To appear in the proceedings of the *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 14-17 May 2019, Larnaca Cyprus, as part of the *SCALE Challenge* track.

Dynamic Scaling of Video Analytics for Wide-area Tracking in Urban Spaces

Aakash Khochare, Sheshadri K. R., Shriram R. and Yogesh Simmhan

Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India
Email: {aakhochare, sheshadrik, shriramr, simmhan}@IISc.ac.in

***Abstract*—*Smart City* deployments typically have thousands to even hundreds of thousands of *Surveillance cameras*. Rapid advancements in computer vision techniques due to *Deep Neural Networks* enable using these camera feeds for performing *non-trivial* analytics. Tracking a moving *object of interest* using a large network of cameras, also known as *object reidentification*, is one such analytic that empowers city administration with capabilities such as finding missing people or prioritizing emergency vehicles. We have built *Anveshak*, a framework for distributed wide-area tracking. *Anveshak* fills in the shortcomings of existing Big Data and Deep Learning frameworks by – exposing an intuitive and composable programming model; automating application deployment and orchestration across *edge, fog* and *cloud* resources and providing *knobs* to the user for managing the application performance. The *knobs* lend the application the ability to scale potentially to thousands of cameras. In this proposal we have designed two representative applications; missing person tracking and priority signalling for emergency vehicles. We empirically verify that the application scales to 1000 cameras on a *Cloud-only* deployment of 10 Azure VMs with 8 cores and $32GB$ RAM each. Alternatively, it scales to 500 cameras on a simulated setup of 100 *edge*, 30 *fog*, and 1 Cloud VM. We also highlight the effect of the *knobs* on the application performance.**

I. Introduction

Safety of people and property is a key concern for city administrators. With 20% of global inhabitants living in cities with over 1 Million residents, making urban spaces safe and resilient is one of the goals of the *UN Sustainable Development Agenda for 2030* [1]. It is estimated that about 180,000 people are reported missing every year in the UK alone, and elderly with failing mental health and children are particularly vulnerable [2], [3]. Many cities are deploying *wide-area surveillance camera networks* [4], and these assist city agencies to locate, track and safely return such *missing people* to their guardians.

More broadly, camera networks help with crime prevention, tracking and resolution, through placement on roads, public transit, retail outlets, and public spaces like parks and libraries. *Advanced video analytics* supported by deep neural networks (DNNs) have started to automate some of these. Such video feeds can also be coupled with traffic signaling to give a "green wave" to *emergency vehicles* [5]. More recently, such cameras, are also serving as *meta-sensors* for Internet of Things (IoT) deployments in smart cities, to monitor and control garbage hotspots, air pollution, parking lots, etc. [6].

Deep learning has revolutionized *computer vision* in the last few years, with DNNs being able to detect and label images and video frames with high accuracy [7]. Such models are vital for enabling the applications listed above. However, there are several challenges that exist in coupling the data with these models. Building useful applications over urban camera feeds require multiple models to be *composed* together. Domain specific programming primitives and orchestration algorithms are required. Such applications are also *latency sensitive*, and require strong Quality of Service (QoS) guarantees to be useful. Moving feeds from 1000's of cameras to a *single cloud or a data center* is impractical due to the high bandwidth it entails, and the latency penalty. *Edge and fog computing resources*, that are co-located with the cameras or are part of smart city deployments need to be effectively used. The application also has to *scale along various dimensions*, such as the number of cameras, the frame rate, number of compute resources, etc.

In this SCALE proposal, we explore the *scalability characteristics* required by such video analytics applications across wide-area camera networks in urban spaces. We propose *tunable performance controls* for scalability, QoS and quality over our *Anveshak* distributed platform, which helps deploy and manage such video analytics applications on edge, fog and cloud resources [8]. We validate these for two example *urban tracking applications* over the camera network: (1) locating and following a missing person, and (2) tracking an emergency vehicle and giving it a "green wave" by controlling traffic signaling. These are validated using real-world datasets, models, and systems. While experiments for the former are presented in this paper, the results for the latter is left to the demonstration.

