Pre-print of article that appears in *IEEE Transactions on Parallel and Distributed Systems*, June 2021, pp. 1479-1493, vol. 32, DOI 10.1109/TPDS.2021.3049450

A Scalable Platform for Distributed Object Tracking across a Many-camera Network

Aakash Khochare, Aravindhan Krishnan, and Yogesh Simmhan, *Senior Member, IEEE*

Abstract—Advances in deep neural networks (DNN) and computer vision (CV) algorithms have made it feasible to extract meaningful insights from large-scale deployments of urban cameras. Tracking an object of interest across the camera network in near real-time is a canonical problem. However, current tracking platforms have two key limitations: 1) They are monolithic, proprietary and lack the ability to rapidly incorporate sophisticated tracking models; and 2) They are less responsive to dynamism across wide-area computing resources that include edge, fog and cloud abstractions. We address these gaps using *Anveshak*, a runtime platform for composing and coordinating distributed tracking applications. It provides a domain-specific dataflow programming model to intuitively compose a tracking application, supporting contemporary CV advances like query fusion and re-identification, and enabling dynamic scoping of the camera network's search space to avoid wasted computation. We also offer tunable batching and data-dropping strategies for dataflow blocks deployed on distributed resources to respond to network and compute variability. These balance the tracking accuracy, its real-time performance and the active camera-set size. We illustrate the concise expressiveness of the programming model for 4 tracking applications. Our detailed experiments for a network of 1000 camera-feeds on modest resources exhibit the tunable scalability, performance and quality trade-offs enabled by our dynamic tracking, batching and dropping strategies.

Index Terms—Big Data Platform, Edge and Fog computing, Video analytics, Distributed stream processing, Internet of Things

1 Introduction

The push for smarter and safer cities has led to the proliferation of video cameras in public spaces. Regions like London, New York, Singapore and China [1] have deployed camera networks with 1000's of feeds to help with *urban safety*, e.g., to detect abandoned objects, to track missing people and for behavioral analysis [2]. They are also used for *citizen services*, e.g., to identify open parking spots and count the traffic flow. Such "many-camera networks", when coupled with sophisticated Computer Vision (CV) algorithms and Deep Learning (DL) models can also serve as *meta-sensors* to replace other physical sensors for IoT applications and to complement on-board cameras for self-driving cars [3].

One canonical application domain that operates over such ubiquitous video feeds is called *tracking* [4]. Here, the goal is to *identify an "object"* or *"entity"* (e.g., a stolen vehicle or a missing child), based on a given sample image, in video streams arriving from cameras distributed across the city, and to *track that entity's movements* across the many-camera network in near real-time [5]. Fig. 1 illustrates a *missing person* being tracked across a network of 5 video cameras, C_A–C_E, on a road network using a smart *spotlight* tracking algorithm. A blue circle indicates the *Field of View (FOV)* of a camera. The path taken by the person between time t_1 and t_5 is indicated by the green dashed arrow. Given an image of the person, the goal is to trace their path across the city with high accuracy, while reducing the application design and computing overheads. These pose several challenges.

Challenge 1 (Composability). The application requires online video analytics across space and time, and this commonly has three stages: *object detection, object tracking,* and *re-identification* [4]. The first filters out objects that do not belong to the same class as the entity while the second tracks objects in a single camera's frame. Re-identification (or re-id) matches the objects in a camera with the given target entity [6]. Recently, a fourth stage, *fusion*, enhances the original entity query with features from the matched images that is then used for tracking, giving better accuracy [7].

Each of these individual problems is well-researched. But these stages have to be *composed* as part of an overall platform, and coupled with a *distributed tracking logic* that operates across the camera network and over time. Stages like *object tracking* may require specialized DNNs to deal with crowded scenes or occlusion. However, contemporary many-camera analysis platforms are *monolithic, proprietary and bespoke* [8], [9] [10]. They offer limited composability and reusability of models, and minimal support for custom tracking strategies. This increases the *time and effort* to incorporate domain intelligence and adopt the rapid advances being made in CV/DL.

Challenge 2 (Distributed Tracking). It is impractical to execute the full video analytics pipeline on all the cameras due to the punitive computing and network costs. E.g., just doing object detection on a 1000-camera network using a contemporary fast neural network requires 5–128 Titan XP GPUs; besides, the bandwidth to move the video streams to the compute resource is high [11]. Instead, these platforms should incorporate *smart tracking strategies* that limit the video processing to the cameras where the object is likely to be present and adapt to *blindspots* [12]. They can use

- *A. Khochare and Y. Simmhan are with the Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, 560012, India E-mail: aakhochare@IISc.ac.in, simmhan@IISc.ac.in*
- *A. Krishnan was an intern at Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, 560012, India E-mail: aravindhank11@gmail.com*

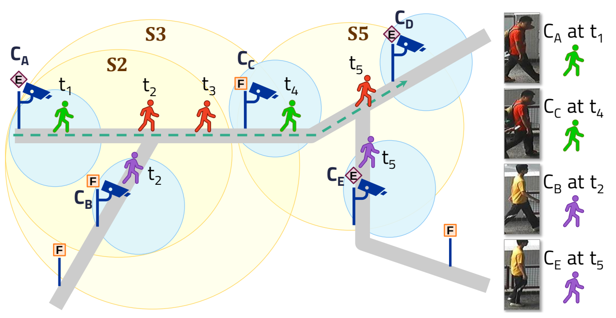


Fig. 1: *Spotlight* strategy for camera activation while tracking. Blue circles are the FOV of cameras C_A–C_E. The person icon shows people on the road at times t_1–t_5. A red person means the entity of interest is in a blindspot while a green person means they are in the FOV of a camera. A purple person indicates someone not being queried for. Sample images used in our experiments are shown to the right [13]. The yellow circles S_i are the calculated spotlight regions that indicate which cameras should be active at time t_i. The *purple diamonds* with an E indicate edge devices co-located with the cameras while the *orange squares* with an F indicate fog devices present across the city.

domain knowledge like the road and transit network, speed of the object, camera location and field of view to make smart choices on the video streams to be actively processed.

Example. In Fig. 1, the cameras need not generate and process video feeds unless *activated*. Initially, at time t_1, the target person (green icon) is within the FOV of C_A, and only this camera is made active. By time t_2, they (red icon) have moved out of the FOV of C_A, and also of all other cameras, i.e., in a *blindspot*. Now, we calculate a *spotlight* region around the camera where they were last seen, and *activate* cameras that fall in this region, as shown by the yellow circle S_2, which contains C_A and C_B. This spotlight grows to S_3 at time t_3 as the person is still in a blindspot, and it activates camera C_C as well. The person reappears in the FOV of C_C at time t_4 and the spotlight shrinks to S_4 with just this single camera being active and the rest are *deactivated*. The spotlight again grows at time t_5 when the person is lost, and S_5 activates cameras C_C, C_D and C_E. ■

Using such a smart tracking logic can reduce the number of active video streams we process, e.g., to 1–3 cameras rather than all 5, in Fig. 1. This reduces the resource usage substantially with limited impact on the tracking accuracy. However, contemporary many-camera analysis platforms do not offer such sophisticated tracking logic.

Challenge 3 (Scaling across Edge and Fog resources). Smart cities are seeing *edge and fog computing resources* being deployed on Metropolitan Area Networks (MAN), to complement *cloud resources* [14]. Such edge and fog computing resources can be used to achieve a judicious use of the Internet bandwidth. [15]. This also brings processing closer to the data source [16]. Fog resources distributed across the city, with higher compute capability and even low-end GPU accelerators, can complement edge resources in efficiently processing video streams [15]. This is important for video tracking, given its low latency, high bandwidth and high compute needs [1], [17]. So tracking platforms must effectively use such heterogeneous, wide-area compute resources that are part of the computing continuum rather than rely exclusively on cloud resources.

For *scalability*, the platform must balance the latency for tracking against the throughput supported for the active camera feeds on the available resources – a high latency can cause the object to be detected late, and lead to the spotlight region growing larger when the person is missing, while a low throughput can limit the number of cameras that can be active at a time, and increase the chances of losing the person. Also, given the *dynamism* of Wide Area Networks (WANs), compute performance and stream rates, the platform must trade-off the accuracy of tracking with the application's performance at runtime. *Current platforms do not offer such tunable adaptivity and scaling* [5], [18].

We make the following specific contributions in this article to address these challenges:

1) We propose a novel *domain-specific dataflow model* for current and emerging tracking applications, with functional operators to plug-in different analytics. Uniquely, it has first-class support for *distributed tracking strategies* to dynamically decide the active cameras (§ 2). These address Challenges 1 and 2.
2) We implement the dataflow model and heuristics in our *Anveshak platform* to execute across distributed edge, fog and cloud resources (§ 3). Further, it incorporates domain-sensitive heuristics for *dropping and batching frames*, which allow users to tune the accuracy, the latency and the scalability under dynamism (§ 4). These address Challenge 3.
3) We illustrate the *flexibility* of the dataflow model using 4 tracking applications, and offer detailed empirical results across latency, accuracy, camera-set sizes and tracking logic to validate the *scalability and tunability* of our platform (§ 5).

We complement these with a review of related work in § 6 and offer our conclusions in § 7.

2 A Domain-specific Dataflow for Tracking

2.1 System Model

A many-camera infrastructure consists of a set of cameras that are statically placed at specific locations in a city, and each can generate a stream of video observations within its FOV [5]. The cameras are connected to a MAN, directly or through an edge device [17]. Fog devices may also be co-located with the cameras or within a few network hops of them, while cloud resources are accessible at data centers over the WAN [14]. While the edge and fog are typically captive city resources, cloud resources are available on-demand for a price. These resources have diverse capacities, and their performance may *vary over time* due to multi-tenancy. The bandwidth and latency between devices on the MAN and the WAN can be *dynamic*, depending on the traffic. These can affect the QoS of distributed applications.

Cameras allow remote access to their video streams over the network and expose endpoints to control parameters such as the frame rate, resolution and FOV [19]. Rather than move these video feeds to a data center for processing, we instead propose to move the analytics to the data by using edge and fog devices close to the cameras, complemented by the cloud for control.

2.2 Domain-specific Programming Model

We propose a domain-specific model for tracking applications as a pre-defined *streaming dataflow* with *modules* that correspond to the logical stages of a tracking application (Fig. 2). We specify the input and output *interfaces* for each module, and statically compose them. Multiple *instances* of a module can data-parallely execute different input events. The user defines an application by providing the *compute logic* for each module stage, which consumes and produces streams of events (e.g., video frames, detections), and specifies the *routing* between module instances.

General purpose dataflow models like ORCC [20], Apache NiFi [21] and Apache Spark [22] allow programmers to connect modules in a flexible manner to help compose diverse applications. In contrast, we give a high-level dataflow composition to meet the specific needs of tracking applications, and focus on specific implementations of these modules based on advances in DL/CV models, and uniquely, control the distributed tracking logic through a custom module. This is like Hadoop MapReduce [23] where the user specifies the *Map* and *Reduce* logic, but the dataflow and execution pattern is pre-defined. Like Hadoop, our runtime platform also offers the benefits of automatic parallelization and performance management.

Next we describe the interfaces of these modules, the dataflow pattern, and the execution model (Fig. 2).

2.2.1 *Filter Controls (FC)*

This module is the entry point for video frames from a camera into the dataflow. It is usually co-located with the camera or on an *edge device* connected to it. Each camera has a single FC instance along with its *local state*. Users provide a logic to decide if a video frame on the input stream of an FC should be forwarded on its output stream to the Video Analytics (VA) module, or ignored. FC can use its local state (e.g., *isActive*) or even the frame content to decide this. If a frame is forwarded, a key-value event is sent on the output stream, with camera ID as key and frame content as value.

Importantly, the FC state for a camera (e.g., *isActive*) can be *updated* by control events from the Tracking Logic (TL), as described in § 2.2.4. This allows *tunable activation* of video streams that will enter the dataflow, on a per-camera basis. E.g., TL can have FC deactivate a camera feed if the target will not be present in its FOV, or reduce/increase the frame-rate based on the target's speed. The FC logic should be simple as it typically runs on edge devices.

2.2.2 *Video Analytics (VA)*

This module receives input event streams from one or more upstream FC modules, and performs *video analytics on a single camera's stream* at a time. Users can define complex compute logic for object detection and tracking, and even invoke external *TensorFlow, PyTorch* or *OpenCV* models [24]. The input API for the logic is an iterator of events, *grouped* by the camera ID, and it can also access the *target query* (e.g., an image of a person), and maintain *local state* across executions. This is similar to the *shuffle and reduce* in MapReduce.

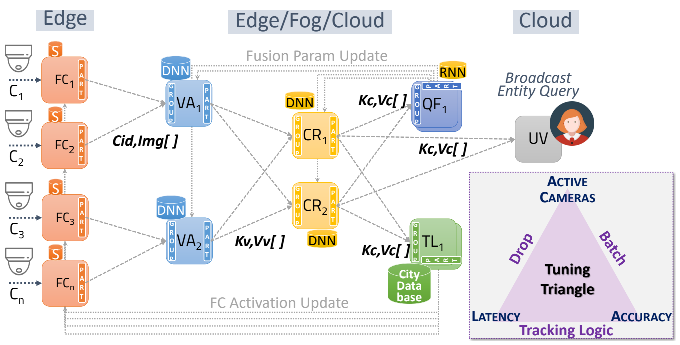


Fig. 2: Domain-specific dataflow and modules for tracking. (Inset) Tunable performance and scalability choices.

Grouping by camera ID gives the user logic access to a *batch of frames* from the same camera for temporal analytics. It also allows batching of inputs for model execution to amortize loading costs, using strategies proposed in § 4.4.

The output of the logic is a batch of key-value pairs, which may be, e.g., the camera ID (key), and bounding boxes for potential target objects in a frame with confidence scores (value). There can be a many-to-many relationship between the input and output events for this module. We allow users to *link* an output event with an input event to let us trace its latency and help with drop strategies we propose in § 4.3. Depending on the compute needs, it may run on edge, fog or cloud resources.

The local state of this module can be updated by the *Query Fusion (QF)* task. This allows dynamic updates to the entity query by *fusion* algorithms [7] to enhance a query's feature vector from successful detections of the entity. The VA can also update its model based on such signals.

2.2.3 *Contention Resolution (CR)*

This module receives a stream of key-value events from one or more VA instances, *grouped by key*. The keys are typically the camera ID and the values contain detections or annotated frames, but these can be overridden by the VA user logic. It has access to the entity query as well. The user can provide logic to analyze results from *multiple cameras*, say, to resolve conflicting detections from different cameras, or use more advanced DL models for a *higher accuracy match*. CR may be triggered only on a conflict or a low confidence detection by a VA, and hence execute less often than VA, but be compute intensive. CR may even degenerate to a human-in-the-loop. This makes it better suited for running on fog or cloud resources. The output stream from CR primarily contains metadata – much smaller than the video input – and this is forked three ways, to TL, QF and UV modules. Like VA, this module can receive updates from QF as well.

2.2.4 *Tracking Logic (TL)*

This is a *novel module* that we propose to help users capture the core logic of distributed tracking across the multi-camera network [25]. The detections that TL receives from CR for each frame may be a *positive* or *negative* match with the target query. On a negative detection, users can define a TL logic to *expand* the search space by activating additional cameras, while if the entity is found in a frame (positive), they can *contract* the search space. The module can use sophisticated tracking algorithms with prior knowledge of the environment and the entity, and devise strategies to (de)activate the cameras to optimize the quality and performance of tracking. It can be hosted on cloud resources.

E.g., in Fig. 1, TL uses knowledge of the road network and camera locations to dynamically decide the camera search space (spotlight), depending on when and in which camera the entity was last detected, and (de)activates those cameras. It can also be more sophisticated and have the cameras focus on an approaching or receding entity, or change the frame-rate based on the entity's speed. This separates the core video analytics logic, from distributed entity tracking across the camera network and camera controls.

2.2.5 Query Fusion (QF)

This module uses information on the detections to enhance the entity query's features. High-confidence entity detections in the input video can be fused with the existing entity query to generate a new query that offers better matches, or even use negative matches to enhance the query [7], [25]. The output of this module updates the entity query at the VA and CR modules for their future input streams.

2.2.6 User Visualization (UV)

This is a user-facing module that can be used to submit the entity query and display the current state of the tracking and detections. This can be a central portal running on the cloud where authorized personnel can view the progress.

