = q$;
      $\theta_{c,n} = 0$;
      $\theta_{c,m} = \theta_{c,m} - q$;
    else
      $\delta^c_{n|m} = \theta_{c,m}$;
      $\theta_{c,n} = \theta_{c,n} + \theta_{c,m}$;
      $\theta_{c,m} = 0$;
    end
  end
end

A. Prototype Implementation

We implement VideoJam with about 400 lines of Python 3 code, using the asyncio [29] library to handle I/O operations of incoming frames and objects to process, and OpenCV v4.5.3 with Cuda v11.2.2 support for various vision models. The library is designed to easily support a variety of existing video analytics applications. We integrate the system in a docker image can be pulled from a public docker hub or built from the Dockerfile available in our public GitHub repository [1]. To evaluate the design effectiveness, we implement the functions of a typical traffic control application: vehicles' number plate detection.

Video Analytics Components. We implement the vehicles' number plate detection pipeline by integrating the following video analytics functions: video source and decoder, background subtractor and vehicle detection (for fixed camera sources), YOLO object detection (for mobile sources), and number/license plate recognition.

[1]The VideoJam implementation code is freely available at https://github.com/ENSL-NS/VideoJam.git

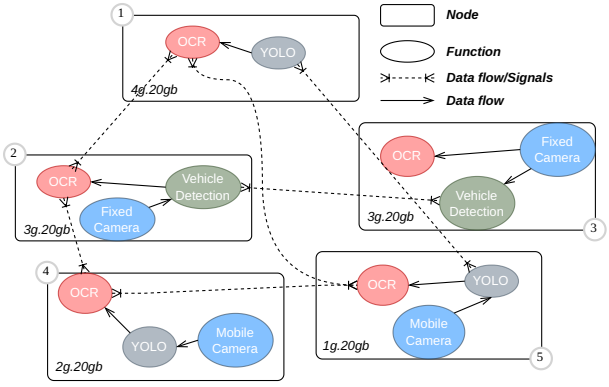


Fig. 6: An example of heterogeneous deployment of VideoJam. Note that not all links between functions are represented to reduce image complexity.

The decoder represents the entry point of the pipeline and takes as input an encoded stream (for the evaluation in this paper we use pre-recorded videos, yet the system supports live streams as well). The decoded video frames are then passed on to the next function for further processing.

For fixed cameras, frames are transmitted to the background subtractor. The background subtractor is a function that generates a foreground mask using a static camera. This mask is then used on the current image to subtract the static scene, *i.e.*, the background, while each moving scene is detected and classified as an object of interest. Extracted objects are then passed to the vehicle detection function, which embeds a machine learning model trained to detect vehicles within an image [30]. Detected vehicles are passed along to the next function, while other objects are discarded.

For detecting vehicles in mobile sources, we integrate a YOLOv5 model [20]. YOLOv5 is the fifth version of the YOLO (You Only Look Once) object detection model. It performs detection and classification, and returns a box for each object in the image taken in input, along with their classes with a high degree of accuracy. This is computationally intensive and is generally used on GPUs to achieve fast and accurate results in real-time. There are many pre-trained check-points available, as well as input size pixel images. For our purposes, we use YOLOv5s with a high (640) and low (416) input pixels size.

Finally, detected vehicles are passed to an object character recognition (OCR) function that used to detect license plates on cars. Numerous frameworks have been developed for this task. In our implementation we integrate Tesseract [31], an open source OCR engine that combines traditional image processing techniques with modern machine learning methods

Parameter	Value and description
Δ	1 second (the monitoring duration)
k	10, represents a monitoring window of $10 * \Delta = 10$ seconds
h	50, the history for short-term forecasting
model	short-term forecasting: DNN for Vehicle detection and Number plate OCR, none for YOLOv5

TABLE I: `VideoJam` configuration

Model	Dell PowerEdge R7525
CPU	AMD EPYC 7452 (Zen 2), 2 CPUs/node, 32 cores/CPU
Memory	128 GB
GPU	2 x Nvidia A100-PCIE-40GB, Compute capability: 8.0

TABLE II: Server configuration used for experiments.

to accurately recognize and convert text from images into a digital format.