The applications leverage state-of-the-art DNNs, while judiciously using classic computer vision techniques wherever possible. While the first application relies purely on video feeds, the second application leverages audio signals along with video data for better accuracy at tracking. We empirically verify that the first application is capable of tracking a person reliably across 1000 cameras on 10; 8 core, 32GB cloud VMs. This should allow city scale deployments of tracking while keeping the compute requirement in-check.

The rest of the paper is organized as follows. We describe our *Anveshak* platform in Sec. II, introduce the two tracking applications in Sec. III, define the domain-sensitive performance metrics and our tunable controls in Sec. IV, offer experimental results and details of the demonstration in Sec. V, and contrast with related work in Sec. VI.

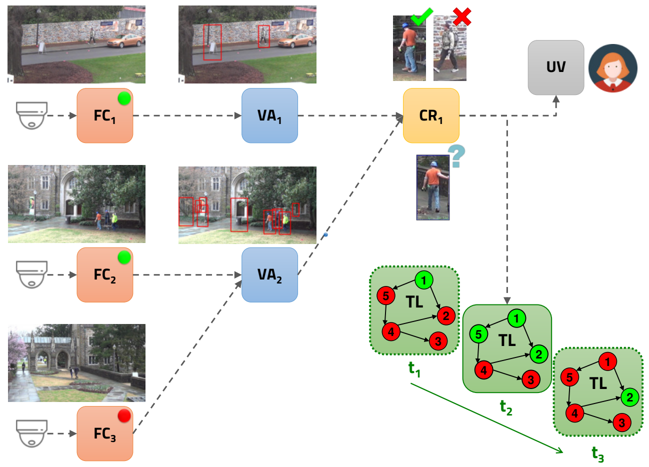


Fig. 1: *Anveshak*'s Modules and Domain-specific Dataflow

II. The *Anveshak* Platform

Anveshak [8] ("explorer" in *Sanskrit*) is a *distributed programming model* for composing video analytics applications that track objects across a camera network. It also includes a *light-weight platform implementation* that can execute the application on edge, fog and cloud resources.

The *Anveshak* domain specific model consists of five *functional modules* that users implement to define their tracking application: *Filter Control (FC), Video Analytics (VA), Contention Resolution (CR), User Visualization (UV)* and *Tracking Logic (TL)*. These are similar to `map` and `reduce` primitives of Hadoop, or the *transformations* of Spark. However, our modules are implicitly composed into a tracking dataflow, as shown in Fig. 1, with the edges representing streams of events, e.g., video frames, annotated bounding-boxes, detections. It operates in a data-parallel mode, with the module instances executing on different streams independently, and potentially on different compute resources.

Each *FC instance* is co-located with the camera on the edge, coupled with its video feed and ingests data into the application. It allows the feed to be activated or deactivated to avoid bandwidth use by infeasible cameras for tracking. The *VA module* performs analytics over the frames from a single camera, independently, with the framework performing the *group by camera ID* before invoking the user logic, which can include DNNs or vision libraries like TensorFlow, PyTorch and OpenCV to detect and classify objects.

The *CR module* operates across outputs from multiple cameras and provides a *high confidence match* with the object of interest, e.g., using a complex DNN. The *TL module* is a novel addition, and helps control the active set of cameras based on the movement of the object. It uses domain knowledge of the road network, points of interest, speed and trajectory of the object. etc. to decide the probable set of cameras that will locate the object being tracked. It controls the FC module to turn on or off camera feeds, and avoids unnecessary processing. The *UV module* reports the tracking progress to the user.

Anveshak is implemented in *Python*, allowing for native integration with popular *vision* and *Deep Learning* libraries such as *PyTorch, Tensorflow* and *OpenCV*, while staying light enough to be deployed on *edge* devices with no more than 1 GB of memory. Any device that is a part of *Anveshak*'s deployment must run a *Worker* Process. The device acting as the Client endpoint must run the *Master* Process. *Queries* are submitted to the *Master*. The *Master* schedules the userlogic onto devices and ensures that the *Worker* Process dynamically loads and executes the *userlogic* on the appropriate device. The platform also supports the various batching and dropping features, discussed later, as part of the tunable performance.