2.3 Composing Tracking Applications

When composing a tracking application, users provide a YAML file pointing to a Python implementation of each of these modules, along with configuration details, which is then executed by our Anveshak platform. Each module has *init*, *compute* and *partitioner* functions with a fixed input and output signature. Algorithm 1 gives the user logic for the *compute* function of the FC, VA, CR, TL and QF modules for a sample *OpenReID (ORID) Application* [26] to track a person entity across a road network. The App takes the image of a person as the input query, and returns detections of the entity in the camera network to the UV module. Fig. 2 shows how these modules are embedded into the predefined dataflow pattern and the data flow between them.

FC uses the *active* state to decide if the camera's output should be passed to the downstream VA module as a series of key-value events, $\langle C_{id}, img\rangle$, having the camera ID and image. At the start, all FCs have *active=true* to let their camera's output be passed through to initially locate the entity. All images from one camera ID are routed to a single VA instance, determined by a *partitioner* function provided by the user, and multiple FCs can send their feeds to one VA, e.g., FC_1 and FC_2 to VA_1, in Fig. 2.

VA executes over a batch of images from one camera at a time, $\langle C_{id}, imgs[\,]\rangle$. For ORID, it uses a feature-based HoG pedestrian detector [27] (line 2) to put bounding boxes (*bbs*) around persons in each image. The user's Python compute logic invokes OpenCV's HoG external library, which executes on the entire batch of images. For each input image, it emits a key-value event, $\langle C_{id}, \langle img, outbbs[\,]\rangle\rangle$, which has

Algorithm 1 Modules' *compute* pseudocode for ORID App

```
1: procedure FC($img, state$)
2:     return $state.get('isActive')$
3: end procedure
```

```
1: procedure VA($C_{id}, imgs[\,], state$)
2:     $bbs[\,][\,]$ = OpenCV.HoG(imgs[ ])
3:     for $img$ in $imgs[\,]$ and $outbbs[\,]$ in $bbs[\,][\,]$ do
4:         EMIT($C_{id}, \langle img, outbbs[\,]\rangle$)
5:     end for
6: end procedure
```

```
1: procedure CR($\langle C_{id}, \langle img, outbbs[\,]\rangle\rangle[\,], state$)
2:     $query = state.get('entity\_query\_img')$
3:     $cropped = [\,]$
4:     for $tuple$ in $\langle img, outbbs[\,]\rangle[\,]$ do
5:         $cropped\_img$ = CROP($img, outbbs[\,]$)
6:         $cropped.append(cropped\_img)$
7:     end for
8:     $detections$ = PyTorch.DNN_CR($cropped, query$)
9:     for $was\_detected$ in $detections[\,]$ do
10:        EMIT($C_{id}, \langle img, was\_detected\rangle$)
11:    end for
12: end procedure
```

```
1: procedure TL_WBFS($\langle C_{id}, \langle img, detections[\,]\rangle\rangle[\,], state$ )
2:     $el$ = GETENTITYLOCATION($\langle C_{id}, detections[\,]\rangle[\,]$)
3:     if $el == \varnothing$ then            ▷ Entity lost. Expand spotlight...
4:         $graph = state.get('road\_network')$
5:         $lsl = state.get('lastSeenLocation')$
6:         $lst = state.get('lastSeenTime')$
7:         $cameras[\,]$ = WEIGHTEDBFS($graph, lsl, lst$)
8:         EXPANDSEARCHSPACE($cameras$)
9:     else
10:        SHRINKSEARCHSPACE($el$)
11:    end if
12: end procedure
```

```
1: procedure QF($\langle C_{id}, \langle img, detections[\,]\rangle[\,]\rangle, state$)
2:     $oldFeature \leftarrow state.get('state')$
3:     for $image$ in $img[\,]$ do
4:         if $detection == true$ then
5:             $newFeature \leftarrow RNN(image, oldFeature)$
6:         end if
7:     end for
8:     $emit(C_{all}, image[\,], out[\,])$
9: end procedure
```

the camera ID, the image and its bounding boxes. It is sent to one of the CR instances determined by the partitioner.

CR receives a batch of tagged images from each camera, crops and extracts the image regions in the bounding boxes, and passes this batch to a high-quality PyTorch DNN for pedestrian detection [26] (line 8). It matches the query entity against the images and emits them with a *true* or *false* flag, $\langle C_{id}, \langle img, was\_detected\rangle\rangle$, which is sent to UV, TL and QF.

UV (not shown) just displays the camera frames having a *true* flag to the user. TL, however, combines the presence or absence of the entity in a camera, with the road and camera network, and the last known location of the entity stored in the *state* variable, to decide the cameras to (de)activate. If the entity is missing from all cameras, we start a *Weighted Breadth First Search (WBFS)* on the road network from the last known position of the entity (line 7), considering the road lengths, the entity's peak speed and the time since its last detection. This identifies the spotlight region where the

TABLE 1: Module mappings for illustrative tracking apps

App	FC	VA	CR	TL	QF
ORID	Active?	HoG [27]	Open Re-id [26]	WBFS	–
PRID	Active?	HoG [27]	Person Re-id [28]	BFS	RNN [7]
VRID	Frame Rate	YOLO for Cars [11]	BoxCar Re-id [29]	WBFS w/ speed	–
PbRID	Active?	Person Re-id (Small) [31]	Person Re-id (Large) [32]	Probabi-listic	–

entity should be present, and TL signals the FC of cameras in this region to activate them (line 8). Else, if the entity is detected in some camera's frame, the spotlight contracts to that camera and deactivates all others (line 10). Lastly, QF uses an RNN [7] to enhance the entity query using high-quality hits and routes them to all VA and CR instances.

Table 1 lists the module logic used by ORID and three other exemplar tracking applications we can compose. We use the TensorFlow-based PersonReID DNN [28] in CR for the *PRID App*, with an *unweighted* BFS logic for TL. The query may also match a vehicle's image, in *VRID*, which uses DNNs for vehicle detection in both VA [11] and CR [29]. Here, TL is also more complex, with awareness of the road lengths and speed limits. In *PbRID*, we use a Naïve Bayes model to give the likelihood of paths that will be taken by the entity to decide the cameras to activate.Applications may also use DNNs trained for crowded traffic [30] as their CR module. More details on the dataflow composition and PRID App are in Appendix A.

3 Anveshak Platform Implementation

We implement this domain-specific dataflow model as *Anveshak* (*Explorer*, in Sanskrit), a Python-based distributed runtime engine that allows users to easily define their tracking application. Its architecture is illustrated in Fig. 3. Anveshak is more *light-weight* than Big Data streaming platforms like Apache Spark Streaming or Flink [22], [33], and designed to operate on a WAN than a Local Area Network (LAN). This allows it to be deployed on *edge, fog or cloud* resources.

Application developers implement their user logic in Python for the different modules of the dataflow, such as in Table 1. External models like OpenCV and TensorFlow are invoked by a user's Python *compute* logic for a module as a library call, by a command-line execution, or by invoking a local *gRPC* service that wraps the model. Using a gRPC service helps amortize the model loading and execution overheads across many events. A *Master* process runs in the cloud at a well-known endpoint and manages the application deployment. The application composed in YAML is submitted to the Master with the module definition, instance count and configurations, e.g., path to a DNN model for VA and CR, or the expected entity speed used by TL.

The Master calls a *Scheduler* logic that decides the mapping of module instances to the resources. The scheduling logic is modular. By default, we use a simple round-robin scheduler with a fixed number of instances per module type, and map specific module types to specific edge, fog or cloud resource abstractions. More advanced scheduling strategies are beyond the scope of this paper.

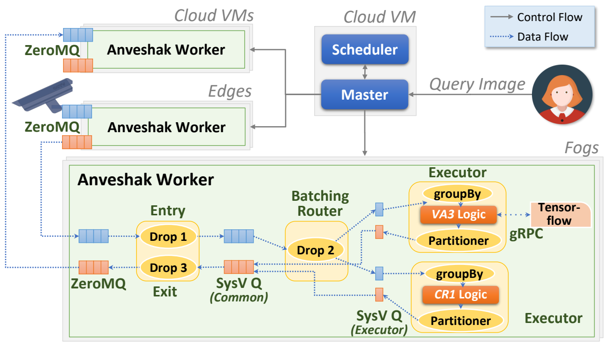


Fig. 3: System Architecture of Anveshak

Each distributed resource available for deploying the dataflow runs a *Worker* process (Fig. 3), which manages module instances on that resource and transfers data between instances on different devices using *ZeroMQ* [34]. The Master initializes module instances on a resource by contacting its Worker. We assume that the required libraries are pre-deployed in the Workers, and in future, this can be replaced by light-weight containers.

A Worker can host multiple FC, VA, CR, etc. module instances with the user logic, and each is encapsulated in a separate *Executor process* (Fig. 3). An *Entry process* copies incoming events arriving over ZeroMQ for a Worker to a common *Sys V* Inter-Process Communication (IPC) queue [35], after dropping delayed events as discussed in § 4.3. From this, a *Router process* retrieves events for a specific Executor, forms a batch using the strategy discussed in § 4.4, and puts it on the Executor's SysV *singleton* queue. The batching triggers when the Executor's previous execution of the module *compute* completes. For each batch placed in its singleton input queue, the Executor invokes the module's *compute* logic on it and generates output events.

The output events are assigned to downstream module instance(s) by calling the module's *partitioner* function defined by the user. This has to be one of the successor module(s) in the static dataflow. Each Worker maintains a lookup table from every deployed module instance to the Worker it is present on, and uses this to route events to those Workers over ZeroMQ. An *Exit* process ensures that delayed events are dropped and not placed in ZeroMQ (§ 4.3). We do not guarantee any ordering across input events arriving at a module instance from different upstream module instances. A Worker can also fetch events from an external endpoint rather than ZeroMQ, such as from the camera for FC instances. Further platform details are in Appendix A.3.

4 Runtime Tuning Strategies

The Anveshak platform operates in a dynamic environment, and needs to be tuned at runtime to adapt to these conditions. We offer a novel *Tuning Triangle* (Fig. 2, bottom right), where users can control the *properties* (corners of the triangle) – *end-to-end latency*, *accuracy* and *camera count scalability* when performing tracking, by modifying *knobs* (shown at the side opposite to a property's corner). The *batching* knob controls the latency property, the *dropping* knob controls the accuracy, and the sophistication of the *tracking logic* knob, already discussed, determines the active camera set size (or scalability). Next, we discuss the two other knobs to control *data drops* and *batching*. Additional discussions on these strategies are provided in Appendix B.

4.1 Approach

We have a captive set of edge, fog and cloud resources having variable compute load due to a changing active set size being processed, and are connected over a MAN/WAN that exhibits dynamism in the latency and bandwidth between resources present on it. So the transient load on resources hosting the active module instances can exceed the available compute or network capacity, which leads to higher event latencies that can cascade up the input event stream.

In such cases, we can gracefully degrade by *dropping events* that cannot be processed within a *maximum tolerable latency* (γ) specified by the user. If we drop potentially stale events early in the dataflow pipeline, we can make make more resources available to the events that are retained and increase their chances of completing within the threshold. This knob helps meet the latency goals and supports a larger active-set size, but it affects the accuracy of tracking if frames containing the entity are dropped. Besides allowing the users to disable dropping, we propose a *smart dropping strategy* in § 4.3 to dynamically vary the accuracy, given a tolerable latency and a peak active camera set size.

For timely processing of the video feeds, it is sufficient for the latency between a frame generated at a camera and its processed response reaching the UV to fall within γ. This can be exploited to enhance the processing throughput by *batching events* passed to the VA/CR modules to amortize the static overheads of invoking the external DL models, while ensuring that the processing latency per event is within permissible limits. However, the time budget available for batching can vary across time, and is non-trivial to estimate without a shared global clock. Besides allowing users to set a fixed batch size, we propose an *adaptive batching strategy* in § 4.4 that maximizes the batch size without violating the latency constraint, for a given accuracy requirement and a peak active camera set size.

Data drops and dynamic batching are featured in stream processing systems. Techniques for load shedding (drops) and batching [36], [37] have been proposed to help determine the the fraction of data to be dropped and the batch size. They use greedy empirical approaches or model it as an optimization problem that is solved using numerical solvers. But they make centralized decisions, are computationally costly and/or expect synchronized device clocks. These are challenging on constrained and wide-area distributed resources. Instead, we design strategies that are lightweight, distributed and resilient to clock-skews.

4.2 Preliminaries

For modeling latency, we decompose the dataflow graph of module instances (tasks) shown in Fig. 2 to a set of sequential task *pipelines*, with a *task selectivity* of 1:1 – the ratio of input to output events. Each sequential pipeline has FC, VA, CR and UV instances, though we assume these are generic tasks, $[\tau_1, \tau_2, ..., \tau_n]$, where τ_1 is the source task and τ_n is the sink. We propose strategies for a single pipeline, which is then generalized to the entire dataflow.

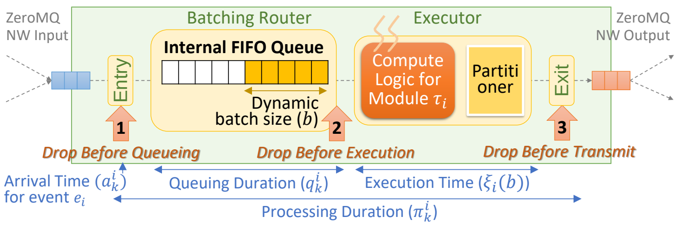


Fig. 4: Event processing at a *Worker*, with batching & drops

Each event e_k arriving at the source task τ_1 of each pipeline is assigned a unique ID k. This ID propagates to all its causal downstream events. Since we have a 1:1 selectivity, an event e_k^i in the pipeline can be uniquely identified by a combination of its source event ID k and the task τ_i it is an input to.

When an event e_k^i arrives at a task τ_i from an upstream task τ_{i-1}, it is placed in a FIFO queue (Fig. 4). Events at the front of the queue are identified by the Executor to form a batch, whose size is dynamically decided, as discussed in § 4.4. The user-logic is triggered on the batch of input events and it returns a batch of output events that is passed to a *partitioner*, which routes each event based on its key to a downstream task.

Let a_k^i indicate the *arrival time* of an event e_k^i at a task τ_i from its upstream task (Fig. 4). This timestamp is measured at the resource hosting the task τ_i. The time spent by the event in the queue before execution is given by the *queuing duration* q_k^i. Once events from the queue are formed into a batch of size, say b, let the function $\xi_i(b)$ give the *estimated execution duration* for the batch by the user-logic for the task τ_i. We assume that the *execution duration* monotonically increases with the batch size, i.e., $\xi(b) < \xi(b+1)$. When $b = 1$, this is a streaming execution with no batching delay. We also define the *processing duration* $\pi_k^i = q_k^i + \xi(b)$, as the time between an event arriving at a task and the resulting output event being placed on its output stream.

We define the *upstream time* for an event e_k^i arriving at task τ_i as $u_k^i = a_k^i - a_k^1$. This is a relative time defined using the timestamps of the source event e_k^1 at the source task and the causal event e_k^i observed at the current task, which in turn depend on their local device clocks κ_1 and κ_i. The arrival time a_k^1 for the source event e_k^1 is propagated to all its causal downstream events in their headers.

While we initially assume all device clocks are synchronized, in Appendix B.3, we discuss how our techniques are resilient to clock-skews between all devices (as is common in MAN/WAN), except those hosting the source and sink tasks of the pipeline, κ_1 and κ_n.

4.3 Strategies to Drop Events

The platform should drop any event e_x^y that cannot reach the last task τ_n before a time $a_x^1 + \gamma$ as it exceeds it maximum tolerable latency γ and is hence *stale*. So a task τ_i may drop an arriving event e_x^i if $a_x^i > a_x^1 + \gamma$. While simple, this waits till the allowed latency is exceeded and does not prevent resource wastage due to execution of tasks prior to the one where the event is dropped. E.g., if at tasks τ_{n-2} and τ_{n-1}, we have $a_x^{n-2} < a_x^1 + \gamma$ and $a_x^{n-1} > a_x^1 + \gamma$, then every event will be processed through the first $(n-2)$ tasks and yet dropped at the $(n-1)^{th}$ task, assuming that the task processing times and network performance stay constant. Ideally, the first task τ_1 should reject a newly arriving event if it *will* be rejected downstream to avoid resource wastage.

