Coral-Pie: A Geo-Distributed Edge-compute Solution for Space-Time Vehicle Tracking

Zhuangdi Xu
xzdandy@gatech.edu
Georgia Institute of Technology

Harshil S Shah
hshah88@gatech.edu
Georgia Institute of Technology

Umakishore Ramachandran
rama@gatech.edu
Georgia Institute of Technology

Abstract

We present a distributed system architecture which is *scalable by design* for cross-camera vehicle tracking at *video ingestion time* dubbed Coral-Pie. To meet the latency bounds for timely processing of every frame at each camera, we associate dedicated low-cost computational resource for each camera, which consists of two Raspberry Pi 3B+'s and one Coral Accelerator (EdgeTpu). The end-to-end system generates and stores the tracks in a graph database for easy querying. We use the Cloud-Edge-Device continuum to appropriately place the components of the distributed system architecture. Using the timing profiles of the sub-tasks involved in the continuous processing that needs to happen on every frame in each camera, we map the elements of the processing onto the computational resource associated with each camera. Performance evaluation of the proof-of-concept system is conducted using live streams from five campus cameras. The evaluation includes microbenchmarks as well as application level studies. The controlled experiments using live cameras are augmented with a simulation-based study to show the self-healing property of the system and the system scalability.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; **Sensor networks**; **Real-time system architecture**; • **Human-centered computing** → *Ubiquitous and mobile computing*.

Keywords: geo-distributed edge architecture, large-scale camera network, multi-camera vehicle tracking

ACM Reference Format:
Zhuangdi Xu, Harshil S Shah, and Umakishore Ramachandran. 2020. Coral-Pie: A Geo-Distributed Edge-compute Solution for Space-Time Vehicle Tracking. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425686

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425686

1 Introduction

The ubiquity of cameras and other sensors in our environment coupled with advances in computer vision and machine learning has enabled several novel applications combining sensing, processing, and actuation. Often referred to as *situation awareness* applications [28], they span a variety of domains including safety (*e.g.,* surveillance [3, 35]), retail (*e.g.,* drone delivery [2, 34]), and transportation (*e.g.,* assisted/autonomous driving [21, 44]). Since many of these applications are latency sensitive and network bandwidth hungry [26] in addition to being geo-distributed, *edge/fog* computing [7, 29] has emerged as a new trend in catering to their computational needs. Low-cost processing resources such as Rasperry Pis [10], and TPUs [12] are enabling just-in-time processing of sensor streams close to their sources.
As a concrete example, consider creating a space-time track of a suspicious vehicle using camera networks in a city. Manually checking the camera streams and searching for the trajectory of a suspicious vehicle after an incident has occurred (*e.g.,* robbery) is extremely labor-intensive and error-prone due to lapses in attention [23]. With all the advances in domain expertise (such as computer vision and machine learning), and computational infrastructure (sensors and edge computing), time is ripe for building intelligent systems that replace manual labor and aid human decision-making. In the surveillance example, an intelligent system would analyze the distributed camera streams at *ingestion time* and generate the space-time tracks rather than postmortem. Such an intelligent camera network system combines distributed systems research and computer vision domain expertise.
Using *Space-Time Vehicle Tracking (STVT)* at *ingestion time* as a concrete example, this paper builds the elements of a distributed system architecture and its implementation that allows the necessary computer vision modules to be *plugged in* easing the burden on the domain experts for developing such next generation applications.
Systems exist for tracking vehicles with known signatures (*e.g.,* license plate number) [8, 31]. The limitation with such systems is that suspicious vehicles have to be registered in advance so that they can be detected and tracked in real time. There has been work on the database side for building efficient query processing systems on archived video [13, 22].

However, we are not aware of any end-to-end system that cross-correlates video streams from a network of cameras to build space-time tracks of vehicles in real time.

In this work, we propose a system called Coral-Pie for generating the space-time tracks of *all* vehicles *all* the time at *video ingestion time* from a distributed camera network. Such a system would alleviate the need to know the signature of a vehicle we wish to track ahead of time since all vehicles are being tracked all the time.