Our domain specific programming model offers developers a balance between the *flexibility* of incorporating diverse and emerging video analytics algorithms and the *pre-defined structure* of various phases of the tracking composition. The TL module is unique in allowing the users actively steering the tracking region, and also optimizing the application performance. The *Anveshak* platform allows users to deploy and coordinate the application on distributed edge, fog and cloud resources. This helps move the compute closer to the video feed source, reduces bandwidth and latency costs, and also leverages compute resources on the field.

III. Tracking Applications

We describe two realistic applications that highlight the expressibility of our programming model, and are used to validate various scalability metrics of the platform.

A. Locating and Tracking a Missing Person

This application identifies a missing person (*Person of Interest (poi)*) present within the camera network based on their photograph (*query image*) and last known location and time, and tracks them across space and time until they are rescued by safety personnel.

The application uses a simple FC with a boolean state that toggles the camera on or off based on the notification from TL. The VA uses a *pedestrian detector* based on a Histogram of Gradients (HoG) logic from OpenCV that puts bounding boxes around any pedestrians in the video frame. The CR module extracts these regions and compares them against the person's image using the *Open-ReID DNN* implemented in PyTorch [9]. This returns the probability of the match, and if it crosses a threshold, the information is passed to the UV module where a safety officer can respond.

The TL module uses the last seen time and location of the person, and their expected walking speed, to estimate the region where they should now be present. This uses a "spotlight" algorithm which starts from the last seen point, and expands circularly around this to activate neighboring cameras. The longer the person is missing, the wider the circle. This uses a weighted Breadth First Search (TL-WBFS), with knowledge of the road network, street lengths, the walking speed, and the camera locations. Once a person is found on a camera, the active set drops to that single camera, and expands again when the person falls in a blind-spot between cameras.

The faster the walking speed, the greater the expansion of the spotlight region and more cameras that are activated. Both positive and negative matches from CR are sent to the TL module to enable this logic. TL informs the FC's of the cameras within the spotlight region to activate, and deactivates others.

E.g., in Fig. 1, TL has access to the road and camera network, shown inset as a graph with 5 vertices. Initially, at time t_1, only camera 1 is active (green) since the person was last seen at this location. Then, at time t_2, when the person is not found, the search space expands to cameras $1, 2$ and 5 that are adjacent in the network, and these are activated. At time t_3, the person has been located in the field of view of camera 2, and the active set narrows to just this camera.

This application has a soft-deadline between the video being captured, and the detection being reported to the user at the UV module, to allow the safety personnel to respond. This can be on the order of minutes. The region of interest can grow large if the person is immobile within a blind-spot, or can walk fast. Lastly, it may be acceptable to occasionally miss locating the person in a camera feed since they are likely to be spotted soon after in the same or nearby feed. These factors influence the performance, scalability and QoS of the application. These are later described and validated in our experiments.

B. *"Green Wave" Signaling for Emergency Vehicles*

Tracking applications may use multi-modal data from diverse sensors deployed in the cities. Here, we combine analytics over *video feeds with audio feeds* from noise-level monitors present in various locations [10]. These are used to detect and track emergency response vehicles in the city, and pro-actively provide a "green wave" of traffic signals along their path to the nearest hospital [5].

The FC module is connected to sensors that provide video or audio feeds. Based on the sensor type, it routes the feed to one of two types of VA modules. For video feeds, the VA module uses YOLOv3 DNN implemented using TensorFlow [11] for detecting vehicles, while the CR module uses a vehicle re-identification DNN [12] to detect if it is an emergency vehicle. The audio feeds are sent to a VA that just accumulates a 5 *secs* audio clip that is passed to a CR module, which then uses a DNN [13] to identify if it is a siren.

The video and audio CR modules send their positive and negative detections to TL. This locates the emergency vehicle within the road network, identifies the closest among probable incident locations or hospitals, and sets the traffic lights along the shortest path to be green. It also activates the FC for relevant video and audio sensors along the possible routes. The UV module is informed about the latest position of the vehicle for visualization.