We use these functions to create a heterogeneous application with two different pipelines. The first takes as input fixed video camera feeds passed along to the background subtractor, which forwards the data to the vehicle detector for classification before the final function, the license plate detection. The second pipeline takes data from a moving camera and processes them using YOLO, then passes the result to the license plate detection. This last function is shared by both pipelines for reuse and optimization. With such a deployment, a workload imbalance situation can arise at any time, and forecasting becomes more challenging. The complete application used for deployment is presented in fig. 6.

System configuration and model tuning. We configure the `VideoJam` load-balancing system using the parameters summarized in Table I. Regarding the predictive model, we opt for an architecture that minimizes inference time while guaranteeing acceptable performance. We choose a neural network (NN) with a single dense layer of 512 units trained over 100 epochs. The model takes in input a window of size h and can predict a window of size k (see Table I for the values of these parameters). We also tried out different architectures, such as a convolutional neural networks (CNN), and LSTMs. However, such models result to be difficult to use since (1) CNN requires a significant amount of time for the inference, although it has high levels of accuracy; (2) in our case, LSTM presents poor performance in terms of both accuracy and inference time given its specific use on time series (different from our case).

B. Evaluation Setup

Baselines. We evaluate `VideoJam` compared against three different baselines: (1) a video analytics pipeline that does not implement any load balancing, (2) one that implements Weighted Round Robin (WRR), and (3) Distream [6], a state-of-the-art solution. In WRR, neighbors determine their processing rates based on an initial estimation, share this information with their neighbors, and collectively assign weights based on their capacity to create a load-balancing policy. They then apply this policy to distribute incoming workload to their local queues or to neighbors, with offloaded work being placed directly in a neighbor's local queue. For Distream, we start with the version available at the project repository[2]. We then transform the Golang code into Python for integration into our deployment framework. Finally, we also adapt the code to support batch processing, mentioned in the article but not implemented in the open source version. The final code is also available on our public GitHub. Note that Distream does not support heterogeneous pipelines, thus we solely integrate the pipeline with YOLO into its architecture.

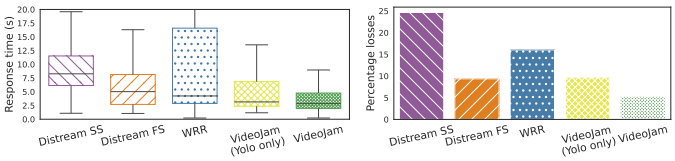
Deployment Infrastructure. We conduct experiments deploying multiple docker containers on a server grade machine equipped with Nvidia A100 GPUs (full specifications are shown in Table II). For functions necessitating access to GPU resources, we leverage the MIG (Multi-Instance GPU) that Nvidia GPU offers to split the available GPUs into multiple instances. We emulate an heterogenous edge infrastructure by splitting the GPU into six nodes with heterogeneous compute cores (*i.e.*, *4g.20gb*, 2× *3g.20gb*, *2g.10gb* and 2× *1g.5gb*) [32]. Given the lack of enough CPU cores we leave all containers to concurrently use all available CPUs. While this reduces the realism in terms of CPU isolation, the implemented functions mostly rely on the GPU for heavier computations, thus not introducing unwanted bottlenecks to the setup. Finally, the bandwidth between containers is limited to 1Gbit/s with the traffic control (*tc*) tool.

Evaluation metrics. We evaluate the performance of `VideoJam` and the other baselines using three metrics: (1) the response time, (2) the loss rate, and (3) the total bandwidth utilization. The response time for a frame refers to the time elapsed from its introduction into the system (*i.e.*, from the source) to its complete processing by the last function in the pipeline. It can also be measured at the level of a specific pipeline function. For example, the response time for a frame measured at the vehicle detection level corresponds to the time elapsed between its entry into the pipeline and its exit from this function. A low response time is an indicator of the system's ability to quickly extract the information generated by the application. The percentage of losses corresponds to the number of objects lost over the total number of objects to be processed. In general, each function has a queue with a maximum number of items that can be held in it. When the incoming load exceeds a function's capacity, it begins to accumulate load in its queue. When the queue is full, any new incoming objects arrive they are dropped. Since we do not focus on function selection, losses become the main indicator of accuracy reduction. Finally, total bandwidth utilization corresponds to the total amount of data transmitted during load balancing between functions. This is an important measure, as it enables us to measure the impact of the different

[2]https://github.com/AIoT-MLSys-Lab/Distream/tree/main

Type of camera	Duration (min)	Resolution	Total videos
Mobile	11-80	720p	9
Fixed	5-60	720p	11

TABLE III: Video cameras used for experiments.