We capture the potential staleness of an event at a task τ_j using a *completion budget* β_j. This is the duration allowed for an arriving event to complete processing at this task, including the upstream time spent since its source task, i.e., if $u_k^i + \pi_k^i > \beta_i$ for an event e_k^i, it is stale and can be dropped. Since π_k^i is not known when the event arrives but only after it is queued and executed, this drop decision is taken thrice within a task, as shown in Fig. 4 and described below.

This completion budget for a task can change often during the lifetime of an application as the system reacts to variability. Later, in § 4.5, we discuss how β_i is actively updated to encapsulate this variability. It guarantees that for a given budget, if the downstream tasks do not exhibit further variability, then any event that meets the budget will be processed within γ, and vice versa.

4.3.1 Drop Point 1

The first drop decision is when an event arrives at a task but before it is placed in its input queue (Fig. 4). This checks if the observed upstream time already expended plus the fastest possible execution duration for the event on this task, i.e., using a batch size of $b = 1$, will cause the event to exceed it completion budget, even in the absence of any queuing. Since we do not know the actual queuing delay and batch size for this event at this time, we are conservative in this decision. So events that pass this test may still be dropped at subsequent drop points based on how long they spent in the queue and the actual execution duration.

```
1: procedure DropBeforeQueuing($a_k^1, a_k^i$)
2:     $u_k^i = a_k^i - a_k^1$
3:     if $(u_k^i + \xi_i(1)) \le \beta_i$ then return $false$        ▷ Retain
4:     else return $true$        ▷ Drop this event
5:     end if
6: end procedure
```

4.3.2 Drop Point 2

The second drop point is after the event is queued and put in a batch, but before the batch is executed. At this time, we have a batch of events B of size b, which gives us the expected execution time $\xi_i(b)$, and the queuing duration q_k^i for each of its events. If the predicted time to complete executing this event exceeds the completion budget, i.e., $u_k^i + q_k^i + \xi_i(b) > \beta_i$, we drop this event. The function is passed the entire batch and it returns an updated batch B' without events that should be dropped.

```
1: procedure DropBeforeExec($B[\,], b$)
2:     for $\langle a_k^1, a_k^i, q_k^i, e_k^i \rangle$ in $B$ do
3:         $u_k^i = a_k^i - a_k^1$
4:         if $(u_k^i + q_k^i + \xi_i(b)) \le \beta_i$ then $B' \leftarrow e_k^i$        ▷ Retain
5:         end if
6:     end for
7:     return $B'$        ▷ Events that should be executed
8: end procedure
```

4.3.3 Drop Point 3

It is possible that the actual execution time was longer than estimated. So we trigger the third drop point after the batch execution, where the processing time π_k^i has been spent on an event, but before its output events are sent on the output stream. Here, we check if the generated event e_k^{i+1} at time $u_k^i + \pi_k^i$ has exceeded its completion budget β_i. This drop point is also important if the dataflow has branches, as discussed next.

```
1: procedure DropBeforeTransmit($a_k^1, a_k^i, \pi_k^i$)
2:     $u_k^i = a_k^i - a_k^1$
3:     if $(u_k^i + \pi_k^i) \le \beta_i$ then return $false$        ▷ Retain
4:     else return $true$        ▷ Drop this event
5:     end if
6: end procedure
```

By providing these three light-weight drop points, we achieve fine-grained control in avoiding wasted network or compute resources, and yet perform event drops just-in-time when they are guaranteed to exceed the budget. This balances application accuracy and performance. As a further optimization, we allow the user-logic to flag an event as *avoid drop*, e.g., if it has a positive match, and the platform avoids dropping such events even if they exceed the tolerable latency. This can improve the accuracy and manage the active set size.

Each of the three drop points performs a constant-time comparison operation per event (Line 2 in Drop Point 1, Line 4 in Drop Point 2, Line Line 3 in Drop Point 3), for a time complexity of $\mathcal{O}(1)$ per drop point. In practice, this translates to an overhead of $\approx$ 2–13 ms per event for the ORID App's VA module evaluated in § 5, which is $\approx$ 0.3–4% of the total module execution time for an event.

As shown in Figs. 3 and 4, the drop points are implemented at various point within an Anveshak Worker's execution for a module instance. Drop point 1 is checked by the *Entry* process of a Worker, when it reads an event from the ZeroMQ input, before placing it in the SysV common input queue for the Worker. Drop point 2 is checked by the *Router* when events for a module instance are added to a batch, before the *Executor* invokes *compute* on it. Lastly, drop point 3 is verified by the *Exit* process before the output event is placed in the ZeroMQ output queue to the next Worker.

4.3.4 Non-linear Pipelines

While the drop logic has been defined for a linear pipeline, a module instance (task) in our dataflow can send an event to one of several downstream module instances, based on the partitioning function. However, the destination task for an output event is known only after the partitioner operates on that event, at drop point 3. The completion budget for a task depends on the network and compute performance of the downstream tasks that the event flows through, which can vary for the different task-paths taken. So for each task, we maintain one budget per downstream task.

4.4 Strategies for Dynamic Batching of Events

Batching and executing events in a stream improves the throughput and reduces the average event latency [38]. When events arrive early at a task τ_i and/or the application has a relaxed γ, there may adequate completion budget β_i

to accumulate events from the input queue into a batch and execute them together, while not violating the budget and causing a drop. Since β_i and the input event rates can vary over time, this batch size has to be dynamically decided.

We define the *event deadline* $\delta^i_k = \beta_i + a^1_k$ for an event e^i_k as the time at the task τ_i by which it must complete processing to avoid being dropped. Similarly, we define the *batch deadline* $\Delta^i_p = \min(\delta^i_1, ..., \delta^i_m)$ as the latest time by which the batch B_p having m events must complete execution, and it is defined as the earliest event deadline among all events in the batch. Since temporal event ordering is not assumed, this may not be the first event in the batch.

The batching logic considers the event e^i_x at the head of the queue at the present time t_i for adding to the "current batch" B_p having size m by checking if $t_i + \xi_i(m+1) > \min(\Delta^i_p, \delta^i_x)$, i.e., will adding this event to the batch cause the new execution time of the batch $(t_i+\xi_i(m+1))$ to exceed the deadline of the batch Δ^i_p or the new event δ^i_x. If not, we add the event to the current batch and update the batch deadline. We incrementally check and add events from the queue into the current batch. If the event at the head of the queue cannot be added to the batch, we submit the current batch for execution and add the head event to a new empty batch that becomes the current batch. Even if the queue is empty, the current batch is automatically submitted for execution when the local clock reaches the time, $\Delta^i_p - \xi_i(m)$.

This dynamic batching logic is implemented in the *Batching Router* process of a Worker (Figs. 3 and 4), as it accumulates a batch from the common SysV event queue into a module's SysV input queue.

4.5 Updating the Completion Budget

The *completion budget* β for a task is central to determining the events to be dropped as well as the batch size. To deal with the dynamism in the system, the budget for all tasks must change over time. To enable this, each task τ_i stores a 3-tuple $\langle d^i_k, q^i_k, m^i_k \rangle$ for every event e^i_k it has processed: the *departure time* $d^i_k = u^i_k + \pi^i_k$, which sums the upstream time and the processing duration; the *queuing duration* q^i_k; and the *batch size* m^i_k that the event was part of. Further, each downstream event sent by task τ_i in the pipeline is augmented with two header fields: the *sum of execution times* $\overline{\xi}^i_k = \sum_{j=1..i} \xi_j(m^j_k)$ and the *sum of the queuing delay* $\overline{q}^i_k = \sum_{j=1..i} q^j_k$, spent at the preceding tasks.

As an event executes through the pipeline, we either increase or decrease the budgets for the upstream tasks based on whether the event arrives at the destination task early or is dropped by a task in-between, respectively. The logic used for these budget changes are described next.

4.5.1 Reducing the budget

If an event is processed within its completion budget at a task, it should also complete processing that pipeline within the maximum tolerable latency, if there is no downstream variability. However, if an event e_k gets dropped at task τ_j, it means that the downstream latency has deteriorated and hence, the completion budget of all the upstream tasks $\{\tau_i | i = 1..j-1\}$ must be reduced. If the event has exceeded the completion budget by $\epsilon = d^i_k - \beta_i$, then the sum of the upstream completion budgets must be reduced by ϵ. Intuitively, we reduce the budget at each upstream task τ_i proportional to the time spent in the queue and batch before execution. This causes batches with fewer events to be formed for execution. Using just the queuing time ratio for reducing the budget also avoids penalizing tasks with longer execution times.

Let $\overleftarrow{\lambda}^i_k$ be the duration by which the budget β_i at an upstream task τ_i has to be *reduced* due to an event e^j_k, being dropped at τ_j, where $i < j$.

$$\overleftarrow{\lambda}^i_k = \min(\epsilon \times \frac{q^i_k}{\overline{q}^j_k},\ \ \xi_i(m^i_k) - \xi_i(1)) \quad (1)$$

The first term in the min operator reflects the excess time ϵ scaled by the ratio of the queuing delay for the task relative to the sum of the delays at all the tasks upstream of the dropping task. The second term ensures that the budget reduction does not fall below the minimum possible budget required when streaming the event through with $b = 1$.

Whenever an event is dropped at τ_i, it sends a *reject signal* to its upstream tasks with the event ID k, the excess duration over the budget ϵ and the sum of the queuing delays $\overline{q}^j_k$. The receiving task τ_i combines these with the 3-tuple it maintains for the event to calculate $\overleftarrow{\lambda}^i_k$, and updates its budget as:

$$\beta^{new}_i = \min(d^i_k - \overleftarrow{\lambda}^i_k,\ \ \beta^{old}_i)$$

The first term determines the updated budget as the earlier departure time for that event, less the reduction in budget. Here, the min operator selects the lower of the previous and the new budget to make the model be resilient to out of order accept or reject signals.

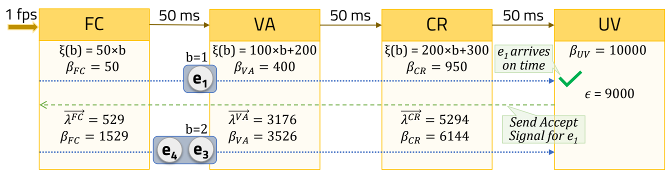
4.5.2 Increasing the Budget

Events that arrive at the final task much earlier than the maximum tolerable latency indicate lost opportunity costs in improving the throughput and scalability of the pipeline by forming larger batches. Therefore, when an event arrives at the final task at $\epsilon = \beta_n - u^i_k$ duration earlier than its completion budget $\beta_n = \gamma$, and this value is greater than some set threshold, ϵ^{max}, the completion budget of the upstream tasks must be increased. Intuitively, we increase the budget of a task proportional to its execution time, relative to the total execution times for all upstream tasks. This gives more weight to tasks with longer execution times, allowing them to increase their throughput which is likely to be the least in the pipeline.

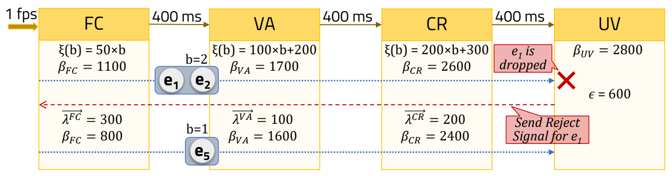
If $\overrightarrow{\lambda}^i_k$ is the duration by which the budget β_i at an upstream task τ_i has to be reduced due to an event e^n_k completing ahead of time at the final task τ_n, then:

$$\overrightarrow{\lambda}^i_k = \min(\epsilon \times \frac{\xi_j(m^i_k)}{\overline{\xi}^{n-1}_k},\ \ (m^{max} - m^i_k) \times \frac{q^i_k}{m^i_k} + \xi_i(m^{max}) - \xi_i(m^i_k)) \quad (2)$$

The first term in the min operator scales the ϵ by the relative time spent in the execution duration for the task τ_i, relative to the execution time at all tasks until (but not including) the final task. The second term ensures that the budget does not exceed the time to create and execute the largest batch size m^{max} allowed by the user. This assumes that the queuing time scales linearly with the number of events. As the prior budget already considers the queuing and execution time for a batch size m^i_k, we subtract it from m^{max}.



(a) Increasing budget on early event



(b) Decreasing budget on delayed event

Fig. 5: Dynamically changing the completion budget in response to early or delayed events. All times are in ms.

A batch will have events with different queuing duration but the same batch execution duration. So some events in the batch will always arrive at the final task before γ elapses. However, we should not increase the budget based on these early events in a batch. Rather, the decision to increase the budget is made only if event with the highest latency in a batch is below $\gamma + \epsilon^{max}$. If so, the task τ_n sends an *accept signal* to all the upstream tasks with the slowest event's ID k, the duration of early arrival ϵ and the sum of upstream execution time, $\overline{\xi}_k^{n-1}$. These are used to calculate the value of $\overrightarrow{\lambda}_k^i$ at tasks τ_i and update their budgets using the 3-tuples for that event: The completion budget for an task τ_j is increases as follows:

$$\beta_i = \max(d_k^i + \overrightarrow{\lambda}_k^i, \ \ \beta_i^{old})$$

As before, selecting the max against the previous budget is to make the model resilient to out of order signals.

The task budgets are increased when an event successfully reaches the final task ahead of time. But transient conditions may cause the system to reduce the budgets to such a low value that no subsequent events flow through to the final task without being dropped. In such cases, even if the conditions improve, the budget may never get updated. To address this, the system periodically sends *probe signals* for every k^{th} event that is dropped at a task τ_j. This probe is forwarded downstream without being dropped. If this signal reaches the final task within γ, then the system calculates and sends the *accept signal* so that the budget for the upstream tasks can be increased and regular events may start flowing through.

Fig. 5a and Fig. 5b illustrate how we use this strategy to increase or decrease the *completion budget*, for the 4 modules of the Anveshak dataflow executing a video feed arriving at 1 fps. For simplicity, the network time for moving events between tasks is static, irrespective of the batch size b, e.g., 50 ms in Fig. 5a. In Fig. 5a, the first event e_1 has completion budgets of $\beta_{FC} = 50\ ms$, $\beta_{VA} = 400\ ms$, $\beta_{CR} = 950\ ms$ and $\beta_{CR} = 10000\ ms$, and it streams through with $b = 1$ to reach UV 9000 ms earlier than the 10 $secs$ allowed. Hence an *accept signal* is propagated to the upstream tasks from UV, with $\epsilon = 9000\ ms$. Using this and the prior states maintained at the tasks, we calculate $\overrightarrow{\lambda}$ using Eqn. 2 and increase their completion budgets. As a result, future events $[e_3, e_4]$ are placed in batches of a larger size $b = 2$, increasing the throughput of the pipeline while still avoiding event delays. Similarly in Fig. 5b, we see that the event e_1 is dropped at UV, and this triggers a *cancel signal* upstream with $\epsilon = 600\ ms$, which is used to calculate $\overleftarrow{\lambda}$ using Eqn. 1 and reduce the completion budget at the prior tasks. This in turn reduces the batch sizes of future events, e.g., from $b = 2$ to $b = 1$ at VA, to ensure the deadline is met.

When *bootstrapping* the application initially, the batch size for all tasks is fixed at $b = 1$ and no budgets are assigned except $\beta_n = \gamma + a_k^1$. Subsequently, when accept or reject signals are triggered, these values are updated (without considering β^{old}) and they stabilize to the new budget.

5 Experiments

We perform targeted and detailed experiments to evaluate the benefits of the domain-sensitive *Tuning Triangle knobs* (Fig. 2, inset) we offer: (1) a smarter tracking logic, (2) dynamic batching capability, and (3) multi-stage dropping strategies. We empirically demonstrate our proposition that these knobs can influence their respective performance properties, and help users achieve a trade-off between them.

5.1 Setup

System Setup. We mimic the resource conditions of 96 Raspberry Pi 3B edge devices on a local cluster, which has 1 *head node* and 10 *compute nodes*. The compute nodes each have an 8-core/16-hyperthread Intel Xeon CPU E5-2620 v4 CPU @2.10 GHz and 64 GB DDR4 RAM, while the head node has the same CPU in a dual socket configuration and 512 GB RAM. Each Xeon CPU core performs comparable to a 4-core Pi 3B, as measured using the CoreMark benchmark. All the nodes have a 1 $Gbps$ Ethernet interface. The nodes run Centos v7.5 with Linux 3.10.0 kernel release, Java 1.8 and Python v3. The head node hosts a Kafka v2.11.0 pub-sub broker for routing input video streams while the compute nodes have PyTorch v1.0.1 and Tensorflow 1.2 [24] installed.