Here, the latency requirements for detection are more critical due to the high vehicle speed, and inherent delays required for safe signaling. It cannot afford to miss detections either. However, the TL intelligence can narrow down the routes based on the possible destinations, and limit the active set of sensors.

IV. Tunable Performance Controls

A. *Performance and Scalability Metrics*

The end user of a tracking application is primarily interested in three QoS metrics – the *end-to-end latency* for an input event, the *accuracy of tracking* without any loss, and the *maximum number of active cameras* supported by the platform.

The **latency** for processing an input frame from the camera (or an event from any other sensor) is the wall-clock time between its output from FC and its input to the UV module, after passing through the dataflow. As we saw from the two applications, this latency may need *soft or hard guarantees*, ranging from $\mathcal{O}(secs)$ for the traffic signaling to $\mathcal{O}(mins)$ for locating the missing person. Events processed beyond a certain delay may be stale, and not useful. We allow users to define a *maximum tolerable latency* that is the QoS that has to be met for events to be considered as successfully processed.

The accuracy of tracking depends on the specific logic used in the analytics modules. However, we characterize **accuracy** based on the number of incoming events successfully processed from start to finish. This arises due to our use of data drops as a means of throttling the resource usage and meeting the latency goals of the application, as discussed next. We define the *system accuracy* as $\frac{e_s - o_s}{e_s}$, where e_s is the total number of events generated and expected to arrive at UV, while o_s is the actual observed number of events processed by the dataflow. Alternatively, the *domain accuracy* is $\frac{e_d - o_d}{e_d}$, where e_d is the expected number of events that are a *positive match* for the input query, while o_d is the observed number of events with a positive match processed through till UV. Hence, dropping an event *will* cause the system accuracy to reduce, and *may* cause the domain accuracy to drop if it contained a positive match for the user's query.

Lastly, the maximum active cameras count is the largest subset of the deployed cameras whose input feeds the dataflow can process within the maximum tolerable latency. This is a direct measure of the **scalability** of the application on the *Anveshak* platform. When this count increases, the cumulative data rate generated from all the FC's and the compute load on the system increase. If we can meet the required latency with a linear increase in the quanta of compute resources available, then the application *weakly scales out*. Another means to report this scalability is the largest active camera count that can be supported for a *fixed set* of computing resources. Note that based on the TL logic, the active count of cameras will *dynamically change*. So we are interested in the *peak active camera count* that can be transiently supported for the given TL, as well as the *sustained camera count*.

B. *Performance Management using Tuning Triangle*

We incorporate a *novel and intuitive* mechanism for the users to trade-off the performance, accuracy and scalability metrics for their application running in *Anveshak* using a *Tuning Triangle* (Fig. 2). The vertices of the triangle capture the three metrics introduced above: the maximum tolerable latency, the system accuracy and the active set size. Correspondingly, on the edge opposite to this vertex, is a *tuning*

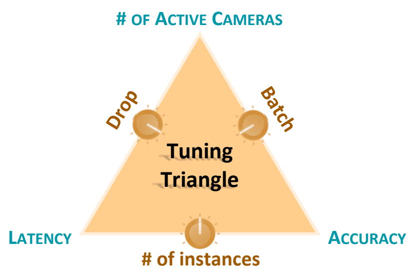


Fig. 2: *Tuning Triangle* for performance and scaling trade-off

knob available in the *Anveshak* platform that directly affects this metric. The design of the tuning triangle is such that the users fix two of the three metrics, and then modify the third based on its knob. These are discussed below.

Number of instances. The modules in our dataflow are data-parallel, typically at the granularity of a stream. Increasing the *number of instances per module*, with a proportional increase in the compute resources allocated to them, allows them exploit this parallelism. This impacts the *scale of active camera count* supported by the application, at a given latency and accuracy level. With a larger active set size, we can support a larger camera deployment, more aggressive TL logic, concurrent dataflow execution, etc. Also, choosing the right degree of parallelism per module in *Anveshak* ensures efficient utilization of the compute resources.