(a) Response time. (b) Percentage losses.

Fig. 7: Evaluation on a heterogeneous architecture shows a response time up to 2.91× lower than comparative approaches, and with fewer losses.

approaches used on network resources.

Datasets. We use public videos from YouTube for both fixed and mobile cameras. These videos have different resolutions and durations, as shown in Table III. Part of this data is used for training the forecasting model we used for short-term forecasting. Note that, while fixed and mobile videos do not target the same scene, they are used to evaluate the system's ability to handle different types of video sources. As `VideoJam`, does not rely on any correlation between the videos, we do not expect this to affect obtained results.

V. Evaluation

We evaluate `VideoJam` in different scenarios and against the three baselines previously described. First, we compare it against Distream [6] to demonstrate the benefits of localized load balancing at function level, rather than a centralized approach for global load balancing. Next, we evaluate the system under different levels of loads to measure the performance of `VideoJam` as the workload increases. In addition, we test `VideoJam`'s ability to adapt to system configuration changes or failures, by subjecting it to critical situations such as the failure of a function or the failure of a node. Finally, we test the system's ability to deal with mobile cameras leaving and joining the architecture, demonstrating `VideoJam`'s ability to adapt to forecasting errors caused by sudden changes of video content.

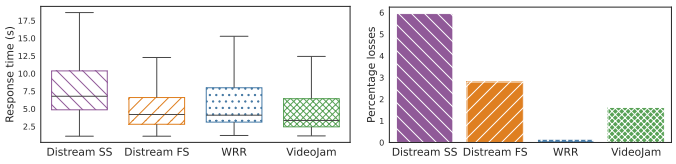
A. Comparison with Distream

Distream's load balancing architecture is based on two main key concepts: the cross-camera workload and the partition point. The cross-camera workload determines the workload balance among cameras (also called "Ends") only. The partition point defines which functions of each pipeline are processed by Ends, while other functions are then executed on the server (called "Edge"). The choice of offload proportion is handled in one of two different ways by the Ends: either a *full-stochastic* (FS) or *semi-stochastic* (SS) partitioning. In full-stochastic partitioning, the Ends generate a random partitioning proportion based on a Bernoulli random variable with probability set proportionally to the compute power of the Edge and End nodes. At every step of the pipeline, Ends draw a value from this variable and determine whether to process the function locally or offload it to the Edge. In semi-stochastic, the random value is drawn only at the partition point. In this case, the partition point is calculated as the point in the pipeline that evenly splits it proportionally to the compute power of the two components.

`VideoJam` outperforms baselines in heterogeneous deployments. We compare `VideoJam` and other baselines in dealing with mixed traffic of mobile and fixed video cameras. To carry out this experiment, we deploy Distream using five GPU-equipped nodes (as described in Section IV). We deploy one instance of YOLO and one instance of OCR on each node, as required by Distream. We use the same five nodes for `VideoJam`, but we partition nodes hosting the pipeline for fixed and mobile cameras. In particular, we deploy two vehicle detection instances, three YOLOs, and five OCRs. We also deploy two other baselines, WRR and a `VideoJam` version that solely employs YOLOs for detection, and using the same configuration as `VideoJam`. Finally, we use four video sources, two mobile cameras and two fixed ones, draw randomly from the dataset presented in Section IV and set to 20 frames per second.