Anveshak Setup[1]**.** We have two Anveshak Workers on each compute node and the head node. The number of FC instances equals the number of cameras used in that experiment, which ranges from 100 – 1000. In addition, we have 10 VA, 10 CR, 1 TL and 1 UV instances. The FC instances are scheduled across the 10 compute nodes in a round-robin manner for load balancing, and run on one of the two Workers on the node. The VA and CR instances are also placed in a round-robin manner on these nodes, on the other Worker. This co-locates a subset of the FC, VA and CR on the same server and minimizes their network transfer overheads. Since each instance runs on a separate Executor process within the Worker, each in-effect runs on a Pi 3B-class CPU core. The TL and UV instances run on a Worker each on the head node.

1. Anveshak source code can be downloaded from http://cds.iisc.ac.in/faculty/simmhan/share/anveshak-1.0.zip

Applications. We implement two tracking applications, *ORID* and *PRID*, described in Table. 1 and evaluate them in our experiments. These omit the QF module given its nascency. Further, we use three TL algorithms for the applications. *TL-All* is a naïve baseline that keeps all the cameras in the network active all the time. *TL-BFS* has access to the underlying road network, but assumes a fixed road-length for all edges when performing the spotlight BFS strategy. *TL-WBFS* is similar, but aware of the exact lengths of each road segment (Alg. 1). Both *TL-BFS* and *TL-WBFS* are configured with the *expected peak walking speed (es)* of the entity being tracked, which varies across experiments. The maximum tolerable latency is set as $\gamma = 15\ secs$. We provide a detailed analysis for ORID below, and report additional empirical discussions and PRID results in Appendix C.

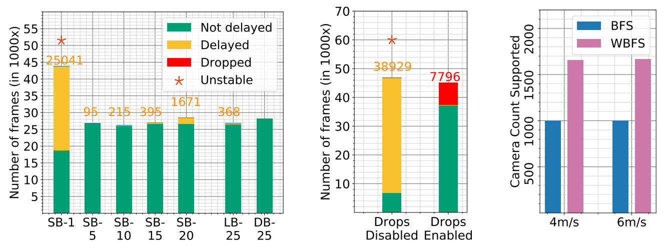
Workload. For the road network, we extracted a circular region of 7 km^2, centered at the Indian Institute of Science, Bangalore campus, from *Open Street Maps* [39]. This has $1,000$ vertices and $2,817$ edges, with an average road length of $84.5\ m$. We use this as the fixed road length for TL-BFS. We use the *CUHK03* Person Re-identification image dataset [13] with $1,360$ unique persons who can be queried for, and $10,531$ images, which provide *true positives or negatives* for the models used. Each JPG image is $64 \times 128\ px$ in size with RGB colors, and a median file size of $2.9\ kB$. Sample images are shown in Fig. 2.

We use these images to simulate video feeds that mimic the movement of the query entity through a road network. The *simulator* takes as input the road network with the road lengths, the speed of the entity being tracked, their starting vertex in the network, and the labeled images for the entity. Cameras are "placed" on all of the road vertices, but may be fewer for some experiments, as reported. We simulate the movement of the entity from the source vertex as a *random walk* at a speed of $1\ m/sec$ ($3.6\ km/hr$). Each camera generates a timestamped feed of images at $1\ fps$ using the true negative images (i.e., images not containing the entity), but uses the true positive person's images for the time intervals when the tracked entity is within the camera's FOV during the walk. For each camera, the simulator publishes its image feed in real-time to a unique topic using the Kafka broker. The FC module for the camera subscribes to its relevant topic to acquire the input stream.

Baseline. We also design a *Lookup-based batching (LB)* baseline to evaluate the effectiveness of our dynamic batching. This uses prior benchmarking on the stable system to determine the smallest batch size that can meet specific input rates without any drops or delays, for rates of 1–1000 $events/sec$, in steps of 10. This forms a *lookup table*. During the application execution, the platform dynamically picks the batch size for the rate closest to the current input rate from this table. Under static system conditions, this strategy will find the best-fit batch size, maximizing the throughput while minimizing the latency, but requires the construction of the lookup table *a priori*.

5.2 Effectiveness of Tuning Triangle

We first show the overall efficacy of the tuning triangle, before offering additional analysis in later sections and in Appendix C. Here, we show how the *batching*, *dropping* and



(a) *Batching vs. latency, es=6 m/sec* (b) *Drops vs. Accuracy, es=7 m/sec* (c) *Tracking Logic vs. Camera Count*

Fig. 6: Distribution of the average end-to-end event latencies for the different batching, dropping and TL strategies

tracking logic knobs have a direct impact on the *latency*, *accuracy* and *camera count scaling* properties, respectively (Fig. 2, inset), and can help improve the application performance.

5.2.1 Batching

In the tuning triangle, the end-to-end latency to process a frame through the dataflow pipeline is affected by the batching strategy, which groups the input events (frames) for a module before executing them using the compute logic. A simple strategy is to use a *static batch size (SB-b)* with b events per batch. But this does not account for variability in input frame rates due to different numbers of cameras being active over time. Another baseline is the *Lookup-based Batching (LB)*, which uses an offline lookup table to select the ideal batch size for a module's input rate. But it does not respond to changes in network performance. Lastly, Anveshak provides a *dynamic batching (DB)* strategy that automatically tunes this knob to help meet the user's latency needs under variable conditions.

To evaluate these, we execute the *ORID application* with 1000 cameras in the road network, a peak entity speed of $es = 6\ m/s$, using BFS tracking logic and with dropping disabled, for a duration of $10\ mins$ and $\approx 600k$ total input frames at $1\ fps$. We evaluate SB with sizes $b =$ 1–20, and LB and DB with a maximum batch size set to $b^{max} = 25$, i.e., SB-b, LB-25 and DB-25. Fig. 6a reports a count of the frames whose end-to-end latency was within the user-specified $\gamma = 15\ secs$, i.e., *not delayed* (green), and those with latency exceeding γ, i.e., *delayed* (yellow, labeled).

SB-1 with one event per batch streams the executing of each event, and delays over $25k$ events since it is *unstable* (marked $\star$). This configuration is not sustainable for the input rate, and causes the input queue to grow exponentially and the latencies of all future events to be delayed. SB-5 is one of the better strategies with only 95 events delayed. While this translates to just 0.3% of delayed events, there are only 21 frames in total containing the entity being queried and they may fall within this. Also, the choice of $b = 5$ performing well is not known *a priori*. Increasing b further for SB causes more events to be delayed, between $215 - 1671$ events for SB-10 – SB-20, since more event per batch increases the queuing latency per event but potentially offers a higher throughput.

Unlike SB, LB adapts to variability in the input rates by automatically changing b at runtime. But it still causes

368 events to be delayed (Fig. 6a, LB-25) as it assumes the network latency is constant. Lastly, Anveshak's DB strategy does not delay any events (Fig. 6a, DB-25). It uses feedback from prior events in the pipeline to automatically set a per-event time budget that picks a near-ideal batch size for each module, which ensures that the frames are not delayed despite network variation. We provide a more detailed analysis of batching in § 5.3.

The total *processed* frames by different batching strategies varies, and is well below the $600k$ input frames. Batching affects the timely detection of a positive or a negative match by the tracking logic and hence the camera activation, which determines if the input frames from a feed flows through the pipeline or not.

5.2.2 Dropping

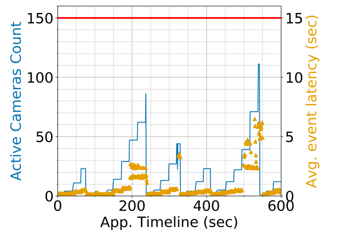
The *data drops knob* affects the accuracy of the application, since dropped events can cause frames having the entity of interest to be missed. But dropping some events may be necessary to ensure that a lot more events are not delayed. To validate this, we run ORID with BFS tracking logic and DB-25, but at a faster entity speed of $es = 7m/sec$, and compare the performance with drops *disabled* and *enabled*. In Fig. 6b, when drops are *disabled*, over $38k$ events are delayed and only 15% of events are on-time. This is also an unstable configuration ($\star$). When we enable drops, 83% of the events are processed on-time. However, $7.8k$ events are dropped in the process, reducing the *accuracy*. This is the trade-off that the *drops* knob provides the users, where much more video frames can be processed within the user's latency γ and at a sustainable rate, but with some of the frames missed in the process. This is discussed in more detail in § 5.5.

5.2.3 Tracking Logic

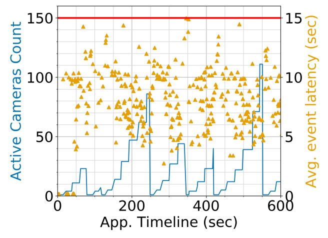
Lastly, the *TL knob* allows users to define or select a suitable camera activation logic that can reduce the number of cameras active for locating and tracking the entity – more cameras that are activated, lower the system *scalability* over the fixed set of compute resources. We run ORID with DB-25 and drops disabled at two entity speeds, $es = 4\ m/s$ and $es = 6\ m/s$. For each, we evaluate BFS and WBFS tracking logic, and measure the peak number of cameras from among the 1000 that they activate. For BFS, a maximum of 111 cameras are active at $es = 4\ m/s$ and 255 are active at $es = 6\ m/s$, while for WBFS, the corresponding camera counts are fewer at 67 and 153. In other words, for the given computing resources, using BFS TL can support a 1000 camera network while using WBFS, we can support a larger $\frac{111}{67} \times 1000 \approx 1657$ and $\frac{255}{153} \times 1000 \approx 1667$ camera network. This scaled comparison is plotted in Fig. 6c. So an intelligent TL can help scale to a larger camera network.

5.3 Analysis of Batching Strategy

The varying number of active cameras and its consequence on the latency motivates the use of variable batch sizes at runtime. Here, we further analyze the benefits of Anveshak's Dynamic Batching (DB-25) against the Lookup-based batching (LB-25). The setup is identical to § 5.2.1, for the ORID App, with TL-BFS, drops disabled, $\gamma = 15\ secs$ and run for $10\ mins$ – except, we use a slower $es = 4\ m/s$.

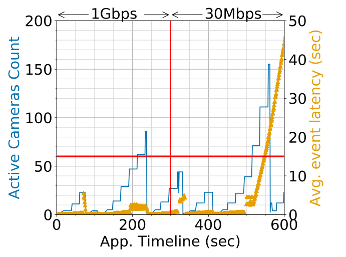


(a) Lookup-based Batching (LB-25)

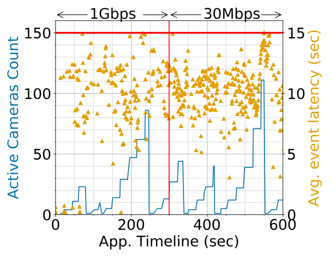


(b) Dynamic Batching (DB-25)

Fig. 7: *# of active cameras* (left Y axis, blue line) and *Avg. end-to-end event latency* (right Y axis, yellow dots) over *Application execution timeline* (X axis) for ORID App using TL-BFS, $es = 4\ m/sec$. Red horizontal line shows $\gamma = 15\ secs$.



(a) Lookup-based Batching (LB-25)



(b) Dynamic Batching (DB-25)

Fig. 8: Adapting to network variation. The system bandwidth drops from $1\ Gbps$ to $30\ Mbps$ after the $300^{th}\ sec$.

Figs. 7a and 7b show the *application timeline* for LB and DB, with the application's wall-clock *execution timeline* (X axis), the *number of active cameras* picked by TL (left Y axis, blue line), and the *end-to-end event latency*, from the source to the sink task averaged for every $1\ sec$ (right Y axis, yellow dots). A red line shows the tolerable latency, $\gamma = 15\ secs$.

We see that there are no delayed events in Anveshak's DB-25 while 90 events are delayed for LB-25, at time points $350\ secs$ and $520\ secs$ (delays not visible due to $1\ sec$ averaging). This is despite LB selecting a best-fit batch size from its lookup table as the system executes. But it assumes that the input rate is uniform for all instances of a module, which does not hold in practice and causes instances receiving a higher rate to use a smaller batch size and hence violate γ. But Anveshak's batching prevents delays in *all cases* as it modulates its batch size per-instance.

LB does offer a low latency distribution, at a median of $0.4\ secs$ due to its selection of batch size of $b = 2$ and $b = 5$, approaching a streaming scenario. The median latency for DB is $7.66\ secs$, with a wide variety of batch sizes and latency values (Fig. 7b). But reducing the latency is not a goal; we ensure that all events reach within γ.

Adapting to network variation. The complexity of Anveshak's batching logic is partly attributed to its ability to respond to network and computation variability. The former is more common in WAN and MAN. We evaluate Anveshak's ability to adapt to even sharp changes in the network performance. Using the same setup for LB-25 and DB-25 as above, we drop the bandwidth between compute nodes from $1\ Gbps$ to $30\ Mbps$ midway through the application execution at $300\ secs$. The timeline plots for LB and DB are shown in Figs. 8a and 8b.

The first 300 $secs$ is identical to the earlier plots, and neither configuration has event delays. But once the bandwidth drops, DB keeps the system stable with no event delays as it reacts to event latencies increasing. As the network degrades, the budget available to each task reduces, and DB forms smaller batches. E.g., the median CR batch size rapidly drops from $b = 8$ to 5, and the number of batches with 1 and 2 events rise from only $\approx 18\%$ before 300 $secs$ to $\approx 30\%$ after the network slowdown. LB, however, becomes unstable beyond 500 $secs$. This is due to its lookup table being created for a certain system and network performance and that not holding at runtime.

5.4 Analysis of Tracking Logic

We further analyze and compare the *TL-BFS* and *TL-WBFS* tracking logic for the $es = 4\ m/s$ setup of ORID App with static batching, drops disabled and $\gamma = 15\ secs$, against *TL-All*, a baseline logic that keeps all cameras active, similar to contemporary systems. Since the resources are inadequate to support all 1000 cameras being active for TL-All, we do two runs, with 100 and 200 cameras placed on a proportionally smaller road network, and all active. For TL-All, we use a static batch size of $b = 20$, which offers the best configuration, while for TL-BFS, we try two setups, SB-1 and SB-20, and use SB-1 for TL-WBFS.

Fig. 9a plots the application timeline (X axis) and the event latency averaged over 1 sec (right Y axis) for the 100 and 200 cameras of *TL-All*. While the event latency is stable without any delays for 100 cameras with a median latency of 2.80 $secs$), it is unstable and grows rapidly with 200 active cameras, indicating inadequate resources. The total frames processed is $\approx 60k$ in the former, and $\approx 120k$ in the latter with over 55% delayed. Obviously, this tracking logic does not scale to 1000 cameras.

For *TL-BFS* operating on 1000 cameras, we show a similar timeline in Figs. 9b and 9c, and also plot the active camera count (left Y axis). The *SB*-1 setup has a low median latency at 218 $ms \ll \gamma$. But the latency occasionally exceeds γ (for 25 events), when the active camera count is > 100. The camera count (Fig. 9b, blue line) has a saw-tooth behavior – the spotlight logic increases the active set of cameras when the entity is in a blindspot, and drops this to 1 when it is reacquired by an active camera. At $\approx 550\ secs$, the entity is in a blindspot long enough that the count spikes to 111 cameras, stressing the available resources and causing the latency to grow to 16.8 $secs$. This is due to CR, whose DNN is the slowest task and supports only 8.33 $events/sec$ per Executor instance. When feeds from 111 active cameras at 1 fps are mapped to 10 CR instances, 8 of these receive more than 8.33 $events/sec$, and cause the latency spike.

For *TL-BFS* with *SB*-20, the median latency has increased to 3.65 $secs$ (Fig. 9c). But even with static batching, this improved tracking logic does not have any delayed events. Interestingly, in periods where the active camera count increases, like between 140–240 $secs$, the mean latency decreases – more cameras means a higher input rate, which fills up a batch and triggers it faster.

Similarly, the more advanced TL strategy *TL-WBFS* supports 1000 total cameras on the same set of resources, and has a stable latency even with SB-1 where events stream through. Its median latency of 291 ms is lower than BFS SB-20 and comparable to BFS SB-1, but with no events delayed. The active camera count grows in more granular steps using WBFS since it is aware of the road lengths and leads to a more measured growth of active cameras. Further, its peak active camera count is 67, relative to 111 when using TL-BFS. So WBFS can help scale to a larger set of total cameras or for a longer period of the entity being in a blindspot.

While a better TL helps, it is not a substitute for dynamic batching since we can have scenarios where a static batch is not adequate. E.g., for a faster $es = 6\ m/sec$, TL-BFS with SB-20 causes 603 events to be delayed, compared to no delays using dynamic batching (not shown).