Event Batching. The module logic is often DNN model inferencing or other external libraries. Invoking such tasks from the *Anveshak* platform has certain inherent inter-process overheads. It has also been shown that DNN models offer a higher throughput when executed for multiple inputs as a batch, rather than for individual ones [14]. *Anveshak* allows users to *control the number of events that are micro-batched* before invoking the module logic [8]. Larger the batch size, greater the amortization of the overheads and better the throughput, i.e., more active cameras. But this also *increases the latency* for execution of the input events since there is a queuing delay when we accumulate events. *Anveshak* also includes heuristics for dynamic batching that tries to auto-tune the batch size to meet a specific latency limit, while keeping the active camera count and accuracy unchanged.

Event Drops. The maximum tolerable latency set by the user indicates that any event not processed before this deadline is not useful. Events that arrive late at UV are not just stale but have also wasted compute resources during their execution, preventing others from using them. Having a large active set size or a low latency threshold with inadequate resources can cause this frequently, and make the application unusable. Even a well-configured deployment may become unstable when, say, wide-area IoT network characteristics change.

Dropping events that cannot complete before their deadline as early as possible in the dataflow avoids this. *Anveshak* allows users to enable *event drop heuristics* which estimate the expected processing time for each event based on feedback signals, and drops events that cannot reach UV within the maximum tolerable latency [8]. However, such drops will affect the *system accuracy* for the application, while supporting a higher active set or lower latency. As an optimization, users may set a *do not drop* flag on events with a positive match. These will be prioritized and not dropped even if delayed, which can improve the *domain accuracy*.

V. Results and Demonstration

Here, we present experimental results to validate the impact of two of the three tuning knobs – *number of instances* and *event drops*. We illustrate these using the *missing person tracking* application. We defer details for the event batching knob and for the emergency vehicle signaling application to the demonstration. Further, we also report results for running these experiments exclusively on *cloud Virtual Machines (VMs)* and within a virtual IoT environment having *edge, fog and cloud (EFC) resources*, to illustrate *Anveshak*'s suitability for cloud data-center and wide-area IoT deployments.

A. Experiment Setup

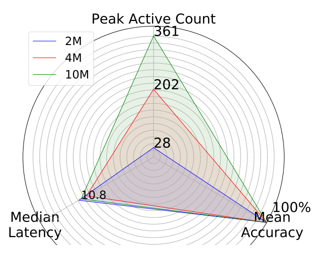
System Setup. We use Microsoft Azure `D8v3` VMs as our *compute nodes*. Each has an Intel Xeon 2673 v3 8 core CPU at 2.4 GHz, 32 GB of RAM and 1 Gbps inter-VM bandwidth. One Azure `D16v3` VM with 16 cores and 64 GB RAM serves as the *head node* to run the *Anveshak Master service* and a *Kafka v2.11.0 pub-sub broker* to generate the video feeds on its topics.

While the *cloud-only* setup directly uses 10 compute nodes for executing the module instances, the EFC setup instead uses *Docker v18.09 containers* with their CPU and memory constrained to match the performance of a *Raspberry Pi* edge with 1 GB RAM or a *Nvidia Jetson TX1* fog with 4 GB RAM. These containers are orchestrated using *Kubernetes v1.13* on the 10 Azure D8 VMs, with 10 edge and 3 fog devices per VM, and one D8 VM serves as the cloud resource. This gives a total of 131 compute resources for EFC on 11 host VMs.

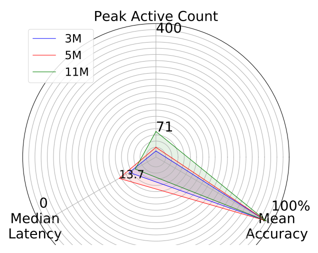
Application. The missing person tracking application uses a simple camera activation logic for FC, besides subscribing to the Kafka broker to acquire its video feed. It runs HoG for VA, Open-ReID DNN for CR, and the WBFS logic for TL. We run as many instances of FC as the number of cameras – up to 1000 in our experiments. For the cloud setup, each VM has up to 100 FC, 1 VA and 2 CR instances. In EFC, we place up to 10 FC instances per edge, along with 1 VA and 2 CR instances across 3 fogs. These result in 100:1:2 instances of FC:VA:CR per VM, directly or on edge and fog containers. One TL and UV instance are in one of these VMs for the cloud setup, and on the separate VM for EFC.