There are several challenges to building an end-to-end system as envisioned by Coral-Pie. The first challenge is timely processing of the video from a camera to detect and generate a signature for each vehicle appearing in the field of view (FOV) of a camera. Of course, such processing has to happen simultaneously on all the cameras concurrently. Further, for cross-correlation across cameras, a detected vehicle should be re-identified if it is part of a space-time track for a specific vehicle that is being constructed on the fly. This calls for a distributed processing architecture to support the camera network with appropriate *cost-effective* building blocks in terms of hardware resources for each camera, and a mapping of the application components on to these resources for camera processing. The second challenge is communicating the signatures of detected vehicles to adjacent cameras in a scalable manner that reduces the overall necessity to coordinate across cameras. The third challenge is camera topology management. When camera failures occur or new cameras are deployed, the geographical relationship between cameras (*i.e.,* the camera topology) and the coordination between cameras needs to be updated. Pausing the whole system, and manually reconfiguring the camera topology is not desirable, which calls for automating the camera topology management.

In this work, we address the above challenges to building an end-to-end camera network management system. Though we use STVT as a specific use case to ground the work, aspects of this system are generic and would be reusable for other applications that desire to use such a camera network. Specifically, we make the following contributions:

- A *scalable by design* distributed edge architecture for tracking *all* vehicles *all* the time that addresses the processing needs for each camera with *pluggable* computer vision components for detection and re-identification. The novelty of our architecture lies in the horizontal communication among edge (camera) nodes, autonomous camera topology management, and design considerations for scalable implementation for a large-scale camera network.
- A proof-of-concept implementation of the architecture, Coral-Pie, using a combination of Raspberry Pis and Coral TPU as an edge node.

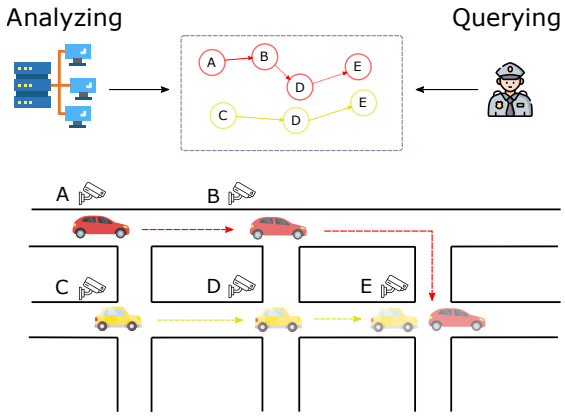


Figure 1: This figure shows an example of Space-Time Vehicle Tracking (STVT). Five cameras A, B, C, D, E are located at road intersections, continuously monitoring the vehicles' movement. We have a distributed system analyzing the camera feeds and construct the trajectory of all vehicles (e.g., the red and yellow one). Meanwhile, the authority can query the trajectory of any target vehicle. The reconstructed trajectory may contain errors. In this given example, the red vehicle did not go through camera D.

- An *in situ* evaluation of Coral-Pie using video streams from real-world roadside cameras to demonstrate the practicality of Coral-Pie.

The rest of the paper is organized as follows. Section 2 defines the problem statement and the scope of this paper. Section 3 presents the system architecture addressing the three challenges mentioned earlier. Section 4 presents the implementation details of Coral-Pie. Section 5 shows the results of evaluating Coral-Pie with a limited number of on-campus camera streams and the efficacy of Coral-Pie for scaling up to large-scale camera networks. Section 6 discusses how the Coral-Pie architecture can be generalized and reused for other smart camera applications. Section 7 discusses the related work. Section 8 presents concluding remarks and directions for future work.

2 Problem Statement

Figure 1 shows the setup for the problem being addressed in this paper. Cameras are statically and geographically distributed at the road intersections (or along the road), continually monitoring the movement of vehicles.[1] Cameras need not necessarily be "smart" by themselves; but each camera should have computational resources "nearby" to process its video stream in real-time. The computational resources can be dedicated edge devices or virtualized containers, either physically collocated with the cameras or connected to the cameras through a local area network. We assume a well-connected network (e.g., LAN for a dedicated environment such as a campus or WAN for a city-scale deployment) allowing the cameras plus their associated computational

[1]Moving cameras can be supported in Coral-Pie with the camera topology server (presented in Section 3.3), but they are outside the scope of this paper.

resources to communicate amongst themselves faster than the movement of the vehicles between any two cameras. In other words, the horizontal network resources across the cameras are not considered as constraints in this work.
In this paper, we use the following taxonomy:

- **Device**: A low-cost platform (*e.g.,* Raspberry Pi and associated camera) with limited computational capability.
- **Edge**: A multi-tenant micro-datacenter housed in a small-footprint location (*e.g.,* central offices of Telcos), expected to host up to a few server racks. Edge sites will be typically one (or few) network hop(s) away from the entities that it directly interacts with.
- **Cloud**: A multi-tenant datacenter (*e.g.,* Amazon AWS) with virtually infinite resources, and nondeterministic latency to the edge (due to WAN routing).