5.5 Analysis of Dropping Strategy

Even TL and dynamic batching may not suffice when the spotlight grows large. This can cause the resources to be overwhelmed, latencies to grow unabated, and cascade to all future events. Anveshak's smart dropping strategy is beneficial here, causing events to drop early in order to avoid resource wastage, and reduce overall event delays.

We examine the results from § 5.2.2 in more detail, where we run ORID with TL-BFS and DB-25 at $es = 7\ m/s$. In Fig. 10a, we report a timeline plot of the active camera count (left Y axis) and the average event latency (right Y axis) for the experiment, with drops enabled and disabled. The red line indicates $\gamma = 15\ secs$ allowed latency.

When the entity moves faster, the spotlight also grows faster when it is in a blindspot. Under such conditions, when *drops are disabled*, we see that the latency grows sharply $\gg \gamma$ as the active cameras grow from 100–500. This causes each CR instance to receive a peak of $\approx 49\ events/sec$, while its processing capacity is only 19 $events/sec$. This causes 85% of events to be delayed (Fig. 6b), and the App is unstable.

When *drops are enabled*, the application's latency is stable and within $\gamma = 15\ secs$ even when the active camera count grows as high as 389 (Fig. 10b). The drops start (far right Y axis, red dots) when the camera count exceeds 200, which matches $\approx 20\ events/sec$ for each CR task. While 17% of all events are dropped, the rest of the events are processed without any delays (Fig. 6b). Dropping frames containing the entity can delay locating the entity and cause the active set to grow. However, none of the 21 frames carrying the detected entity is dropped. This is merely incidental, but enabling the *do not drop* flag will ensure this. The batch sizes for VA and CR are smaller here (not shown) than for $es = 4\ m/sec$ with dynamic batching but no drops. When the input event rate is high, the system first reduces the queuing time by forming smaller batches and then resorts to drops.

6 Related Work

6.1 Video Surveillance Systems

Intelligent Video surveillance systems have been designed for machine learning, pattern analysis and data management of video footage [9]. These span various generations. The first generation systems only capture and store analog data; the second generation introduce CV algorithms

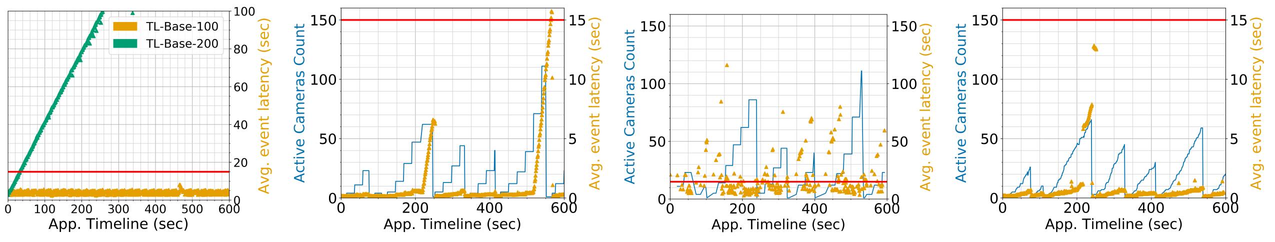

(a) TL-All (SB-20) (b) TL-BFS (SB-1) (c) TL-BFS (SB-20) (d) TL-WBFS (SB-1)

Fig. 9: Effect of *tracking logic* on performance of ORID, with $es = 4\ m/sec$, static batching and drops disabled

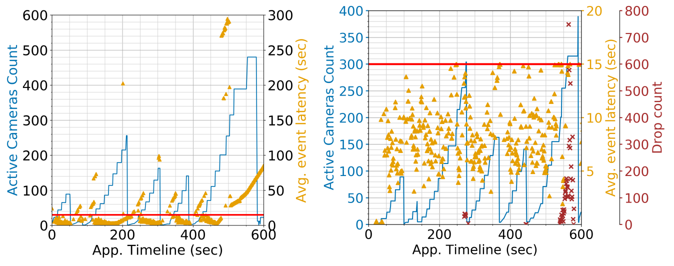

(a) Drops Disabled (b) Drops Enabled

Fig. 10: Perf. with *drops dis/enabled*, *TL-BFS* and *es=7 m/sec*

applied centrally; and the third generation supports automated wide area surveillance, with distributed intelligence and data fusion from multiple cameras and other sensors.

ADVISOR [8] supports tracking, crowd counting and behavioral analysis over camera feeds from train stations to assist human operators. But these are pre-defined applications, run centrally on a private data center and process all camera feeds all the time. *IBM's Smart Surveillance System (S3)* [18] is a proprietary platform for video data management, analysis and real-time alerts. While it offers limited composition of modules, it too performs centralized analysis and does not consider performance optimizations. Early works examine edge computing for basic preprocessing [40]. But the edge logic is static, and the rest of the analytics done centrally using dedicated networks.

Several frameworks have been developed specifically for distributed video analytics across the edge, fog and cloud resources. The *Ella Middleware* uses a publish-subscribe model with hierarchical brokers to route video and event streams between analytics deployed on edge devices. However, their platform design resembles a general-purpose event-driven middleware, without any specific analytics support or runtime optimizations for video processing, unlike us. *EdgeEye* [41] efficiently deploys DNN models on the edge, using a JavaScript API for users to specify their parameters. It offers performance optimizations for DNNs, but does not consider distributed systems issues, such as batching, dropping and network variability.

Video Storm [42] is a video analytics system with the goals of approximation and delay tolerance. It schedules video analytics query workloads on a cluster of machines, where each query has a deadline and a priority. *VideoEdge* [19] extends this to support scheduling on a hierarchy of edge, fog and cloud resources. Both these provide tuning knobs which are conceptually similar to our ours. But the key distinction is that these degrees of freedom requires the specification of objective functions to define the impact of the knobs on metrics. This makes it challenging to use out of the box. Our domain-sensitive Tuning Triangle intuitively captures the impact of the 3 well-defined knobs on the 3 metrics that impact tracking applications the most.

More recently, RES [43] tackles the problem of running video analytics using edge and cloud resources while meeting Quality of Service (QoS) constraints. They identify filtration and identification phases to split the tasks across edge and cloud, with three types of operations: basic, filter and machine learning. In contrast, our modules are better-tuned for entity tracking across a many-camera network, with distinctive modules such as tracking logic and query fusion. Our runtime also offers mechanisms to deal with network variability. Hu et al. [44] develop specialized image recognition algorithms for video surveillance using mobile edges to achieve high accuracy and low recognition time. These can be incorporated within Anveshak to facilitate deployment and scalable execution across city-scale camera networks. VU [45] identifies camera feeds in a many-camera network which are not useful due to occlusion or blurring, and drops such feeds from being processed. Such techniques can complement our tracking and dropping strategies. Privacy preserving video analytics platforms is an active area of research [46]. While not a primary goal, we provide privacy benefits by processing most video feeds on local edge and fog devices.

6.2 Big Data platforms and DSL

General purpose dataflow models such as ORCC [20] and Apache NiFi [21] give programmers the flexibility to compose complex applications using logic blocks, often providing pre-defined blocks and a graphical UI. These are then compiled and executed within a runtime environment. Similarly, Big Data stream processing platforms like *Apache Storm*, *Flink* and *Spark Streaming* [22], [33], [47] offer flexible dataflow composition. Instead, we define a domain-specific dataflow pattern for tracking applications, with a fixed dataflow composition from the 7 modules. But users provide the logic for each module that matches the given module signatures. This curtails flexibility but allows users to rapidly design, upgrade and deploy a variety of tracking scenarios, incorporating contemporary advances in DNNs.

Google's TensorFlow is a DSL for defining DNNs and CV algorithms, and to deploy trained models for inference [24]. However, TensorFlow is not meant for composing arbitrary modules together. The tasks take a Tensor as an input and give a Tensor as the output, and there are no native patterns such as Map and Reduce to ease composability. Yahoo's *TensorFlow on Spark* [48] adds flexibility by allowing Spark's Executors to feed RDD data into TensorFlow. Thus, users can couple Spark's operations with TensorFlow's neural networks. But Anveshak is at a level of abstraction higher, allowing for rapid development of tracking applications with fewer lines of code or sometimes just a configuration change. Also, Spark is not designed for distributed computing on WANs or edge/fog devices, which we address in the Anveshak runtime.

6.3 Streaming Performance Management

There are several performance optimizations adopted by stream processing systems, which we extend. *Apache Flink* [33] and *Storm* [47] support *back-pressure*, where a slow task sends signals to its upstream task to reduce its input rate. This may eventually lead to data drops, but the data being dropped are the new ones generated upstream rather than the stale ones that are already in-flight. This sacrifices freshness in favor of fairness. Our drops prioritize recent events over stale events, and importantly, adjust the budget more precisely.

Google's Millwheel [49] uses the concept of *low watermarks* to determine the progress of the system, defined as the timestamp of the oldest unprocessed event in the system. It guarantees that no event older than the watermark may enter the system. Watermarks can thus be used to trigger computations, such as window operations, safely. While our batching and drop strategies are similar, watermarks cannot budget the time left for a message in the pipeline and has no notion of user-defined latency.

Aurora [36] adopts *load shedding*, which is similar to our data drops. They define QoS as a multidimensional function, including attributes such as response time, similar to our maximum latency. Given this function, the objective is to maximize the QoS. *Borealis* [37] extends this to a distributed setup. Anveshak uses multiple drop points even within a task, which offers fine-grained control and robustness. Features like "do not drop" and resilience to clock skews are other domain and system-specific optimizations.

7 Conclusions

In this paper, we have proposed an intuitive domain-specific dataflow model for composing distributed object tracking applications over a many-camera network. Besides offering an expressive and concise pattern, we surface the Tracking Logic module as a powerful abstraction that can perform intelligent tracking and manage the active cameras. This enhances the scalability of the application and makes efficient use of resources. Further, we offer tunable runtime strategies for dropping and batching that help users easily balance between the goals of performance, accuracy and scalability. Our design is sensitive to the unique needs of a many-camera tracking domain and for distributed edge, fog and cloud resources on wide-area networks. Our experiments validate these for a real-tracking application on deployments of up to 1000 cameras.

As future work, we plan to explore intelligent scheduling of the module instances on edge, fog and cloud resources; allow modules to be dynamically replaced for better accuracy or performance; and handle mobile camera platforms such as drones. In a real setting, multiple objects of interest would be tracked across the camera network. This should lead to interesting scheduling problems as well as an opportunity to share compute across multiple queries. A practical deployment of Anveshak would use containers for dependency management. However, co-locating containers can lead to interference and QoS violations [15]. It would be worth exploring models to estimate such performance interference, which can influence the execution time estimates used in the batching and dropping strategies. It will also be useful to support camera-specific DNNs to handle, say, crowded scenes that may be visible to specific cameras.

Acknowledgments

This work was supported by grant number 4(16)/2019-ITEA from the Ministry of Electronic and Information Technology (MeitY), Government of India. The first author was supported by a fellowship from the Robert Bosch Center for Cyber-Physical Systems, Indian Institute of Science, Bangalore.

References

[1] G. Ananthanarayanan, P. Bahl, P. Bodik, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *IEEE Computer*, vol. 50, 2017.

[2] L. Wang and D. Sng, "Deep learning algorithms with applications to video analytics for a smart city: A survey," *arXiv preprint arXiv:1512.03131*, 2015.

[3] A. Khochare, P. Ravindra, S. P. Reddy, and Y. Simmhan, "Distributed video analytics across edge and cloud using echo," in *International Conference on Service Oriented Computing (ICSOC) Demo*, 2017.

[4] A. Bedagkar-Gala and S. K. Shah, "A survey of approaches and trends in person re-identification," *Image and Vision Computing*, vol. 32, no. 4, 2014.

[5] P. Natarajan, P. K. Atrey, and M. Kankanhalli, "Multi-camera coordination and control in surveillance systems: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 11, no. 4, 2015.

[6] X. Liu, W. Liu, H. Ma, and H. Fu, "Large-scale vehicle re-identification in urban surveillance videos," in *IEEE International Conference on Multimedia and Expo*, 2016.

[7] N. Murthy, R. K. Sarvadevabhatla, R. V. Babu, and A. Chakraborty, "Deep sequential multi-camera feature fusion for person re-identification," *arXiv preprint arXiv:1807.07295*, 2018.

[8] N. T. Siebel and S. Maybank, "The advisor visual surveillance system," in *Workshop applications of computer vision*, 2004.

[9] M. Valera and S. A. Velastin, "Intelligent distributed surveillance systems: a review," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 152, no. 2, 2005.

[10] M. K. Lim, S. Tang, and C. S. Chan, "isurveillance: Intelligent framework for multiple events detection in surveillance videos," *Expert Systems with Applications*, vol. 41, no. 10, 2014.

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

[12] L. Esterle, P. R. Lewis, M. Bogdanski, B. Rinner, and X. Yao, "A socio-economic approach to online vision graph generation and handover in distributed smart camera networks," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[13] W. Li, R. Zhao, T. Xiao, and X. Wang, "Deepreid: Deep filter pairing neural network for person re-identification," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[14] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," *Software: Practice and Experience*, vol. 50, no. 5, 2019.

[15] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study," *IEEE Access*, vol. 5, 2017.

[16] P. G. Lopez *et al.*, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Reviews*, vol. 45, no. 5, 2015.

[17] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, 2015.

[18] C.-F. Shu, A. Hampapur, M. Lu, L. Brown, J. Connell, A. Senior, and Y. Tian, "Ibm smart surveillance system (s3)," in *IEEE Conference on Advanced Video and Signal Based Surveillance*, 2005.

[19] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *IEEE Symposium on Edge Computing*, 2018.

[20] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *ACM International Conference on Multimedia*, 2013.

[21] "Apache nifi," https://nifi.apache.org/.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *USENIX Workshop on Hot Topics in Cloud Computing*, vol. 10, no. 10-10, 2010.

[23] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[24] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[25] K. Shiva Kumar, K. Ramakrishnan, and G. Rathna, "Distributed person of interest tracking in camera networks," in *ACM International Conference on Distributed Smart Cameras (ICDSC)*, 2017.

[26] Tong Xiao, "Open-ReID," https://cysu.github.io/open-reid/.

[27] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

[28] E. Ahmed, M. Jones, and T. K. Marks, "An improved deep learning architecture for person re-identification," in *IEEE Conference on Computer Vision and Pattern Recognitione (CVPR)*, 2015.

[29] 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*, vol. 20, no. 1, 2018.

[30] L. Ren, J. Lu, Z. Wang, Q. Tian, and J. Zhou, "Collaborative deep reinforcement learning for multi-object tracking," in *European Conference on Computer Vision (ECCV)*, 2018.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[32] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI Conference on Artificial Intelligence*, 2017.

[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Bulletin of the Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[34] F. Akgul, *ZeroMQ*. Packt Publishing, 2013.

[35] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[36] D. J. Abadi *et al.*, "Aurora: a new model and architecture for data stream management," *International Journal on Very Large Data Bases (VLDB)*, vol. 12, no. 2, 2003.

[37] D.J. Abadi *et. al.*, "The design of the borealis stream processing engine." in *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[38] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *CoRR*, vol. abs/1605.07678, 2016.

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

[40] A. Kornecki, "Middleware for distributed video surveillance," *IEEE Distributed Systems Online*, vol. 9, no. 2, 2008.

[41] P. Liu, B. Qi, and S. Banerjee, "Edgeeye: An edge service framework for real-time intelligent video analytics," in *International Workshop on Edge Systems, Analytics and Networking*, 2018.

[42] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[43] M. Ali, A. Anjum, O. Rana, A. R. Zamani, D. Balouek-Thomert, and M. Parashar, "Res: Real-time video stream analytics using edge enhanced clouds," *IEEE Transactions on Cloud Computing (TCC)*, 2020.

[44] H. Hu, H. Shan, C. Wang, T. Sun, X. Zhen, K. Yang, L. Yu, Z. Zhang, and T. Q. Quek, "Video surveillance on mobile edge networks–a reinforcement learning based approach," *IEEE Internet of Things Journal (IoTJ)*, vol. 7, no. 6, 2020.

[45] H. Sun, W. Shi, X. Liang, and Y. Yu, "Vu: Edge computing-enabled video usefulness detection and its application in large-scale video surveillance systems," *IEEE Internet of Things Journal (IoTJ)*, vol. 7, no. 2, 2019.