The *maximum tolerable latency* for the cloud setup is 15 *secs* and for the EFC setup is 25 *secs*. We use a *fixed batch size* of 10 events for all modules.

Workload. We use $54,000$ images from the DukeMTMC dataset [15] for simulating our video frames, and these have 20 unique images for each person we can track. The false positives and negatives are removed to allow for quantification of accuracy loss specifically due to data drops and not the model. These color images have a resolution of $1900 \times 1080 px$,

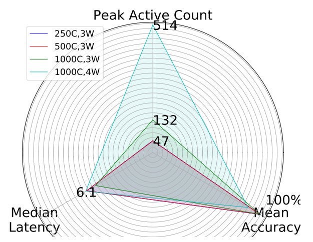


(a) Cloud

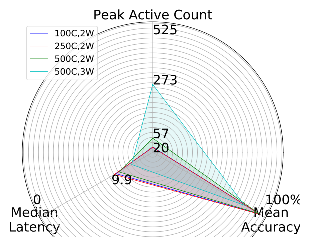


(b) EFC

Fig. 3: *Weak-scaling* of cameras with resources (w/o drops)

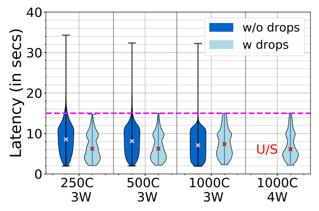


(a) Cloud

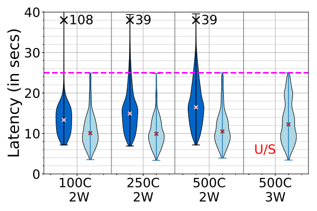


(b) EFC

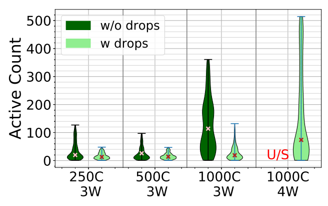
Fig. 4: Effect of *enabling drops* on accuracy and performance



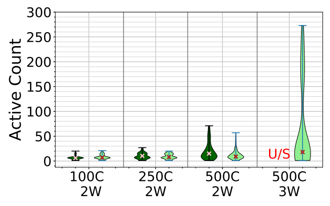
(a) Latency Distribution (Cloud)



(b) Latency Distribution (EFC)



(c) Active Set Distribution (Cloud)



(d) Active Set Distribution (EFC)

Fig. 5: Performance with and without data drops

with a median size of 574 kB. For the road network, we extracted a 7 km^2 region centered at the Indian Institute of Science campus from Open Street Maps [16]. This has $1,000$ vertices, $2,817$ edges, and an average road length of 84.5 m.

We simulate video feeds by overlaying virtual cameras on high-centrality vertices of the road network, surrounding the location where the search for the missing person starts. We simulate a random walk of the missing person through this road and camera network, with a given speed of walk. The cameras generate timestamped *negative* images from the dataset as video feeds at a fixed rate of 1 fps. But when the walking person "appears" at a camera vertex, the feed for this camera generates *positive* images of the missing person, for a duration of 3 $secs$. Each video feed is published to a distinct camera topic on Kafka, to which an FC instance subscribes.

B. Results

Tuning number of instances for weak-scaling. In this experiment, we tune the number of instances of each module and corresponding resources allocated, and examine if we can support a correspondingly higher number of deployed and active cameras, without any change in latency or accuracy performance (Fig. 2). For the cloud setup, we use 100 deployed cameras per VM, and run it for 200, 400 and 1000 cameras using 2, 4 and 10 VMs. For EFC, we use 50 cameras per VM, and run it for 100, 250 and 500 deployed cameras using 3 VMs (100 edge/6 fog/1 cloud), 6 VMs (250/15/1) and 11 VMs (500/30/1). These are the peak stable placement configurations. Drops are disabled.