Given this setup, we would like to explore the design space for a smart camera system wherein the live camera streams are processed locally, and each camera is communicating and collaborating with other cameras to build up the trajectory of all the vehicles moving through the road network. We do not assume that the cameras are present at each road intersection, we do expect a dense deployment of cameras in the rest of the paper. In Section 5.5, we discuss what happens when the density of the cameras decreases.

2.1 Scope of the Paper

The focus of the paper is on solving the *system challenges* for such a distributed camera network for enabling sophisticated multi-camera vision applications. The vision components themselves are *pluggable* for the application of interest. In this work, we use well-known computer vision techniques for vehicle detection, tracking, and re-identification. We show in Section 5.6, that even with off-shelf computer vision algorithms and models, Coral-Pie can already achieve reasonable results.
The system can be queried for the space-time track of any vehicle of interest (*e.g.,* for law enforcement)[2]. Accuracy of the space-time tracks generated by Coral-Pie (*i.e., false positives* and *false negatives*) is tied to the sophistication of the computer vision techniques used. For *e.g.,* as shown in Figure 1, the system returns $A \rightarrow B \rightarrow D \rightarrow E$ as the track for the red car; but the ground truth shows that it did not go through camera D. Such ambiguities due to the vision algorithms can be easily pruned by analyzing a few frames of videos around the ambiguity either manually or by employing more sophisticated vision techniques.

3 Distributed System Architecture

In this section, we explore the design space of a scalable system architecture for a geo-distributed camera network.
One intuitive design is to stream camera feeds to the cloud for analysis, since in principle, there are virtually infinite computational capacity in the cloud. However, the backhaul network bandwidth needed to stream the video from a dense deployment of a geo-distributed camera network is infeasible [3]. Typical IP camera bandwidth requirement is between 2-24 Mbps [9], and if the camera stream is transmitted using HTTP/MJPEG protocol, the bandwidth can go up to 32Mpbs [42]. Furthermore, due to the network bandwidth pressure it becomes untenable to sustain a good frame rate from the cameras to the cloud thus affecting the quality of the camera processing results. For *e.g.,* prior art [42] has shown that only a frame rate of 3 FPS was possible at the cloud while it was possible to get 15 FPS at the edge for accessing camera streams from a campus camera network. Lastly, despite the fact that there are virtually infinite resources in the cloud, we wish to provide *guaranteed* computational resources for each camera stream toward our goal of *scalable by design.*
For these reasons, Coral-Pie uses a distributed edge architecture (Figure 2) which accomplishes four sub-goals of scalable by design: (a) eliminates the network bandwidth pressure for streaming the cameras to the cloud, (b) provides the necessary computational capacity for each camera on the device associated with it, (c) manages the mapping of the software components in each device to achieve the necessary frame rate for camera processing, and (d) uses the device-edge-cloud continuum for appropriately placing the application components of STVT.

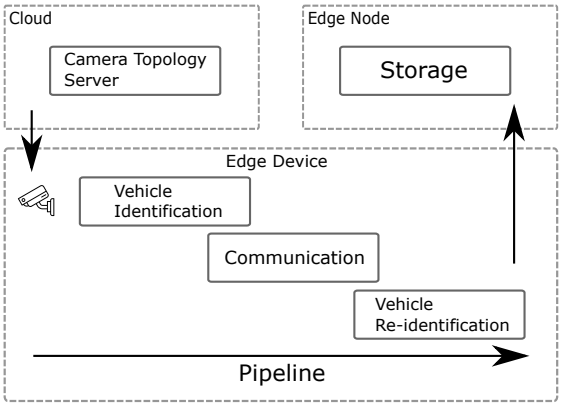


Figure 2: System Architecture: The figure shows the system architecture that spans Cloud, Edge, and Devices; and the mapping of the application components of STVT.

We summarize the functional components of STVT as depicted in Figure 2:

1. Upon a camera joining the system, it registers with the *Camera Topology Server* and gets the topology information of the local road network.
2. For each frame, *Vehicle Identification* recognizes whenever a vehicle enters and leaves the camera's FOV, and generates a *vehicle detection event* for each vehicle.
3. Then, for each vehicle detection event, *Communication* sends the event to the *downstream cameras* (determined

[2]Note that design and evaluation of the query interface is outside the scope of this paper and is part of future work (Section 8).

by the road network and the direction of vehicle's motion) to inform them that the detected vehicle is likely to appear in their respective FOV shortly. Every downstream camera receives this detection event and adds it into its candidate pool of *Vehicle Re-identification*.