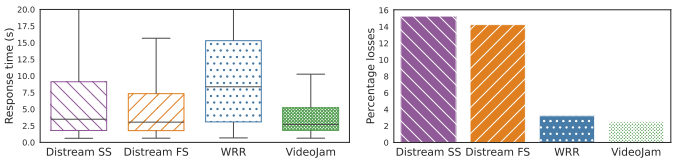
Figure 7 shows that `VideoJam` outperforms all other solutions, both in terms of response time, with 2.91× and 1.77× less response time compared to Distream's solutions, and reduces objects loss by 19% and 4%, respectively. This highlights the advantages of using a localized load balancing technique like `VideoJam`, and the limitation of approaches like Distream or a simplistic WRR. Further, the figure highlights that the use of mixed pipelines for different sources of traffic improves response time. Indeed, the use of dedicated pipelines for fixed and mobile cameras, *i.e.*, background substractor vs YOLO as explained Section II, improves classification performance incurring less load depending on the number of objects in each frame. Consequently, if the amount of objects in the video scene is low, no action, *i.e.*, inference, will be taken, whereas detection techniques such as YOLO will take action even if no objects are present in the frames. This is supported by the performance improvement observed when comparing the two approaches on `VideoJam` (*i.e.*, the one using vehicle detection over the one using only YOLOs). Furthermore, the results show that `VideoJam` has a response time 1.11× lower than `VideoJam` with YOLO only.

`VideoJam` outperforms Distream in mobile only scenarios. In the previous experiment we have shown that `VideoJam` outperforms baselines when processing mixed types of video traffic. We now explore whether our solution can still outperform Distream when solely processing video traffic generated by mobile cameras. We do so to evaluate `VideoJam`'s load balancing technique, understanding whether the advantages presented previously are to be solely attributed to the use of different pipelines for different types of traffic or to the load



(a) Response time. (b) Percentage losses.

Fig. 8: Evaluation on mobile cameras shows that VideoJam's response time is 1.25× shorter than Distream's.



(a) Response time. (b) Percentage losses.

Fig. 9: Evaluation of Distream and VideoJam in the event of node failure, with the latter recording fewer losses while maintaining better response time.

balancing as well. In this experiment, we use the same experimental setup as in the previous experiment, the only difference being that we use YOLOs throughout the deployment for all baselines, and we use four mobile cameras.

Figure 8 shows the performance of WRR, Distream, and VideoJam. We can observe that the semi-stochastic version of Distream incurs the worst performance in terms of both response time and loss: about 2.02× and 3.68×, respectively, compared to VideoJam. Indeed, given the design of Distream, it is difficult to define the load balancing policy when considering the pipeline as a whole. In fact, two different functions (*e.g.*, YOLO and OCR) on different nodes may become overloaded as traffic loads vary in time. This makes it very difficult to define an optimal load balancing policy for load distribution. Furthermore, in our observation, the considerable loss observed is due to the fact that a large portion of the video traffic remains continuously blocked in the Edge, which is unable to empty it quickly enough.

Nevertheless, we observe similar response time results between the full-stochastic approach and VideoJam. The reason for this lies behind the simple offload policy implemented in this approach: as the pipeline consists of only two functions, the partition is only necessary at either YOLO or OCR, often leaving the Edge in charge of the full processing. Yet, this simplicity can incur increased loss, when these nodes become overloaded (about 3% loss). We also observe that WRR experiences a lower percentage of loss compared to VideoJam. This is due to WRR's simpler load-balancing policy, which in some cases can be beneficial to loss, when frames spend more time being transmitted between nodes, slowing down the volume of traffic reaching the OCRs. In fact, we observe that WRR tends to balance load more aggressively across all available instances, thus increasing network usage (more details on this later in this section). Consequently, these frames are not queued fast enough to fill the ORC queues, explaining the small loss for WRR. This also explains the WRR's lower performance in terms of response time (1.22× lower response time than WRR). Ultimately, this shows that in certain instances, there might be a tradeoff to explore between response time and information loss. We leave this exploration for future work.

VideoJam better handles node failures. The aim of this experiment is to determine the impact of node failure on the performance of Distream, WRR, and VideoJam. To do this, we simulate two types of failure: the first is an End failure, the second an Edge failure.

Figure 9 summarizes the performance of each solution. We can observe a significant loss for Distream (about 15% of the total traffic). The reason for this large loss is that after the Edge has failed, the last policy calculated by the Edge, *i.e.*, the partitioning point, is still executed by the Ends and is not updated during the Edge's absence. So when the Edge comes back, the computation it is supposed to be dealing with since the last partition point update suddenly arrives, saturating the Edge in the process and causing several losses. Also, Distream's low response time is due to the fact that fewer frames are waiting in the queue to be processed after a large proportion of them have been lost.