Fig. 3 shows radial plots of the performance of the three metrics for cloud and EFC setups, for their *three VM instance counts* (e.g., 2M, 4M, 10M for cloud). The *peak active camera count*, *median latency* and *mean accuracy* are plotted. In all cases, since drops are disabled, the accuracy is at 100%. We also see that the median latency values across all input frames are close for all configurations, between 10.2–12.2 $secs$ for cloud, and 13.7–16.7 $secs$ for EFC. Since the ratio between the *number of deployed cameras* and the *VMs allocated* are nearly constant for all configurations, the application weakly-scales along this metric.

The *peak active camera count* actually shows better scaling as we increase the number of resources. In Fig. 3a, the cloud setup support 28 active cameras with 2 VMs, but 361 cameras with 10 VMs. We see a similar trend for EFC too. This shows that *Anveshak* scales well for larger camera and resource deployments, using cloud and EFC setups.

Tuning data drops and accuracy. Next, we examine the effect of enabling dynamic data drops to meet the *maximum tolerable latency*, and examine its impact on the system accuracy, and scaling of the active camera count (Fig. 2). Here, we used the peak number of resources for both cloud and EFC setups – 10 VMs for former and 131 devices in 11 VMs for latter. We the increase the number of deployed cameras, from 250–1000 for cloud, and 100–500 for EFC, and also use faster speeds of walks ($2W$–$4W$). Both of these cause the active camera set size and cumulative input frame rate to grow, and we measure the latency and accuracy.

The radial plots in Fig. 4 with drops enabled show that we are able to support as many as 514 active cameras for on the cloud, and 273 active cameras on EFC, with drops enabled. This is a 42–280% increase, relative to without drops. This has minimal impact on the latency, at about 6 $secs$ for the cloud, and 12 $secs$ for EFC. However, we note that the system accuracy marginally drops as the active set size increases, from about 99.8% with the fewest deployed cameras to about 86% with the most cameras. *This illustrates the accuracy–scalability trade-off enabled by this knob.*

The violin plots in Figs. 5 examines the latency and active set count distributions, both with and without drops. We note that without drops, the latencies occasionally go past the deadline (top row, magenta dashed line), but this never happens with drops as such events are discarded. We also see that the application is unstable (U/S) for the last configuration without drops (1000C/4W for cloud, 500C/3W for EFC), while it completes successfully with drops enabled. However, as we saw, the accuracy here drops to about 86%. We also see a long-tail for the active set count (bottom row) for these peak scenarios, with over half the deployed cameras active.

C. *Proposed Demonstration*

We will demonstrate the *green wave for emergency vehicles* application described earlier. The application is supported by full-fledged video analytics and DNN models, and dataset sources from the real-world and replayed realistically to mimic video feeds from large camera deployments. Our preliminary results described above show us scale to a 1000-camera deployment, and we expect to scale to 5000 camera feeds during the demonstration. We will use a container-based setup on a host commodity cluster to mimic the behavior of a city-scale edge, fog and cloud deployment, and illustrate the scale-out benefits by conserving compute and bandwidth.

We will also showcase the real-time monitoring of matching video feeds at the User Visualization module, and simulate the effect of live traffic signaling for the second application in a geo-spatial interface. All the performance and scalability metrics and knobs we have discussed above will be visualized through real-time monitoring of the application and the compute resources in our portal, including the dynamic changes to active camera size, accuracy, and latency, besides resource metrics like CPU, memory and network usage. We will provide data points to allow users to compare the scalability of *Anveshak* with a competent baseline.

VI. Related Work

Existing solutions for object *tracking* as *monolithic, proprietary, and bespoke*. In [8] we have highlighted the differences between *Anveshak* and existing surveillance and big data frameworks. The differences are briefly summarized as follows: (1) The existing frameworks either expect user expertise in computer vision or ship analytics as black boxes. (2) Popular computer vision libraries such as Tensorflow and PyTorch are not designed to be deployed across *edge*, *fog* and *cloud* resources. (3) *Knobs* that can affect application performance are not exposed to the user or their effect on the performance is difficult to predict. For example [17] does use knobs to achieve scalability in a *Cloud-only* environment. But the latency to quality tradeoff must be application developer provided. Instead *Anveshak* provides knobs that will predictably affect application performance.