4. Meanwhile, *Vehicle Re-identification* tries to match each vehicle detection event from those in its candidate pool. In other words, *Vehicle Re-identification* (if successful) confirms that a vehicle detected by an upstream camera also passed through the FOV of this camera.
5. Upon completion of processing a camera frame, *Storage* persistently stores the frame, registers and extends a vehicle's trajectory based on *Vehicle Re-identification* results.

We elaborate on the details of each of these functional components and their placement in the computational continuum to achieve our goal of scalable by design in the following subsections.

3.1 Placement of the Application Components

Steps 2-5 in the above summary happens on each captured frame from the camera in a pipelined fashion. The pipelined processing steps include vehicle identification, communication to peers, vehicle re-identification, and finally storage of the frame. Figure 2 shows a suggested mapping of the application components across the device-edge-cloud continuum. All the heavy-lifting with respect to the real-time camera processing, which includes *Vehicle Identification*, *Communication*, and *Vehicle Re-identification*, are performed on the dedicated computational resource (*i.e.,* device) associated with each camera. This arrangement ensures that we can provide deterministic latency bounds for these image processing tasks. Alternatively, we could also satiate the computational needs for these tasks on the edge node via virtualized resources so long as the latency bounds can be met. Most importantly, the camera streams need not be pushed to the cloud ensuring that there is no pressure on the backhaul network bandwidth.

Our goal is to achieve *scalable by design* both in terms of cost and performance for STVT. To this end, the cost of the "device" should just be a reasonable increment to the cost of the camera itself. Thus the device will not have sufficient storage and computational capability to support on-device storage. Therefore, a viable alternative is to offload *Storage* to a nearby edge node as shown in Figure 2, which by definition is more powerful. Since we are offloading to a local edge node, the backhaul wide-area network (WAN) is not pressured. As we will see in Section 4, this design decision also helps balance the pipeline of tasks that need to be performed on each frame.

Camera topology management maintains the geographical relationship between the cameras and gets updated when a new camera is deployed or an old camera is removed. We do not expect such churn in the topology to be frequent; further topology management is also not in the critical path of real-time camera processing. Therefore, it makes sense to have that functionality to be hosted in the cloud without hurting the scalability of the system.

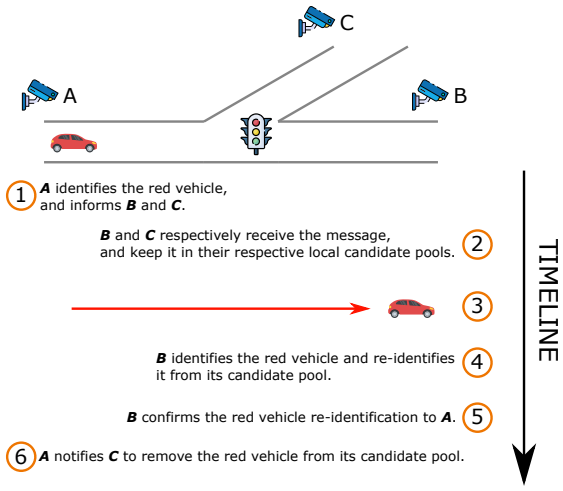


Figure 3: Communication Protocol: This figure summarizes a complete communication session when a vehicle moves between Camera A and Camera B. Upon identification by Camera A, the minimum downstream camera set (Cameras B and C) are notified; Upon re-identification by Camera B, the red vehicle is removed from the candidate pool of Camera C.

3.2 Communication Between Cameras

Optimal communication between cameras is critical for the STVT, from the perspectives of scalability, processing speed (end-to-end latency), and accuracy of the system. Flooding detection events to all cameras in a broadcast manner is no good. Such flooding not only hurts the scalability of the system but also increases the computational burden (both latency and false positives) on vehicle re-identification since the size of the search space (*i.e.,* the candidate pool) becomes larger.

Under a dense camera deployment, the topography of the road network and the direction of the vehicle's motion provide us a direct hint as to which set of potential cameras the vehicle is likely to pass through next. Take the yellow vehicle in Figure 1 as an example. When camera C saw the yellow vehicle moving to the right, it knew the yellow vehicle would enter into camera D's FOV shortly [3], so camera C only needs to inform camera D about the arrival of the yellow vehicle. The general idea holds, even if we do not have cameras deployed at every road intersection, but the number of cameras that should be informed will increase. For *e.g.,* in Figure 3, without a camera at the traffic light, camera A needs to inform both cameras B and C, since it does not know whether the red vehicle will make a left turn or not at the intersection.