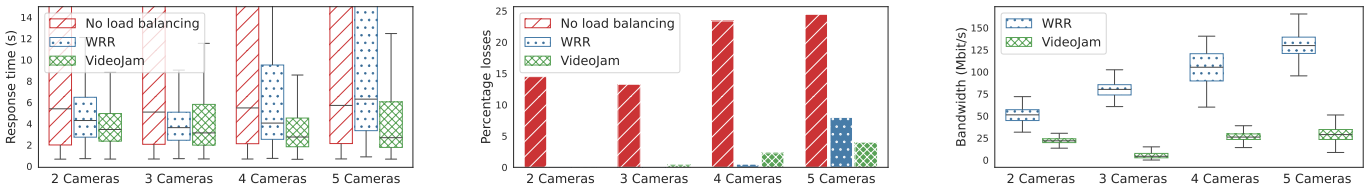
Furthermore, we see the ability of VideoJam to be robust to failures and to react a recovery. Indeed, when a node failure occurs, VideoJam adapts the load balancing policy calculation to the available resources. When the failed node returns, the policy is recomputed and all the workload already accumulated is redistributed. This explains the low losses of around 2% which are 6× lower than Distream's ones. WRR, on the other hand, does not present the same ability of robustness to failures, even though its policy is updated every time the system is stressed. And since its policy does not take into account the workload of the instances, the load previously accumulated during the downtime is not redistributed.

B. Load Balancing Ablation Study

In this section, we evaluate the effectiveness of VideoJam's load balancing algorithm with respect to two alternative baselines, no load balancing and WRR.

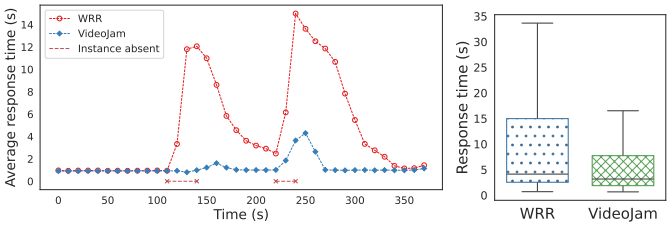
VideoJam Under Different Workloads. To evaluate the sensitivity of VideoJam to the level of workload, we subject it to different workloads, increasing the number of source cameras given a fixed function placement (deployment). In this scenario, we use six GPU-equipped nodes, and we set the number of YOLO and OCR instances to three and five respectively, while we increase the number of mobile cameras from two to five with a rate of 15 frames per second.

Figure 10 shows the system response time, the percentage of lost frames, and the bandwidth used during the offloading. VideoJam generally shows better performance with respect to the other compared solutions. The poor performance of no-balancing strategy highlights the need of load balancing solutions in this context. WRR reasonably shows longer response times as the load increases since such solution struggles in presence of network congestion. This is also highlighted by



(a) Response time of the system. (b) Percentage of frame lost. (c) Bandwidth used for offloading to neighbors.

Fig. 10: Evaluation on several configurations shows the need for a load-balancing technique and the effectiveness of VideoJam in achieving lower response times with less bandwidth usage.



(a) Averages for frames exiting YOLO. (b) Response time.

Fig. 11: Evaluation under node failure conditions: WRR's poor performance and VideoJam's adaptability under failures.

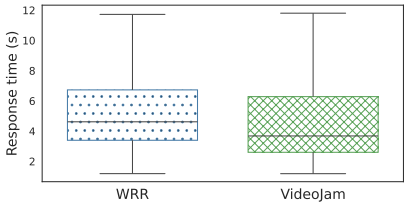


Fig. 12: Little effect of source mobility and abrupt flow changes on VideoJam performance.

the increase in the bandwidth utilization showed in Figure 10c. VideoJam, on the other hand, even in the event of congestion, only offloads the difference between instances to prevent two instances from sending load to each other, which explains the low bandwidth used. Furthermore, given the efficient collaboration among neighbors, VideoJam maintains stable performance as the workload increases.

In summary, this experiment has shown that VideoJam balances the workload without compromising performance, while minimizing system response time. In addition, while WRR statically balances incoming workload to neighbors, VideoJam adapts to the current situation by computing the most appropriate policy, and prevents bandwidth wastage due to bidirectional load migration.