[46] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor: Privacy-preserving video analytics as a cloud service," in *USENIX Security Symposium*, 2020.

[47] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *ACM International Conference on Management of Data (SIGMOD)*, 2014.

[48] Yahoo, "Tensorflow on Spark," https://github.com/yahoo/TensorFlowOnSpark/wiki, accessed: 2018/06/16.

[49] T. Akidau *et al.*, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 11, 2013.

[50] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

Aakash Khochare Aakash is a Ph.D. candidate at the Indian Institute of Science, Bangalore. His research involves designing systems, abstractions and heuristics that enable analytics on edge, fog and drone platforms. He is a recipient of the IEEE TCSC SCALE Challenge Award in 2019.

Yogesh Simmhan Yogesh Simmhan is an Associate Professor and a Swarna Jayanti Fellow at the Indian Institute of Science, Bangalore. His research explores abstractions, algorithms and applications on distributed systems, including Cloud and Edge Computing, Scalable Graph Processing, and Distributed storage and analytics to support Big Data and Internet of Things (IoT) applications. He is an Associate Editor-in-Chief of the Journal of Parallel and Distributed Systems (JPDC), and earlier served as an Associate Editor of IEEE Transactions on Cloud Computing. Yogesh has a Ph.D. in Computer Science from Indiana University, and was previously a research faculty at the University of Southern California (USC), and a Postdoc at Microsoft Research. He is a Senior Member of the IEEE and the ACM.

Appendix A
Additional Details on Dataflow and Runtime

A.1 Dataflow Composition Details

In § 2.3 of the main article, we concisely described how the Anveshak domain specific dataflow is used to compose the OpenReID (ORID) application, and described the *compute* method of it modules. Here, in the Appendix, we provide additional details about the Anveshak dataflow composition, and the specification of tracking applications by the user. We use the PersonReID (PRID) to illustrate this.

As seen in Fig. 2 of the article, the Anveshak dataflow consists of 6 types of modules, *FC, VA, CR, TL, QF,* and *UV*. Each module exposes three methods *init*, *compute*, and *partition* for the programmer to implement. Additionally, the *FC*, *VA*, and *CR* modules also expose an *update* method. The signatures of the functions are listed as part of the pseudocode for each module provided for the PRID App, in Algs. 2 – 5.

A.1.1 Functions within a Module

Init. Each module maintains a *local state* object that the programmer can use as a local data store for variables and counters across several invocations of the *compute* method, for different event batches it receives. The *init* should contain logic to initialize the *local state* object. We also recommend loading the DNN model into memory (neural network architecture and weights) as part of the *init* method, in the local process or as part of a local gRPC service, and storing a reference to the network in the *local state* object. This avoids the overhead of loading the DNN into memory for every *compute* invocation.

Update. In a single tracking lifecyle, the *init* method is invoked once, while the *compute* and *partition* methods are invoked once per input and output event batch, respectively. The *update* method is invoked if a module receives a feedback *signal* from a downstream module, e.g., it would be invoked on the *FC* module if it is sent a signal by the *TL* module to activate/de-activate/otherwise control the camera.

Partition. Each module can have several instances running during execution time to allow data parallelism and the use of distributed edge and fog resources, e.g., in Fig. 2 of the article, we show two instances each of the *VA* and *CR* modules, connected to four instances of *FC* and 2 instances of *VA*, respectively. The *partition* method of a module instance selects the instance of the subsequent downstream module to which an output event is sent. Here, the event is in the form of a *key-value* pair, and partition function provided by the user operates on the *key* and returns a module instance ID. This allows any upstream module instance to send an output event to any successor module instance, based on the key. Multiple keys can be mapped to the same instance. For modules connected to multiple downstream modules, such as CR to TL, UV and QF, multiple partitioners are defined and a copy of the output event forked to each module by the Executor. This is similar to the *Stream groupings* concept in Apache Storm [2], and like Storm, Anveshak users can use it to route event with specific keys or types to specific module instances so that it operates on all events of that key or type. This is useful when the module maintains local state for that key across multiple invocations, and/or is performing an aggregation of events of that type or key.

Compute. The *compute* function's signature depends on the module, and is invoked for each batch of input events that the module's instance receives from upstream module instances. The output events from the *compute* function of every module is defined as a *key,value* pair. The *FC* module's *compute* function gets images as input from the camera endpoint and operates on a single image value at a time. Typically, the state of the camera (*active/inactive*) is stored in the *local state*, and the logic implemented simply accesses this boolean state and determines if the frame must be emitted as an output value or not. The *update* method of the *FC* module can alter the local state of the camera, such as changing *isActive* from true to false, or vice-versa.

The *VA* module's *compute* function operates on a single key and a list of values. Here, the key for an input event is typically set to be the camera ID by FC in its output event. Hence, *VA*'s compute operates on a batch of images belonging to a single camera in one invocation. The output of *VA* is a list of events with key-value pairs that the programmer is free to determine in their compute logic. This is similar to the *map* or *flatMap* functions offered by Hadoop MapReduce and Spark. The *compute* function of *CR*, *TL* and *QF* modules also operate on a list of key-value pairs as their inputs. The output of these modules is also a list of key-value pairs.

A.1.2 Dataflow routing

The output of *FC* is routed to the *VA* module, while the output of *VA* goes to the *CR* module. The downstream module instance to route to is determined by the *partition* method of the upstream module instance. The programmer can implement logic that uses the *key* field of each output event to determine the downstream module instance. However, at times, this decision may be independent of the key being used as well, e.g., for a round-robin routing across instances or if just a single downstream instance is present.

The output signal of the *TL* module is routed to the *FC* module, and the specific FC instances depend on the cameras being activated or deactivated. The output signal of the *QF* module is forked and routed to all instances of the *VA* and *CR* modules. It is worth noting that the output signals of the *TL* and *QF* modules are processed using the *update* method in the *FC, VA* and *CR* modules, unlike output events which are operated by the *compute* method.

Recall that in § 4.2 of the main article, we mention that events have to be *causally related* for data drops and dynamic batching to work. However, once the input key-value pairs are passed to the user logic in *compute*, the user is free to manipulate them as they wish. To retain the causality, when *emitting* the output key-value pairs, the user must also specify which input key(s) where used to derive the output. This provides for provenance of data and allows the platform to maintain the causal linkage between events.

2. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., "Storm@ twitter," in ACM SIGMOD, 2014

A.1.3 Application Specification

The tracking application developer first implements the aforementioned functions in *Python*. They then explicitly specify the Python files containing the module implementations as part of a *YAML specification file*, that is illustrated in Fig. 11. The also provide additional configuration parameters for each module and for the overall application. When the application is being deployed, Anveshak parses the YAML file, loads the appropriate functions for each module instance, and connects the instances to ensure that the data can flow between the modules.

For each module, the *count*, *filename* and *class* fields are mandatory. The *count* is an integer that specifies the number of module instances Anveshak must create for a given module type. Since each instance runs on a separate Executor process, the number of instances for a module is indicative of the compute load expected on that module. The *filename* and *class* uniquely identify the user's implementation of the module, based on the above interfaces. Optionally, each module type can also specify *constructor_args* and *state* fields. As the name suggests, *constructor_args* is a key-value dictionary passed as an initialization argument to the constructor of the user's module implementation class. In the example YAML in Fig. 11, we see that the path to the pre-trained model is passed as *checkpoint_dir* key in the *constructor_args*. The *state* field is used to initialize the *local state* stored in each module.

For simplicity, we provide the *entity query* being tracked as part of the YAML, though in a practical deployment, this would be passed at runtime to the *VA*, *CR* and *QF* modules for each input query. The *query* field has characteristics of the tracked entity and the query tuning knob, such as their sample image, the maximum batch size b^{max} for dynamic batching, the maximum tolerable latency γ, etc.

Once defined and composed, updating the tracking application is simple. The user can simply modify the *filename* and *class* fields of this YAML file to select alternative implementations of the *VA*, *CR*, *TL* modules, which may offer different trade-offs between compute requirement against accuracy.

A.2 Composition of the PRID Application

The *Person ReID (PRID) Application* is similar to the *ORID* Application described in § 2.3 of the main article. The key difference between these two Apps is that the *CR* and the *TL* modules are different, while the *FC* and *VA* modules are identical. In fact, the TL logic can also be swapped between the two, as it is generic across applications. Here, we describe all the major functions for each PRID module, not just the *compute* logic.

Alg. 2 outlines the implementation of *init*, *compute*, *partition* and *update* methods for the *FC* module. Line 2 in the *init* method sets a key in the *state* variable to indicate if the camera connected to the *FC* is active. The *compute* method simply returns this state. If the *compute* method of *FC* returns *true*, the module forwards the Camera ID as the key and the image as the value to the *partitioner*. The state variable is updated by *TL* using the *update* method. Finally, we demonstrate a round-robin partitioner in the *partition* method. Here, the key sent to the partitioner is ignored.

```
filter:
      count: 1000
      filename: filter.py
      class: UserFilter
video_analytics:
      count: 10
      filename: analytics.py
      class: UserAnalytics
contention_resolution:
      count: 10
      filename: resolution.py
      class: ContentionResolution
      constructor_args:
            checkpoint_dir: <--path to the
                        trained DNN model-->
            port: 8080
tracking_logic:
      count: 1
      filename: domain_update_default.py
      class: UserDomainUpdate
      constructor_args:
            graphpath: <--path to the city
                     graph-->
            domaindata:
                  spotlight_size: 1
                  speed_of_walk: 4
      state:
            activeSet:
                  - <--node-id-->
visualization_unit:
      count: 1
      filename: visualization_unit.py
      class: UserVisualizationUnit
query:
      query_id: 1
      query_image_path: <--URI for the entity
                    image-->
      upper_limit_batch_size: 25
      max_tolerable_latency: 15
```

Fig. 11: Sample YAML file for defining a tracking application using Anveshak

Data is sent to the downstream *CR* instances in a round robin manner to spread the load across them.

Alg. 3 specifies the logic of *init*, *compute* and *partition* methods for the *VA* module. Line 2 in *compute* directly invokes the *Histogram of Gradients* function in the *OpenCV* Python library [3], which serves as a pedestrian detector. The output of the detector is a list of bounding boxes around pedestrians per image. The *compute* emits the Camera ID as the key and the image along with the list of bounding boxes as the value. Here, the *partitioner* uses the key passed to the *partitioner* to decide the downstream module instance of *CR*. This ensures that data of the same key (camera) is passed to the same downstream CR instance.

3. N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in IEEE CVPR, 2005

Algorithm 2 Pseudocode for FC of PRID

```
1: procedure INIT($constructor\_args$)
2:     $state.put('isActive') \leftarrow constructor\_args.get('isActive')$
3:     return $state$
4: end procedure

1: procedure COMPUTE($img, state$)
2:     return $state.get('isActive')$
3: end procedure

1: procedure PARTITION($key, state, list\_downstream\_mods$)
2:     next ← state.get('dmodule')
3:     next = (next + 1)% len(list_downstream_mods)
4:     state.put('dmodule') = next
5:     return list_downstream_mods[next]
6: end procedure

1: procedure UPDATE($key, value$)
2:     $state.put('isActive') \leftarrow value$
3:     return $state$
4: end procedure
```

Alg. 4 provides the pseudo-code of *init*, *compute* and *partition* methods for the *CR* module. In the *init* method, the query image is loaded into memory and the a local *gRPC* server on the *port* specified in the *constructor_args* is created for *Person ReID Tensorflow DNN* [4]. A *gRPC* channel is created to the server launched on this port. The *compute* method receives a batch of camera ids, images and bounding boxes as the input. First, the image is cropped into smaller images based on the bounding boxes (Line 6). Then Line 9 invokes the *PROCESS* method in the local gRPC server that was spawned in *init* and which wraps the PersonReID DNN. The *PROCESS* method of the gRPC server simply invokes the Tensorflow DNN for detections. This DNN returns detections of the image matches in the cropped frames, if any. The output key is still the camera ID and output values are the input image and a boolean flag that reflects if the image matches the query or not. The *partitioner* used is round-robin.

Alg. 5 describes the implementation of *init*, *compute* and *partition* methods for the *TL* module. In the *init* method, we load the road-network graph into memory and save it in the *state* dictionary. The *compute* method then fetches this graph along with other characteristics, such as the last known location of the entity and timestamp, to help expand the search space as part of the spotlight camera activation logic. For PRID, the graph only has the road network structure and does not have the length of the roads (edges). Hence, in Alg. 5 *compute*, Line 7, invokes an *(unweighted) breadth first search (BFS)* on the graph starting from the last known location of the person. Recall that in ORID, we performed a *weighted breadth first search (WBFS)*. Finally, we use a hash-based *partitioner* for the TL module that maps from camera ID to the FC module to signal the relevant FCs about their de/activation.

4. E. Ahmed, M. Jones, and T. K. Marks, "An improved deep learning architecture for person re-identification," in IEEE CVPR, 2015

Algorithm 3 Pseudocode for VA of PRID

```
1: procedure INIT($constructor\_args$)
2:     return $state$
3: end procedure

1: procedure COMPUTE($C_{id}, imgs[\,], state$)
2:     $bbs[\,][\,]$ = OPENCV.HOG(imgs[ ]))
3:     for $img$ in $imgs[\,]$ and $outbbs[\,]$ in $bbs[\,][\,]$ do
4:         EMIT($C_{id}, \langle img, outbbs[\,]\rangle$))
5:     end for
6: end procedure

1: procedure PARTITION($key, state, list\_downstream\_mods$)
2:     return list_downstream_mods[hash(key) % len(list_downstream_mods)]
3: end procedure
```

Algorithm 4 Pseudocode for CR of PRID

```
1: procedure INIT($constructor\_args$)
2:     $path \leftarrow constructor\_args.get('checkpoint\_dir')$
3:     $query \leftarrow load(constructor\_args.get('query\_image\_path')$
4:     $port \leftarrow constructor\_args.get('port')$
5:     $grpc\_channel \leftarrow grpc.new\_channel(port)$
6:     $state.put('grpc') \leftarrow grpc\_channel$
7:     $state.put('entity\_query\_img') \leftarrow query$
8:     TFLOW.DNN_CR.LOAD($path$)
9:     $grpc.start\_server(port, TFlow.DNN\_CR)$
10:    return $state$
11: end procedure

1: procedure COMPUTE($\langle C_{id}, img, outbbs[\,]\rangle[\,], state$)
2:     $query = state.get('entity\_query\_img')$
3:     $channel \leftarrow state.get('grpc')$
4:     $cropped = [\,]$
5:     for $tuple$ in $\langle img, outbbs[\,]\rangle[\,]$ do
6:         $cropped\_img$ = CROP($img, outbbs[\,]$)
7:         $cropped.append(cropped\_img)$
8:     end for
9:     $detections$ = CHANNEL.PROCESS($cropped, query$)
10:    for $was\_detected$ in $detections[\,]$ do
11:        EMIT($C_{id}, \langle img, was\_detected\rangle$)
12:    end for
13: end procedure

1: procedure PARTITION($key, state, list\_downstream\_mods$)
2:     next ← state.get('dmodule')
3:     next = (next + 1)% len(list_downstream_mods)
4:     state.put('dmodule') = next
5:     return list_downstream_mods[next]
6: end procedure

1: procedure PROCESS($cropped, query$)
2:     $detections$ = TFLOW.DNN_CR($cropped, query$)
3:     return $detections$
4: end procedure
```

A.3 Platform Runtime Details for Anveshak

Here, we complement § 3 of the main article that introduced the Anveshak runtime platform with more internal details of its architecture and execution model.

The Anveshak platform comprises of two logical components – Anveshak *Master* and Anveshak *Worker* (Fig. 3 in main article). Both the components are implemented in Python 3.7. Anveshak *Master* runs in the cloud and exposes an endpoint for users to submit and deploy an application on the platform. The *Master* accepts the application in the

Algorithm 5 Pseudocode for TL-BFS of PRID

```
1: procedure INIT($constructor\_args$)
2:     $road\_network \leftarrow load(constructor\_args('graphPath')$
3:     $state.put('road\_network') \leftarrow road\_network$
4:     return $state$
5: end procedure
```

```
1: procedure COMPUTE($\langle C_{id}, \langle img, detections[\,]\rangle\rangle[\,], state$ )
2:     $el$ = GETENTITYLOCATION($\langle C_{id}, detections[\,]\rangle[\,]$)
3:     if $el == \varnothing$ then            ▷ Entity lost. Expand spotlight...
4:         $graph = state.get('road\_network')$
5:         $lsl = state.get('lastSeenLocation')$
6:         $lst = state.get('lastSeenTime')$
7:         $cameras[\,]$ = BFS($graph, lsl, lst$)
8:         EXPANDSEARCHSPACE($cameras$)
9:     else
10:        SHRINKSEARCHSPACE($el$)
11:    end if
12: end procedure
```

```
1: procedure PARTITION($key, state, list\_downstream\_mods$)
2:     return list_downstream_mods[hash(key) % len(list_downstream_mods)]
3: end procedure
```

YAML format described above. The *Master* process parses the YAML to determine the number of instances defined for each module. It then invokes the *Scheduler* logic, which determines the placement of these module instances onto Anveshak *Workers*. Each *Worker* runs on a edge, fog or cloud resource, and forms the unit of resource scheduling.