[3] U-turn is not discussed in this paper. U-turn can be supported by including a given camera in its own minimum downstream camera set.

Based on the above observations, we design the following communication protocol for STVT. First, we call the set of cameras that the detected vehicle could potentially pass through *first* before it can reach other cameras in the system — *minimum downstream camera set (abbrev., MDCS)*. For example, in Figure 3, the detected red vehicle (heading towards the right) at Camera A has a minimum downstream camera set {B, C}. The proposed protocol has two stages — the *informing* stage and the *confirming* stage. At the informing stage, a camera sends a vehicle detection event (described in Section 4.1.2) to the cameras in its MDCS. Upon the vehicle arriving at one of the cameras in the MDCS set, confirming stage of the protocol begins. The camera that successfully re-identifies the vehicle will confirm the re-identification event to its predecessor. The predecessor camera will then communicate the confirmation message to all the others in its MDCS completing the confirming stage of the protocol. The confirming stage of the protocol is important for garbage collection of the candidate pool (see Section 4.1.3). Figure 3 shows the communication protocol in action when the red vehicle moves from Camera A to B.

For a given topography of the road network, the MDCS for *any* camera is a *finite small* set. By definition the MDCS for a given camera is proximal to its geo-location. Thus, regardless of the size of the camera network deployment, the communication complexity for each vehicle detection that a given camera is involved in (*i.e.,* the number of messages exchanged) is finite and small. And due to the geo-local nature of the communication protocol, there is minimal network interference for the communication that occurs simultaneously across disjoint road segments. Therefore, the communication protocol adheres to the *scalable by design* goal of Coral-Pie.

3.3 Camera Topology Management

Camera topology management deals with the deployment of the cameras on the road network and determination of the MDCS for each camera with respect to the direction of movement of vehicles that appear in its FOV. There are two important observations in this regard: (1) under a static camera topology (and in the absence of any camera failures), the MDCS for a given camera is the *same* for *all* detected vehicles that are headed in the same direction; (2) a vehicle in the FOV of a given camera has only a *finite* set of moving directions due to the road network. Based on these observations, we can use the camera topology to *preconfigure* the MDCS for all cameras and accordingly set up long-term communication channels between a given camera and its MDCS set. Such preconfiguration avoids the overhead of calculating MDCS at runtime and is yet another element of the *scalable by design* principle of Coral-Pie.

It is tedious to manually configure the MDCS for each camera. Further, cameras are also vulnerable to failures. Examples include hardware failures, power failures, network failures, cameras becoming dysfunctional due to the flakiness

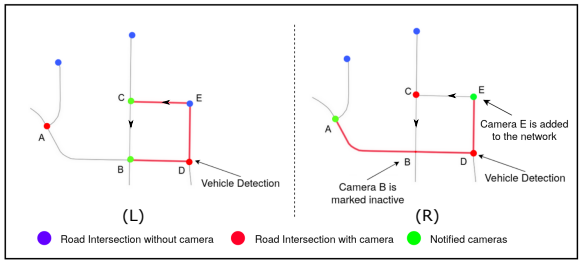


Figure 4: The camera topology is stored as a graph with road intersections as vertices and lanes as directed edges. The vertices are marked as either equipped with a camera or not. EC and CB are one-way roads while the rest are two-way. We have four cameras A, B, C, D deployed in the left camera topology. For example, doing a DFS from camera D (shown as red edges in the figure), we know its MDCS is either {B} for ← direction or {C} for ↑ direction. Later, in the right camera topology, we remove the camera B (*e.g.,* due to failures) and deploy a new camera E to the network. Similarly, doing another DFS from camera D, we get its new MDCS which is {A} for ← direction or {E} for ↑ direction.

of streams they produce, and software crashes. These failures will change the camera topology in real-time from the original deployment, and consequentially change the MDCS for the affected cameras. Fortunately, the preconfiguration of MDCS can be fully automated. In Coral-Pie, a *camera topology server* (shown in Figure 2) is responsible for automatically updating the camera topology and MDCS upon the addition of new cameras or the removal of existing dysfunctional cameras.

The camera topology server first loads the topology of the road network under the camera system as a graph and annotates the vertices (road intersections) equipped with cameras, as the example shown in Figure 4. For simplicity, discussion in this section only covers the cameras deployed at road intersections. In Section 4.3, we show how to handle cameras along the lanes. To find the MDCS for a given camera along a specified moving direction, we can do a *depth-first search (DFS)* from the camera in the graph, and the recursion branch of DFS returns whenever it visits a vertex equipped with a camera. Figure 4(L) shows the DFS from camera D and reveals its MDCS as {B} for ← direction and {C} for ↑ direction.