VideoJam's Adaptation to Functions Placement. We evaluate the impact that different placement strategies can of on VideoJam, particularly in the case of real-time changes. Many studies have been carried out to find a better solution for placing components or functions to maximize the use of hardware resources while maintaining good precision [33], [12]. As VideoJam is agnostic of the placement strategy, we aim to evaluate if performance remains stable across different configurations. To do this, we start with four sources (15fps), three low-resolution YOLOs and five OCRs. Firstly, we deliberately remove a YOLO function before it is deployed at high resolution, which is more accurate than the first solution, but slower (lower throughput). This corresponds to a configuration change that can occur when placement techniques are used. In a second step, we kill all components (YOLO and OCR) on another node before restarting them after 15 seconds. This simulates a node failure that can occur at any time when dealing with edges.

The system response time and the average response time of frames, as they leave the YOLO component, are shown in Figure 11b and Figure 11a respectively. Initially, YOLOs maintain a consistently low response time, as they efficiently handle the incoming workload that is below their capabilities. At ∼100s (Figure 11a), a YOLO instance fails, with the source previously connected to it now sending its stream to another instance. This results in a sudden increase in the load that is badly distributed between instances for WRR. In contrast, in with VideoJam, instances remain capable of handling all the load generated by the sources. When the function is reintroduced into the system with a different configuration (higher resolution), WRR shows a gradual and slow decrease in response time, as it is unable to redistribute the loads already allocated to instances.

A similar behavior is observed at ∼220s, where a YOLO and an OCR deployed in the same node are killed due to node failure. Here again, the response time increases for WRR and VideoJam. In this particular case, the remaining YOLOs (one high-resolution and one low-resolution) are unable to handle the incoming workload, resulting in an increase in response time. In contrast, VideoJam excludes the outgoing instance from the load-balancing policy until it returns, or new functions are added, allowing previously accumulated overload to be efficiently redistributed between instances.

Impact of Mobility. One of the key features of mobile cameras is their ability to be in the right place to retrieve useful information. Nevertheless, in some cases their stream is interrupted by a number of factors (*e.g.*, displacement), before eventually resuming after a period of time. In this experiment, we therefore used three mobile cameras (20fps) that send their streams to four respective YOLOs, and then to five OCR instances. We want to evaluate the impact of

mobility on VideoJam. At some point, one camera stops sending its stream to the next function for a short period of $\sim$15 seconds. Which represents a camera leaving the system. This has a direct impact on load, as the incoming workload correlation is broken. We observe the system response time in Figure 12. When a source stops sending its data stream to the next function, the latter sees its incoming load drop drastically and then fails the VideoJam's prediction. Even though WRR is agnostic to these events as it continues to distributed frames independently of sources, VideoJam's load balancing compensates for eventual forecasting errors, maintaining 1.25$\times$ better response time.

VI. Related work

In recent years, several techniques have been developed to improve the performance of video analytics applications [34], [35], [10]. Such a topic has been tackled from different perspectives, including the design of different data processing architectures [11], [8], [7], the improvement of pipelines' processing [16], [13], [36], [37] and the privacy of the extracted data [38], [39], [40].

Architecture scaling. Different approaches have been proposed to efficiently manage the computational resources for video analytics [11], [8], [7], [33]. Chameleon, presented in [7], frequently reconfigures the placement of video analytics pipelines to reduce resource consumption with small loss in accuracy. Another example is Spatula [11], which exploits the spatial and temporal correlations among different camera flows to reduce the network and computation costs. However, such solutions only consider video flows coming from fixed cameras.

Deployment strategies. Other solutions mainly focused on the deployment strategies of video analytics applications [6], [33], [41]. Distream [6] is a distributed framework based capable of adapting to workload dynamics to achieve low-latency, high-throughput and scalable live video analytics. Pipelines are deployed on both the smart cameras and the edge, and are jointly partitioned so that part is computed on the smart cameras, while the rest is sent towards the edge, which has more computing power at its disposal. The deployment of application pipelines is adapted to the varying processing load, however there is a lack of adaptability required by the rate of mobile cameras. The work in [41] presents experimental results showing that smartly distributing and processing vision modules in parallel across available edge compute nodes, can ultimately lead to better resource utilization and improved performance. The same approach is also used by VideoStorm [33] which places different video functions across multiple available workers to satisfy users' requests. We assume a deployment of pipelines in line with this latter work given the higher flexibility, higher scalability and the better use of resources of this approach.