The *Worker* is a multi-process component that executes and routes the key-value streams among the module instances. We choose a multi-processed approach over a multi-threaded approach to circumvent Python's multi-threading shortcomings due to the *Global Interpreter Lock (GIL)*. The *Worker* spawns one *Entry/Exit* process for managing the input and output of events between instances, *over the network*. This process exposes a ZeroMQ connection for other *Workers* to transfer events as key-value pairs.

The "Entry" part of the *Entry/Exit* process implements the logic for Drop Point 1, and the "Exit" part of the process enacts Drop Point 3 (Fig. 3 in main article). Events from upstream module instances flow through the ZeroMQ endpoint and are first tested for staleness using Drop Point 1. Events that are not dropped at Drop Point 1 get transferred to the *Batch and Routing* process in the *Worker* using a SysV IPC queue. The *Routing* process implements logic for dynamic batching as well as for Drop Point 2 (Fig. 4 in main article). The Router uses singleton SysV IPC input and output queues (i.e., queues with capacity for just a single entry) as a means to interface with an *Executor* process that is responsible for executing a batch and returning a response. When a *Executor* Process is free, it signals the Router using the output queue and the Router dynamically creates a batch and transfers it to the *Executor* Process using the singleton input SysV queue.

One Worker may spawn and manage multiple *Executor* processes. However, each *Executor* corresponds to exactly one module instance. This architecture allows us to potentially spawn several *Executors* on a single device, if the device has many cores or if the user logic has lightweight resource needs on a device. The *Executor* process first performs a "group by key" if this is a VA module instance, and then invokes *compute* or *update* user logic, appropriately. Finally, the *Executor* invokes the *partioner* on the output key-value pair events, based on which the output events get assigned to one of the next module instances to which the event must be transferred to.

The *Executor* then places this batch of output events into another singleton output queue between the *Executor* and the *Router*. This also signals the *Router* that the *Executor* is free and a new batch can be transferred to it. The *Router* process uses another SysV output queue to send data back to the *Entry/Exit* process. Finally, the logic for Drop Point 3 is verified by the *Entry/Exit* Process before performing a network transfer over ZeroMQ to a remote module instance. The SysV input/output queues between the *Entry/Exit* Process and the *Batch and Routing* Process are common for all the *Executor* Processes spawned in the Worker. Meanwhile, between the *Batch and Routing* Process and the *Executor* Process, one singleton queue is maintained per *Executor*.

A.4 Application Implementation Details

For both the ORID and PRID applications, the VA module uses HoG pedestrian Descriptor that is available as part of the OpenCV library. The features are classified into pedestrians and non-pedestrians using an SVM detector. For the ORID App, the CR module uses the Inception model [50] from the OpenReId library, whose DNN architecture is shown Fig. 12. The network is trained on CUHK-03 dataset and has $23M$ parameters. For the PRID App, the CR module uses a Person ReID module with $2.3M$ parameters, trained on the same CUHK03 dataset. For both the applications we use pre-trained models available in the public domain for evaluation.

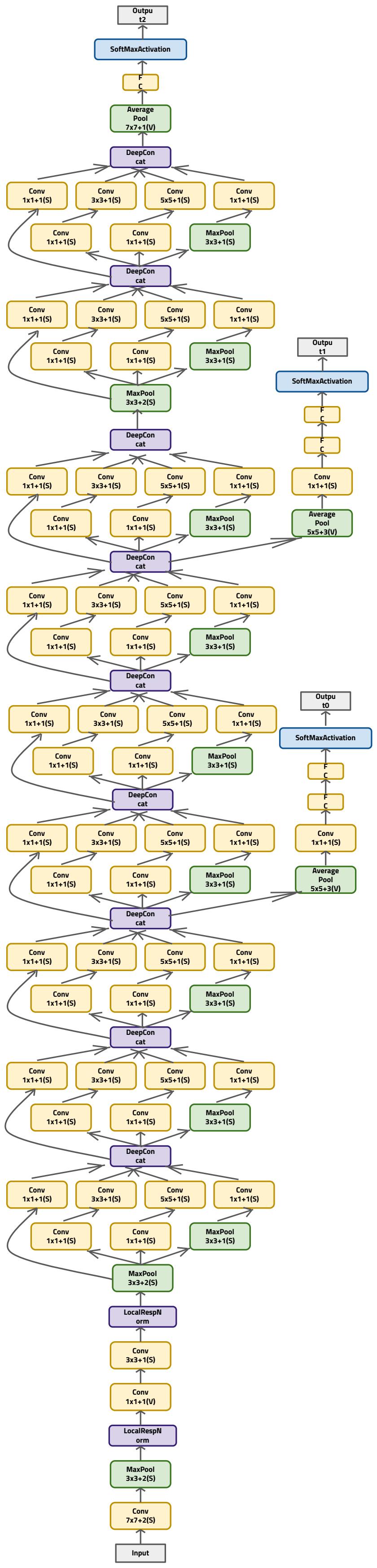


Fig. 12: Inception DNN Architecture using as CR module in ORID App [50]

Appendix B
Additional Details on Runtime Strategies

In this Appendix, we offer more details on various aspects of the runtime tuning strategies that were proposed in § 4 of the main article.

B.1 Formal Bounds for Batch Size and Drop Rate

While our batching is not based on a fixed batch size but rather adapts to the events that arrive, we can formally bound the batch size and drop rate under certain assumptions. Later, we relax some of these assumptions.

We derive bounds on the batch size and drop rate working under the assumption that a dataflow has constant known input rate ω, 1:1 selectivity, no pipelining, the execution time matches the expectation, and the network and compute conditions are static. For simplicity during analysis, we also assume temporal ordering of events and no pipelining of the FIFO queue with the execution.

Batch Size. Here, the goal is to estimate the bounds for the *batch size* m_i at a task τ_i when it has access to the budget and other variables at runtime, under a stable state. The inter-arrival rate between successive events can be written as $a^i_k - a^i_{k+1} = \frac{1}{\omega} \quad \forall k \in \mathbb{N}, i \in n$. Also, due to the temporal ordering of events, the batch deadline is bound by its first event. Then, m_i is the largest Integer value such that:

$$(m_i - 1) \times \frac{1}{\omega} + \xi_i(m) \leq \beta_i - u^i_1 \qquad \text{and} \qquad \xi_i(m_i) \leq \frac{\beta_i - u^i_1}{2}$$

In the first equation, we capture the intuition that the time to queue up the batch and to execute it [5] must not exceed the batch processing deadline. The second equation ensures the stability of the system such that the time to execute a batch does not exceed the deadline for the next batch being accumulated, i.e., execution time for the current batch should be less than the queuing time for next batch. Here, ω and $\xi_i(m)$ are unconstrained natural numbers while β_i and u_i are available at runtime. A solution for m_i may not exist within these constraints, which means that the input rate ω is unsustainable. In such cases, events should be dropped.

Drop Rate. If no solution for m_i exists above, then we find the *drop rate* of events, $(\omega - \omega^{max})$, relative to the largest stable input rate, ω^{max}, that can be support, and the associated batch size. The goal is to maximize m_i and ω^{max} such that:

$$(m_i - 1) \times \frac{1}{\omega^{max}} + \xi_i(m_i) \leq \beta_i - u^i_1 \qquad \text{and}$$

$$\xi_i(m_i) \leq \frac{\beta_i - u^i_1}{2}$$

Compared to streaming execution with $m = 1$, batching adds latency to the overall event processing time while increasing the throughput. We quantify the increase in the *average latency* per event caused by batching, relative to streaming, for a task τ_i as:

$$\frac{m_i - 1}{2 \times \omega} + \xi_i(m_i) - \xi_i(1)$$

Here, the first term is the average queuing time for the m_i events in the batch and the latter indicate the execution time for the batch.

5. There is no queuing delay for the first event, hence $(m - 1) \times \frac{1}{\omega}$. For simplicity, we assume that the estimated execution time ξ_i equals the actual execution time.

B.2 Discussion on Event Ordering

The asynchronous nature of distributed stream processing and network variability makes it hard to enforce temporal order on events arriving from multiple upstream tasks on different devices at a downstream task [6]. Our programming model and runtime do not enforce *temporal ordering* across events arriving from different upstream module instances at a downstream instance. While *temporal ordering* may be required for some vision algorithms such as estimating optical flow [7], the user logic can reorder events in a batch using the timestamps passed in the *key*, *value* pairs. The runtime can enforce ordering in the *group* stage of every module using a temporal window and watermarks [8]. However, we currently do not include such mechanisms.

B.3 Resilience to Unsynchronized Clocks

Devices in a WAN may have unmanaged devices that do not have tightly-synchronized clocks [9]. While our drop and batch decisions are based on the timestamps at the different devices, these have been designed to withstand skews across the device clocks. The drop logic as well as the completion budget are based on the *upstream time*, with the other time variables within a task being defined relative to it. So making the upstream time resilient to unsynchronized clocks with consequently address all other time calculations.

Let $\kappa_1..\kappa_n$ be the clocks for the n devices hosting the tasks in a pipeline. As stated before, we require that $\kappa_1 = \kappa_n$. Let the signed-values $\sigma_2..\sigma_{n-1}$ be the skew between the clocks $\kappa_2..\kappa_{n-1}$ relative to κ_1 and κ_n, i.e., $\sigma_i = \kappa_i - \kappa_1$. The upstream time $u^i_k = a^i_k - a^1_k$ for an event e^i_k arriving at a task τ_i. When corrected for the skew, we have $\widetilde{u}^i_k = (a^i_k - \sigma_i) - a^1_k$

Similarly the update rule for the *completion budget* $\widetilde{\beta}_i$ for task τ_i, when corrected for skew, can be written by correcting the *departure time* as $(\widetilde{d}^i_k - \overleftarrow{\lambda}^i_k)$ or $(\widetilde{d}^i_k + \overrightarrow{\lambda}^i_k)$, as is the case when updating using a reject or an accept signal. Here, $\widetilde{d}^i_k = \widetilde{u}^i_k + \pi^i_k = u^i_k - \sigma_i + \pi^i_k$, since π is a duration calculated locally within a single device. Also, λ, the budget change factor, depends on ϵ and other local durations, with $\widetilde{\epsilon} = \widetilde{d}^i_k - \widetilde{\beta}_i = \epsilon$. Hence $\widetilde{\beta}_i = \beta_i - \sigma_i$, and it can be shown that $\widetilde{\beta}^{old}_i = \beta^{old}_i - \sigma_i$ with an inductive argument.

As a result, in all three of our drop points, when replacing u^i_k and β_i with their skew-correct forms (in lines 3, 4 and 3, respectively), we have a $-\sigma_i$ term added symmetrically to both the left and the right sides of the comparisons, which cancel each other out. This shows that our drop logic is resilient to clock-skews.

For batching, we use the test $\big(t_i + \xi_i(m+1) > \min(\Delta^i_p, \delta^i_x)\big)$ to decide if we should add an event e^i_x arriving at task τ_i to a batch B_p. Here too we can correct the skew for the time points $\widetilde{t}_i = t_i - \sigma_i$ and $\widetilde{\delta}^i_x = (\widetilde{\beta}_i + a^1_k) =$

6. J. R. Perez Cruz and S. E. Pomares Hernandez, “Temporal alignment model for data streams in wireless sensor networks based on causal dependencies,” International Journal of Distributed Sensor Networks, vol. 10, no. 3, p. 938698, 2014

7. D. Fleet and Y. Weiss, “Optical flow estimation,” in Handbook of mathematical models in computer vision. Springer, 2006

8. T. Akidau et al., “Millwheel: fault-tolerant stream processing at internet scale,” The VLDB Journal, vol. 6, no. 11, 2013

9. F. Buchholz and B. Tjaden, “A brief study of time,” Digital Investigation, vol. 4, 2007

$(\beta_i - \sigma_i + a_k^1) = (\delta_x^i - \sigma_i)$; similarly, $\widetilde{\Delta}_p^i = \Delta_p^i - \sigma_i$ as it derives from $\widetilde{\delta}$. As we see, the $-\sigma_i$ term is added to both the sides of the comparison and cancel each other out, indicating that the batch size is resilient to unsynchronized clocks as well.

B.4 Pseudo-code for Updating the Completion Budget

Updating the *completion budget* is a key runtime capability that allows Anveshak to handle network and compute variations. If the event reaches the *UV* module too early, the *completion budget* at each module traversed by the event can be increased, thereby allowing the future events to spend more time in queuing and execution, leading to a larger batch size and better throughput. On the contrary, if the event reaches late, beyond γ, the *completion budget* must be downgraded to ensure smaller batch sizes and lower upstream latency.

In Alg. 6, we provide at the pseudo-code for *updating* the *completion budget*, to complement the description and illustration in § 4.5 of the main article.

Algorithm 6 Algorithm to update the completion budget

```
1: procedure UpdateBudget($d_k^i, q_k^i, m_k^i, \epsilon$)
2:     if reject signal then
3:         calculate $\overleftarrow{\lambda}_k^i$ using Eqn. 3
4:         return $\min(d_k^i - \overleftarrow{\lambda}_k^i,\ \beta_i^{old})$
5:     else
6:         calculate $\overrightarrow{\lambda}_k^i$ using Eqn. 4
7:         return $\max(d_k^i + \overrightarrow{\lambda}_k^i,\ \beta_i^{old})$
8:     end if
9: end procedure
```

$$\overleftarrow{\lambda}_k^i = \min(\epsilon \times \frac{q_k^i}{\overline{q}_k^j},\ \ \xi_i(m_k^i) - \xi_i(1)) \tag{3}$$

$$\overrightarrow{\lambda}_k^i = \min(\epsilon \times \frac{\xi_j(m_k^i)}{\overline{\xi}_k^{n-1}},\ \ (m^{max} - m_k^i) \times \frac{q_k^i}{m_k^i} + \xi_i(m^{max}) - \xi_i(m_k^i)) \tag{4}$$

Appendix C
Additional Experiments

In this Appendix, we offer additional experiments and analysis of the ORID Application discussed in § 5 of the main article, and also provide an evaluation of Anveshak using the PRID application.

C.1 Analysis of Anveshak's Batching

We first examine the benefits of *dynamic batching* for the ORID application, while keeping the tracking logic fixed at *TL-BFS* and *disabling drops*. Further, the entity's *peak speed* is configured as $es = 4\ m/sec$ with TL-BFS. This is set based on an estimated peak speed since underestimating can cause the entity to be lost due to a slow *Rate of Expansion (RoE)* of the spotlight when the entity is in a blindspot; too high a value can cause a fast spotlight RoE and large active camera count that overwhelms resources or causes drops

C.1.1 Need for Batching

The Anveshak dataflow can be executed in a streaming manner without batching, i.e., a batch size of $b = 1$. But this sacrifices the input throughput, and hence scalability of the number of active cameras that can be supported..

Fig. 15a shows the application timeline plot for the streaming configuration, using a static batch of size $b = 1$ (SB-1). Here, the X axis is the wall-clock time (secs) of the tracking application's execution, the blue line on the left Y axis is the number of active cameras as decided by TL, and the yellow dots on the right Y axis are the end-to-end latency of events (frames) from the source to the sink averaged over every second of the application's execution time. A red horizontal line shows the maximum tolerable latency, $\gamma = 15\ secs$. Further, Fig. 13a shows a violin distribution of these 1 sec average event latencies for the streaming and other batching configurations. Similarly, Fig. 14a shows the number of events ($1000\times$) processed within the deadline of γ against those that were delayed (orange, labeled).