When a new camera joins the system, a timer associated with the camera sends a periodic "heartbeat" message to the server at a preconfigured time interval, which is an indication to the server that the camera is still active in the network. In Section 5.4, we show how the heartbeat ratio affects Coral-Pie's healing time upon camera failures. When the server receives a "heartbeat" message from a new camera, it will use the latitudinal and longitudinal information to add that camera to the existing topology. Then the server performs a DFS search on the graph to determine the MDCS for both the newly-joined camera and the existing cameras that are affected by this change in the topology. For *e.g.,* in Figure 4(R), when camera E joins the network, camera D's ↑ direction

MDCS is affected and becomes {E}. Loss of heartbeat message from a given camera will trigger the topology adjustment as well. Camera B in Figure 4(R) becomes inactive and the topology server will stop receiving heartbeat messages from it. Then the server will update the MDCS for the affected cameras, namely, {A, C, D} in a similar fashion.

4 Implementation

Section 3 presented the high-level architecture of Coral-Pie that ensures that the end-to-end system is *scalable by design* irrespective of the geographical size of the camera network. We also discussed the appropriate placement of the application components in the device-edge-cloud continuum based on the latency criticality of the component.

The continuous processing that happens on *every frame* at *each camera* has to be carefully timed and mapped on to dedicated compute resources associated with each camera to achieve an acceptable end-to-end performance. The elements of this continuous processing, shown in Figure 2, are *vehicle identification*, *vehicle re-identification*, *communication between cameras*, and *storage* of the constructed trajectory and the raw frames. The storage element of this continuous processing is *not* in the critical path of constructing the trajectory of the vehicle. On the other hand, vehicle identification and communication have to be completed before a detected vehicle appears in the FOV of the next downstream camera. Note that there could be multiple vehicles in each frame depending on the traffic conditions and each vehicle identification will result in a communication event. Fortunately, the communication time is upper bounded by the time it takes a vehicle to move between cameras. Even at 100 kmph, and distance between two cameras of 100 meters, this gives a communication latency upper bound of approx. 4 seconds, which can be met quite easily.

However, *all* the elements of this continuous processing have to happen within a tight latency budget to ensure that we can sustain a good *frame rate* and ensure that the application-level performance metrics (*false positives* and *false negatives*) are acceptable. With preliminary experiments, we have established that a minimum frame rate of 10 FPS is necessary to achieve acceptable application-level performance metrics, and is further validated by the evaluations presented in Section 5.6. This gives a latency bound of 100 ms for each sub-task that is involved in the continuous processing on each frame. In Section 4.1, we describe implementation details on how we have selected the computational resources for each device, and how we map the elements of the continuous processing on to them to stay within the latency bound for each sub-task. Section 4.2 describes the implementation of the trajectory and frame storage which is offloaded to an edge node. Finally, Section 4.3 describes the implementation of the camera topology server, which runs in the cloud.

4.1 Continuous Processing on Each Frame

The critical path of continuous processing includes Vehicle Identification, Inter-Camera Communication, and Vehicle Re-identification. Vehicle Identification (Section 4.1.2) is responsible for identifying vehicles within each camera's FOV, while Inter-Camera Communication (Section 4.1.3) and Vehicle Re-identification (Section 4.1.4) are responsible for tracking vehicles across cameras. In the rest of this subsection, we start with the selected computational resources for each camera (Section 4.1.1), then elaborate on how we map these three critical elements on them to meet the latency bound of 100 ms for each sub-task, and finally, we summarize the design space we have explored (Section 4.1.5) before arriving at this implementation decision.

4.1.1 Computational Resources for Each Camera

Vehicle Identification and Re-identification are computationally intensive vision algorithms, which are *pluggable* modules to suit the taste of the domain experts and their goals for achieving their application level performance targets. For the purposes of this work, we have implemented off-the-shelf algorithms for these elements to have an end-to-end solution for STVT. In particular, detection (which is part of vehicle identification) based on DCNN model [11] is the most compute intensive algorithm. Therefore, we dedicate a Google's EdgeTPU [12] for the detection sub-task.