Load balancing strategies. Once the video analytics applications have been deployed on a distributed edge infrastructure, load balancing strategies play a fundamental role in guaranteeing requirements of accuracy and efficiency. In this perspective, several methods has been proposed in video analytics [6], [23], [28]. In [23], authors proposed two dynamic, adaptive and decentralized load balancing algorithms for grid computing environments to minimize response time. The main strength of such an approach is the estimation of system parameters (*e.g.*, arrival rate), which is achieved using exponential smoothing. However, the workload arrival rate assumption and the use of exponential smoothing for prediction may not be suitable for all systems, especially from heterogeneous video sources.

Workload prediction. Workload predictions, based on machine learning models, have been proved to be effective in the design of load balancing policies [42], [43], [28], [6]. In [44], authors used reinforcement learning for performing real-time estimation for dynamic assigning task to the optimal server. While the work in [28], focused on load forecasting by using linear regression model. However, reinforcement learning solution, even though effective, require a significant amount of resources and continuous online training to avoid concept-drift problems [45]. Such a solution method is not suitable for the computational-constrained devices at the edge. In addition, the rapid changes in scenes captured by mobile cameras are more difficult to predict. Therefore, there is a need of lightweight forecasting models that can predict short-term trends, suitable for edge devices and fast enough for real-time prediction.

VII. Conclusion

We present VideoJam, a new approach designed to meet the challenges of video analytics applications that integrate heterogeneous camera sources, *i.e.*, both fixed and mobile. VideoJam responds to scenarios incurring high load variability (such as mobile cameras) by integrating short term load prediction and performing load balancing at function level. Further, the system adapts to varying deployment configurations, not requiring any hard reboots to compensate for them. Thanks to its design, VideoJam reduces response times by 2.91$\times$ lower response time, while reducing video data loss by more than 4.64$\times$ and generating lower bandwidth overheads.

In future work, we plan to tackle the need of accounting of additional constraints in the analytics pipeline. Currently, VideoJam does not consider inter-function link bandwidths to determine load balancing policies. In heterogeneous network environments, where link speeds differ, this omission can have an impact on overall system efficiency. Further, while VideoJam works independently of the existing deployment configuration (*e.g.*, number of replicas for each function), it does not compensate for scenarios where the load exceeds the existing processing capabilities (*e.g.*, too many video sources to process). Future work will address these limitations, with the aim of improving resource utilization and exploring more adaptive deployment strategies.

ACKNOWLEDGMENTS

This work was supported by the ANR Project N^o ANR-21-CE25-0013 (PARFAIT) and the National Science Foundation under Award numbers 2055520 and 2106594.

REFERENCES

[1] R. Parascandola, "New NYPD surveillance cameras to cover stretch of upper east side not easily reached by patrol cars," Dec 2018. [Online]. Available: https://www.nydailynews.com/new-york/nyc-crime/ny-metro-argus-cameras-east-20181024-story.html

[2] J. Ratcliffe, "How many CCTV cameras are there in London? (update for 2020/21)," Nov 2020. [Online]. Available: https://www.cctv.co.uk/how-manfIPOy-cctv-camerfSECas-are-there-in-london/

[3] G. Grassi, K. Jamieson, P. Bahl, and G. Pau, "Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments," in *Proc. of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2017.

[4] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *IEEE Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[5] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[6] 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.

[7] 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.

[8] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *14th USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[9] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[10] M. Hu, Z. Luo, A. Pasdar, Y. C. Lee, Y. Zhou, and D. Wu, "Edge-based video analytics: A survey," *arXiv preprint arXiv:2303.14329*, 2023.

[11] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, P. Bahl, and J. Gonzalez, "Spatula: Efficient cross-camera video analytics on large camera networks," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 110–124.

[12] 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)*. IEEE, 2018, pp. 115–131.

[13] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015, pp. 155–168.

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

[15] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, "Bandwidth-efficient live video analytics for drones via edge computing," in *Proc of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[16] S. Fouladi and al., "Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads," in *Proc. of USENIX NSDI*, 2017, pp. 363–376.

[17] C. Pakha, A. Chowdhery, and J. Jiang, "Reinventing video streaming for distributed vision analytics," in *10th USENIX workshop on hot topics in cloud computing (HotCloud 18)*, 2018.