The applications demonstrated here are standard object *re-identification* problems. Application 1 is a standard *person re-identification* problem that is well studied in the literature [18]. [19] is a survey on traffic management techniques that leverage sensors such as RFIDs, Cameras, and Bluetooth. People have explored techniques for priority traffic management for emergency vehicles [20]. We do not claim novelty for the applications themselves. The applications are designed to merely highlight *Anveshak*'s capabilities.

VII. Conclusion

Anveshak is the first open and modular framework for object tracking that is shown to scale to a 1000 cameras. *Anveshak* will help Smart Cities to draw meaningful insights from the large surveillance camera network, thereby allowing them to better assist citizens. The composability of the framework should also help researchers designing state-of-the art techniques by allowing them to rapidly deploy and test their techniques, since they can focus only on a specific module, such as TL or CR, without having to implement the entire pipeline from scratch. The *tuning triangle* allows users to make an educated guess while tuning the *knobs* to achieve scalability.

References

[1] L. Jensen, "The sustainable development goals report," United Nations, Tech. Rep., 2018.

[2] "Missing people Statistics," https://www.missingpeople.org.uk/about-us/about-the-issue/information-statistics/76-keyinformation2.html, accessed: 2019/02/07.

[3] "Stages of Alzheimers: Wandering," https://www.alz.org/help-support/caregiving/stages-behaviors/wandering, accessed: 2019/02/07.

[4] T. Porter, "A national surveillance camera strategy for england and wales," Surveillance Camera Commissioner, Tech. Rep., March 2017.

[5] Q. He, K. L. Head, and J. Ding, "Multi-modal traffic signal control with priority, signal actuation and coordination," *Transportation Research Part C: Emerging Technologies*, vol. 46, pp. 65–82, 2014.

[6] B. Amrutur and et.al., "An open smart city iot test bed: street light poles as smart city spines," in *IoTDI*. ACM, 2017, pp. 323–324.

[7] R. Geirhos and et.al., "Generalisation in humans and deep neural networks," in *NIPS*, 2018, pp. 7549–7561.

[8] A. Khochare, Y. Simmhan *et al.*, "A scalable framework for distributed object tracking across a many-camera network," *arXiv preprint arXiv:1902.05577*, 2019.

[9] Tong XIAO, "Open-ReID," https://cysu.github.io/open-reid/index.html, 2019.

[10] J. Gubbi and et.al., "A pilot study of urban noise monitoring architecture using wireless sensor networks," in *ICACCI*. IEEE, 2013, pp. 1047–1052.

[11] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint*, 2017.

[12] J. Sochor, J. Špaňhel, and A. Herout, "Boxcars: Improving fine-grained recognition of vehicles using 3-d bounding boxes in traffic surveillance," *IEEE Transactions on Intelligent Transportation Systems*, 2018.

[13] Y. Xu, Q. Kong, W. Wang, and M. D. Plumbley, "Large-scale weakly supervised audio classification using gated convolutional neural network," *ICASSP2018*, 2018.

[14] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[15] E. Ristani and et. al., "Performance measures and a data set for multi-target, multi-camera tracking," in *ECCV workshop on Benchmarking Multi-Target Tracking*, 2016.

[16] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org ," https://www.openstreetmap.org, 2017.

[17] H. Zhang and et.al., "Live video analytics at scale with approximation and delay-tolerance." in *NSDI*, vol. 9, 2017, p. 1.

[18] A. Hauptmann, Y. Yang, and L. Zheng, "Person re-identification: Past, present and future," 2016.

[19] K. Nellore and G. P. Hancke, "A survey on urban traffic management system using wireless sensor networks," *Sensors*, vol. 16, no. 2, p. 157, 2016.

[20] R. Sundar, S. Hebbar, and V. Golla, "Implementing intelligent traffic control system for congestion control, ambulance clearance, and stolen vehicle detection," *IEEE Sensors Journal*, vol. 15, no. 2, pp. 1109–1113, 2015.