In addition to the Google EdgeTPU, each camera is equipped with two Raspberry Pi 3B+'s (abbrev., *RPi*)[4]. The *RPis* are inter-connected through a local area network, and the EdgeTPU is a USB accelerator connected to one of the *RPis*. Thus the dedicated computational resources for each camera amounts to a total cost of approx. $145[5]. Raspbian-lite [19] is chosen as the operating system for each *RPi*, and the Coral-Pie's application instances running on each *RPi* is developed in Python 3.7.

4.1.2 Vehicle Identification

The goal of the vehicle identification element (shown in Figure 2) is to recognize the appearance of each vehicle within one camera and generate a unique vehicle detection event for it. Accomplishing this element involves detecting vehicles in each frame, tracking them as long as they remain in the FOV of the camera, and extracting a feature set for each vehicle. For the effectiveness of the space-time vehicle tracking, the FOV of the cameras should cover all lanes, so that there is no hidden path for vehicles to leave the camera network. Occlusions (e.g., a truck might hide a smaller car)

[4]Recently released Raspberry Pi 4 with 4GB memory and USB 3.0 support can further strengthen our work. Raspberry Pi 3 has only 1GB memory and USB 2.0 support. However, Raspberry Pi 4 was not available when we started this work.

[5]Other popular hardware for machine learning includes NVIDIA Jetson Nano [24] and Intel NCS2 [17]. However, on standard existing benchmarks [1], Google Edge TPU outperforms them both in cost and vehicle identification latency.

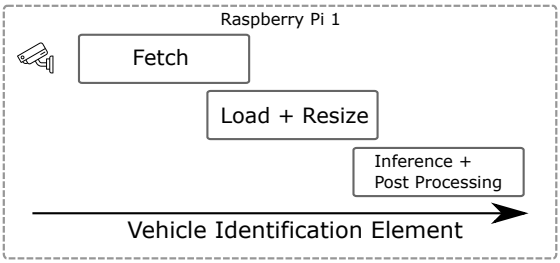


Figure 5: Vehicle Detection on *RPi* 1: Three stage-pipeline with five sub-tasks as shown.

could inevitably cause false negatives for vehicle identification. However, having multiple cameras from different angles at a given intersection would help to reduce the impact of such occlusions.

Vehicle Detection

To detect vehicles from every frame, we use a pre-trained model — MobileNetSSD V2 (COCO) [11] on EdgeTPU for inference, which takes 80-90ms irrespective of the number of vehicles in the frame.

Given that DCNN-based vehicle detection is the most computationally intensive task in the whole pipeline *RPi* 1, to which the EdgeTPU is attached, is dedicated to this task. A three-stage pipeline (each stage is an independent thread) is deployed on *RPi* 1 for vehicle detection, as shown in Figure 5: the first stage fetches the frame from the camera (roughly 60-70 ms at 15 FPS); second stage decodes the frame and resizes it (roughly 90 ms); and third stage does the inference and post-processing (roughly 90 ms).

The inference yields a list of bounding boxes which symbolizes the positions of all vehicles within the frame. Then post-processing uses a 3-step procedure to filter out "undesired" bounding boxes: 1) The label generated by the inference should be one of {car, bus and track}. 2) The "confidence" (returned by the inference) needs to be larger than the predefined *minimum confidence threshold* (e.g., our prototype system in Section 5.1 uses 0.2). 3) The centroid of the bounding box needs to be inside the *Context of Interest (CoI)* area of the camera. For each camera, we manually define a CoI area, which is usually the central area of the camera's FOV. Bounding boxes whose centroids are outside the CoI are discarded because they are usually too blurred to contain the details and would lead to false positives in STVT.

The filtered bounding boxes with the frame are then sent to *RPi* 2 for the remaining tasks of the vehicle identification element on *RPi* 2. The communication between the two *RPis* is based on the non-blocking ZeroMQ message.

Vehicle Tracking and Feature Extraction

Before a vehicle leaves a given camera's FOV, it would appear in consecutive frames and be detected multiple times. We need to identify the fact that these detections (*i.e.,* bounding boxes) are from the same vehicle and ultimately generate a *single* detection event for that vehicle by a given camera.

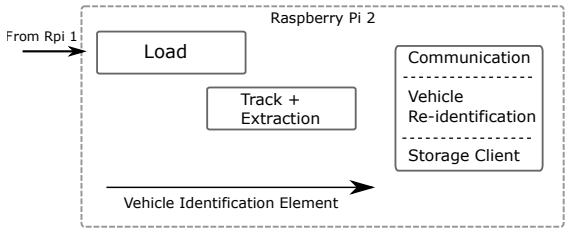


Figure 6: Tasks Mapped onto *RPi* 2: It includes vehicle tracking and feature extraction sub-tasks of vehicle identification; additionally communication between cameras, vehicle re-identification, and the storage client are also mapped onto *RPi* 2.