In Fig. 13a, streaming (SB-1) exhibits the lowest median latency, at $\approx 200ms$, much below $\gamma = 15\ secs$ that is acceptable. However, this is at the cost of the latency occasionally exceeding the threshold, as seen from the violin outliers, and the 25 delayed events in Fig. 14a. In fact, if we configure TL with an entity peak speed of $es = 6\ m/s$, 57% of the input events exceed γ when streaming (Fig. 14b).

As Fig. 15a shows, these delayed frames occur when the active camera count exceeds 100. Here, the blue lines exhibit a saw-tooth behavior – the spotlight logic increases the active set of cameras when the entity is in a blindspot, and this drops to 1 when the entity is reacquired within the FOV of an active camera. At $\approx 550\ secs$, the entity is in a blindspot long enough that the count spikes to 111 cameras, stressing the available resources and causing the latency to grow to 16.8 $secs$.

Specifically, this latency is caused by CR, whose DNN is the slowest task the dataflow at 120 $ms/event$ and can service $\mu = \frac{1000}{120} = 8.33\ events/sec$ per CR instance. For an event arrival rate $\lambda > \mu$, the queue in unstable and the queuing delays will grow exponentially. At the 550^{th} second, the feeds from 111 active cameras at 1 fps are mapped to 10 CR instances, and 8 of these CR instances

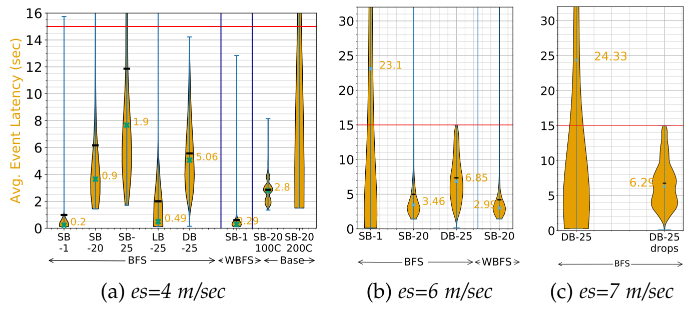


Fig. 13: Distribution of the average end-to-end event latencies for the different batching and TL strategies

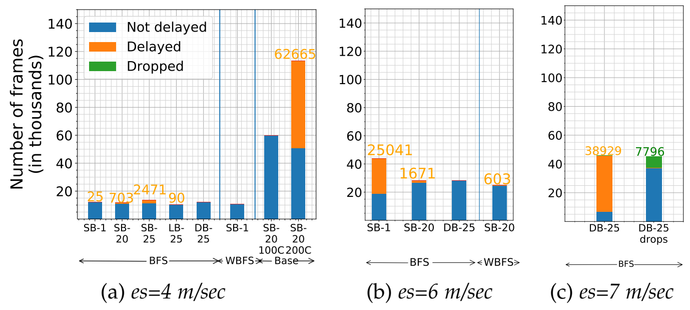


Fig. 14: Events with latency $\leq \gamma$ vs. Delayed vs. Dropped events, for different peak speeds, batching and TL strategies

receive events faster than μ causing a spike in the event latency. However, in this case, the system is able to recover at $\approx 570\ secs$ when the active set size drops to 1.

In summary, it is vital to process *all* the frames in a *timely* manner rather than *many* in a *fast* manner, i.e., taking up to but below γ is better. Batching offers such efficiencies and helps scale to a larger number of active cameras.

C.1.2 Limitations of Static Batching

While batching improves the system throughput, using a fixed or static batch size has two limitations. There can be a missed detection of the entity or certain frames can experience a very high end-to-end latency. Next, we provide empirical evidence for this.

We first set the static batch size $b = 20$ for all tasks, and report its latency, active camera count and events processed without delay in Figs. 15b, 13a and 14a, as before. We see that the median latency has increased to 3.65 $secs$ in Fig. 13a, and there are no sharp growths in the latency in Fig. 15b which would indicate an unstable queue. Interestingly, in periods where the active camera count increases, like between 140–240 $secs$, the mean latency decreases – more cameras means a higher input rate, which fills up a batch and triggers it faster, thus reducing batching delays.

But better stability does not always result is fewer latency violations. In Fig. 14a, we see 6% or 703 events exceed γ, more than the streaming case. Since the batch size is fixed, *there is no bound* on the time spent by an event waiting for the batch to be completed before execution. This causes events to be delayed. However, if using $es = 6\ m/s$, none of the input events exceed γ (Fig. 14b). This too indicates that a fixed value of the batch size will not suffice for all situations.

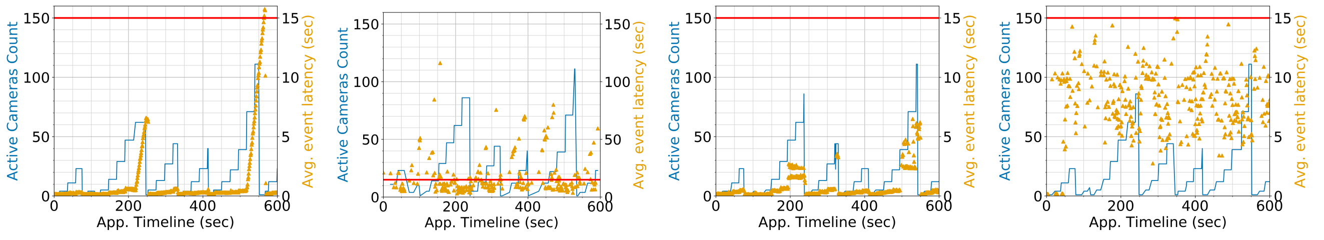

(a) Streaming ($b = 1$) (b) Static Batching (b=20) (c) LB Batching (b^{max}=25) (d) Anveshak Batching (b^{max}=25)

Fig. 15: # *of active cameras* (left Y axis, blue line) and *Avg. end-to-end event latency* (right Y axis, yellow dots) over *Application execution timeline* (X axis) for different batching strategies, TL-BFS, $es = 4\ m/sec$. Red horizontal line shows $\gamma = 15\ secs$.

Selecting an appropriate peak latency γ and peak speed es by the domain user is also important.

Another consequence of event delays due to a large fixed batch size is that TL grows the spotlight larger than necessary as it receives negative detections of the entity late, and also delays shrinking the spotlight as positive detections arrive late. These put further stress on the resources. E.g., at the 220th second, the active set size is 62, 86 and 139 using a batch sizes of $b = 1, 20$ and 25 (latter is not plotted), due to the increase in median latency from 0.2 sec to 3.65 $secs$ and 7.6 sec (Fig. 13a). When $b = 25$, there is a delay of 22 $secs$ in detecting a missing entity, due to which the active camera count sharply jumps to 139, and also causes a detection of the entity in the neighboring cameras to be missed. In fact, 22% of events are delayed for $b = 25$ (Fig. 14a).

C.1.3 Benefits of Dynamic Batching

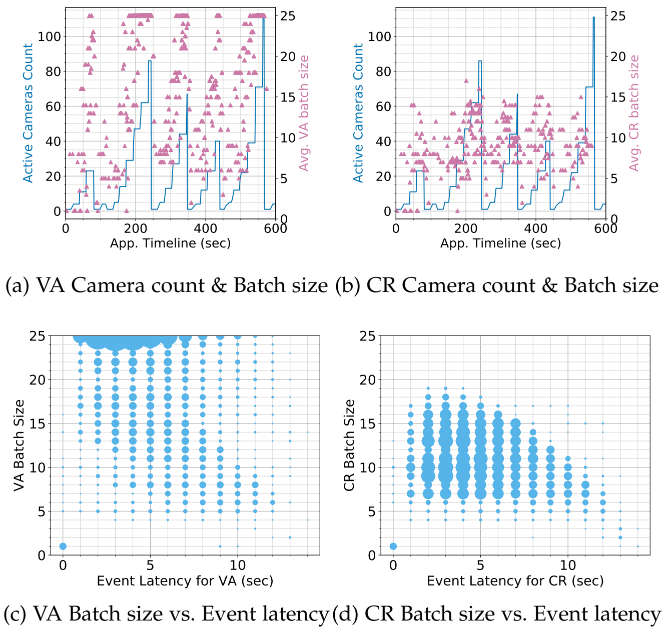
The varying number of active cameras and its consequence on the latency is a strong motivation for using a dynamic batch size. Here, we compare Anveshak's *dynamic batching (DB)* strategy with *Lookup-based Batching (LB)*, also called *Near-optimal Baseline (NOB)*, which also changes the batching strategy at runtime based on a lookup table created from micro-benchmarks. The maximum batch size is limited to $b^{max} = 25$ for both.

The timeline plot for LB-25 (NOB-25) and our DB-25 are shown in Figs. 15c and 15d, with their latency distribution and delay frames listed in Figs. 13a and 14a. They key observation is that there are no delayed events in Anveshak's DB while 90 events are delayed for LB, at time periods 350 $secs$ and 520 $secs$. This is despite LB selecting a best-fit batch size from its lookup table as the system executes. But it assumes that the input rate is uniform for all instances of a module, which does not hold in practice and causes instances receiving a higher rate to use a smaller batch size and hence violate γ. But Anveshak's batching prevents delays in *all cases*.

LB does offer a lower latency distribution, at a median of 0.4 $secs$ (Fig. 13a, LB-25). The batch size it chooses is often between 2 and 5, thus approaching a streaming scenario. The median latency for Anveshak's batching is 7.66 $secs$, with a wide variety of batch sizes. But as discussed before, reducing the latency is not a goal; we just need to ensure that all events reach within γ.

C.1.4 Analysis of Anveshak's Dynamic Batching

We further analyze the behavior of Anveshak's dynamic batching by examining the two key tasks that dominate the execution latency, VA and CR. The dynamic batch sizing operates independently for these two tasks types.


(a) VA Camera count & Batch size (b) CR Camera count & Batch size

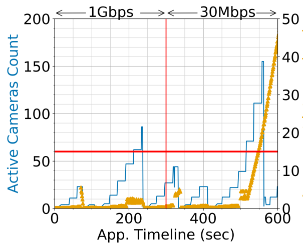
(c) VA Batch size vs. Event latency (d) CR Batch size vs. Event latency

Fig. 16: Performance of Anveshak batching, TL-BFS, *es=4 m/s*

Figs. 16a and 16b show the active cameras count (left Y axis) and the batch size averaged every 1 sec (right Y axis), for all 10 VA and CR tasks for the application execution time-time (X Axis), while Figs. 16c and 16d show a bubble (scatter) plot of the task latency per event against the batch size the event was part of, for all VA and CR instances.

For the *VA task*, we see from Fig. 16a that the batch size increases as the active camera count grows, helping it support a higher input event rate. VA uses almost every batch size between 1 and 25 (Fig. 16c), and the latency varies within a single batch size – events in a batch that arrive later will have a lower latency, and vice versa. For the larger batch sizes, the task latency often ranges from 2–6 $secs$, indicating that the VA module can benefit from a more relaxed b^{max}.

The *CR task* shows a similar trend between the camera count and batch size in Fig. 16b. CR has a lower mean batch size than VA, which is expected since the CR module is a compute intensive DNN which has a larger execution time than the VA module. Fig. 16b which is a timeline plot for mean number of CR shows a similar trend. Also, despite


(a) LB Batching (b^{max}=25)

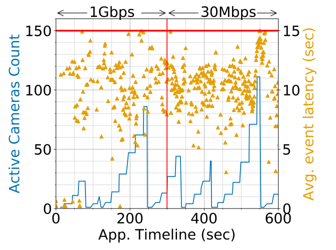

(b) Anveshak Batching (b^{max}=25)

Fig. 17: Adapting to network variation. The system bandwidth drops from 1 $Gbps$ to 30 $Mbps$ after the 300th sec.

$b^{max} = 25$, its peak batch size never exceeds 19.

On further analysis, the maximum budget allocated to a CR task is $\beta = 3.65$ $secs$. At the peak active camera count of 111, this task receives events from 13 cameras. For forming a batch of size $b = 25$, we have its queuing time as 1.92 $secs$, assuming a uniform input rate of 13 $events/sec$, and execution time of $\xi(25) = 1.74$ $secs$, which together at 3.66 $secs$ exceed the budget. Instead, the dynamic batch size selected of $b = 19$ results in a processing time of 2.91 $secs$, which is within the budget.This indicates that Anveshak's dynamic batch sizing is sensitive to the needs of individual tasks. In fact, even with a peak entity speed of $es = 6$ m/sec, it avoids any event delays (Fig. 14b, DB-25).

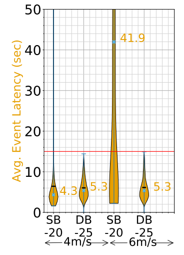
C.1.5 Adapting to network variation

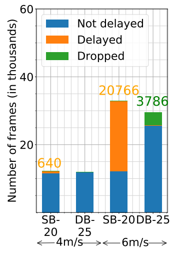
The complexity of Anveshak's batching logic is partly attributed to its ability to respond to handling network and computation variability. The former is more common in WAN and MAN. We evaluate its ability to adapt to even sharp changes in the network performance. Using the dynamic setup for DB and LB from Figs. 15c and 15d, we drop the bandwidth between compute nodes from 1 $Gbps$ to 30 $Mbps$ midway through the application execution at 300 $secs$ into the timeline. These are shown in Figs. 17a and 17b for Anveshak and LB.

The first 300 $secs$ is identical to the earlier plots, and neither configuration has event delays. But once the bandwidth drops, Anveshak manages to keep the system stable with no event delay as it reacts to event latencies increasing. As the network degrades, the budget available to tasks reduce, causing smaller batch sizes to be formed. The median CR batch size rapidly drops from $b = 8$ to 5, and the batches with 1 and 2 events rise from only $\approx 18\%$ before 300 $secs$ to $\approx 30\%$ after the network slowdown. However, LB becomes unstable and its event latency grows beyond γ after 500 $secs$. This is due to its lookup table being created for a certain system performance and that not holding at runtime.

C.2 Analysis of PRID App

In this section, we perform a subset of the ORID experiments for the PRID App to reconfirm the tuning triangle trends. They key difference between the ORID and PRID applications is CR, with the logic for PRID using a more accurate and compute-intensive DNN that takes $\approx 63\%$ longer to process each frame than for ORID. We use the same road network, entity query, 1000-camera setup, $\gamma = 15$ $secs$,


(a) Latency, $4m/sec$


(b) Delays, $4m/sec$

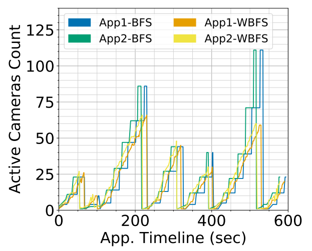

(c) Active camera count for ORID App (App 1) vs. PRID App (App 2) over time

Fig. 18: Latency distribution, event delays and camera count for different configurations of *PRID App*, using *TL-BFS*

BFS tracking logic, drops disabled and peak entity speed of $es = 4$ m/s by default, unless mentioned otherwise. Figs. 18 show the *latency distribution* and the *number of delayed events* for PRID App, for various runs, while Fig. 18c shows the *active camera count* for ORID App (App 1) and PRID App (App 2) using BFS and WBFS tracking logic along the application timeline.

Using a static batch size of $b = 20$, we observe a median latency of 4.33 $secs$ but with $\approx 5\%$ latency violation (Figs. 18a and 18b, BFS SB-20). But with dynamic batching enabled with $b^{max} = 25$, we see a median latency of 5.39 $secs$ but crucially, no latency delays for events (Figs. 18a and 18b, BFS DB-25). This confirms the need for and benefits of *dynamic batching* in PRID App as well.

The tracking logic in PRID App plays an important role in managing the growth in the active camera set size, similar to ORID App. In Fig. 18c, using a static batch size of $b = 20$, we see that TL-WBFS, which uses the knowledge of road lengths for its spotlight expansion, has a more modest increase in camera count, e.g., at ≈ 500 $secs$, compared to TL-BFS. This help it scale to a denser camera deployment or a longer duration of the entity being in a blindspot. Both ORID App and PRID App have similar camera count patterns, partly modulated by the different CRs used.

Finally, for $es = 6m/s$ we see that dynamic batching is inadequate for TL-BFS as it reports a median latency of 41.95 $secs$ and 63% of events delayed (Figs. 18a and 18b, BFS DB-25). But by enabling drops, this reduces the drops to $\approx 12\%$ with other events being processed within time, with a median latency of 5.36 $secs$ (BFS DB-25 Drops). Here again, PRID App exhibits the value of *pro-active and intelligent drops* by Anveshak rather than allow delayed events to flow through and waste resources.