[18] H. Qiu, X. Liu, S. Rallapalli, A. J. Bency, K. Chan, R. Urgaonkar, B. Manjunath, and R. Govindan, "Kestrel: Video analytics for augmented multi-camera vehicle tracking," in *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2018, pp. 48–59.

[19] J. He, G. Baig, and L. Qiu, "Real-time deep video analytics on mobile devices," in *Proceedings of the Twenty-second International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, 2021, pp. 81–90.

[20] G. Jocher, "Yolov5 by ultralytics," if you use YOLOv5, please cite it as below. [Online]. Available: https://github.com/ultralytics/yolov5

[21] E. Yildiz, K. Akkaya, E. Sisikoglu, and M. Y. Sir, "Optimal camera placement for providing angular coverage in wireless video sensor networks," *IEEE transactions on computers*, vol. 63, no. 7, pp. 1812–1825, 2013.

[22] H. Guo, S. Yao, Z. Yang, Q. Zhou, and K. Nahrstedt, "Crossroi: cross-camera region of interest optimization for efficient real time video analytics at scale," in *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021, pp. 186–199.

[23] R. Shah, B. Veeravalli, and M. Misra, "On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments," *IEEE Transactions on parallel and distributed systems*, vol. 18, no. 12, pp. 1675–1686, 2007.

[24] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[25] V. S. Lalapura, J. Amudha, and H. S. Satheesh, "Recurrent neural networks for edge intelligence: a survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–38, 2021.

[26] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 119–135.

[27] P. Remy, "Temporal convolutional networks for keras," https://github.com/philippperemy/keras-tcn, 2020.

[28] R. K. Kombi, N. Lumineau, and P. Lamarre, "A preventive auto-parallelization approach for elastic stream processing," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1532–1542.

[29] "asyncio — asynchronous i/o," https://docs.python.org/3/library/asyncio.html, 2024.

[30] A. W. Ibrahim, "Vehicle detection, cnn." [Online]. Available: https://www.kaggle.com/code/abdallahwagih/vehicle-detection-cnn-acc-99-3/input

[31] "Tesseract." [Online]. Available: https://github.com/tesseract-ocr/tessdoc

[32] Nvidia multi-instance gpu user guide. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[33] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and Delay-Tolerance," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 377–392. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang

[34] N. Ibrahim, P. Maurya, O. Jafari, and P. Nagarkar, "A survey of performance optimization in neural network-based video analytics systems," *arXiv preprint arXiv:2105.14195*, 2021.

[35] R. Xu, S. Razavi, and R. Zheng, "Edge video analytics: A survey on applications, systems and enabling techniques," *IEEE Communications Surveys & Tutorials*, 2023.

[36] A. Padmanabhan, A. P. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali, "Towards memory-efficient inference in edge video analytics," in *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2021, pp. 31–37.

[37] 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.

[38] F. Cangialosi, N. Agarwal, V. Arun, J. Jiang, S. Narayana, A. Sarwate, and R. Netravali, "Privid: Practical, privacy-preserving queries on public video," in *19th USENIX Symposium on Networked Systems Design and Implementation*, 2022.

[39] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor:{Privacy-Preserving} video analytics as a cloud service," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1039–1056.

[40] H. Wu, X. Tian, M. Li, Y. Liu, G. Ananthanarayanan, F. Xu, and S. Zhong, "Pecam: privacy-enhanced video streaming and analytics via

securely-reversible transformation," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 229–241.

[41] 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. New York, NY, USA: Association for Computing Machinery, 2021, p. 13–18. [Online]. Available: https://doi.org/10.1145/3477083.3480153

[42] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 318–321. [Online]. Available: https://doi.org/10.1145/2611286.2611314

[43] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2013.

[44] H. Yuan, G. Tang, X. Li, D. Guo, L. Luo, and X. Luo, "Online dispatching and fair scheduling of edge computing tasks: A learning-based approach," *IEEE Internet of Things Journal*, vol. 8, no. 19, pp. 14 985–14 998, 2021.

[45] H. Zhang, W. Liu, and Q. Liu, "Reinforcement online active learning ensemble for drifting imbalanced data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 8, pp. 3971–3983, 2020.