The function of vehicle tracking sub-task is de-duplication of detection events for the same vehicle during its journey through the FOV of a given camera.

We use *Sort Tracker* [4] to track vehicles between frames during its journey through the FOV of a given camera. We feed the bounding boxes received from *RPi* 1 into the Sort Tracker, which assigns an ID for each bounding box. A new ID implies that the bounding box belongs to a new vehicle that is entering the camera, and an old ID implies that the bounding box belongs to an old vehicle that has been detected in previous frames and fed into the Sort Tracker. In other words, each ID represents a unique vehicle within a camera's FOV.[6] A vehicle is considered leaving the camera when its ID does not appear in the output of the Sort Tracker for *max_age* (*i.e.,* a pre-defined threshold) consecutive frames. The *max_age* parameter can give us some tolerance over the false negatives of vehicle detection, and thus increases the fidelity for correctly generating a detection event for each vehicle. We choose *max_age* to be 3 in our prototype system in Section 5.1.

Upon the vehicle leaving the camera's FOV, two features are extracted from the tracklet of the vehicle (*i.e.,* the sequence of the bounding boxes).

- The direction of motion of the vehicle is estimated by drawing a line linking the centroids of bounding boxes in time order and adjusted by the camera's native videoing angle.
- An adaptive histogram [30] (*i.e., signature*) for the vehicle, which represents the color and shape of the vehicle giving more weightage for the pixels in the center of the bounding boxes for the vehicle. Color-histogram by itself is not capable of distinguishing between vehicles with similar colors. However, coupling this feature with the temporal and spatial locality of a given vehicle's movement between adjacent cameras, it is still possible to achieve a highly accurate cross-camera vehicle re-identification with color histograms [25].

Finally, a detection event is generated for the vehicle which is a JSON object that contains the name of the camera that

[6]The scope of Sort Tracker's ID is local, and it does not represent any features of the vehicle and cannot be used for cross-camera tracking.

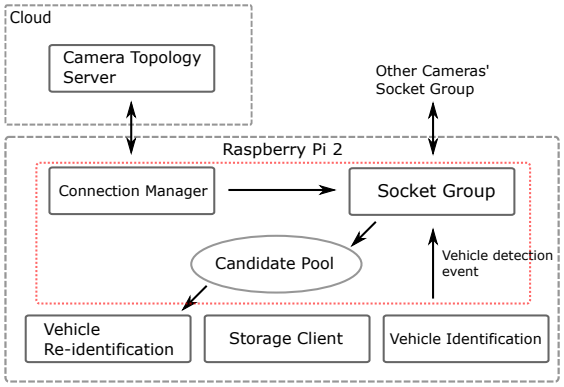


Figure 7: Inter-Camera Communication: The red rectangle zooms into the functionality of the Communication element of continuous processing in each camera.

detected the vehicle, the UTC timestamp of the detection, the features including moving direction and adaptive histogram, the ID of the vehicle generated by the Sort Tracker, and the ID of the corresponding vertex in the trajectory graph (see Section 4.2.1).

Figure 6 shows the tasks that are mapped onto *RPi* 2. There are three stages (each is an independent thread) deployed on *RPi* 2. The "Load" stage on *RPi* 2 is the same as the one on *RPi* 1. The second stage carries out the Vehicle Tracking and Feature Extraction sub-tasks of the vehicle identification element (their combined latency is less than that of "Load"). The output of the Vehicle Identification element — a stream of vehicle detection events is given as the input to the last stage of the pipeline on *RPi* 2, which completes the remaining elements of continuous processing on each frame, namely, Communication, Vehicle Re-identification, and Storage Client.

4.1.3 Inter-Camera Communication

Figure 7 zooms into the communication component on *RPi* 2, which consists of a connection manager, a socket group, and a candidate pool. Socket Group is a collection of socket communication between nearby cameras, more precisely, a hashmap between the moving direction and sockets to the cameras in the corresponding MDCS. Upon the arrival of a detection event from Vehicle Identification, based on the vehicle's moving direction, the Socket Group can directly forward it to the downstream cameras. Meanwhile, the Connection Manager sends heartbeat messages to the Camera Topology Server periodically and receives the latest MDCS information in the background. Upon a change in the topology (due to the removal or addition of nearby cameras), it re-configures the Socket Group accordingly. Socket Group also includes a listening thread to handle messages from other cameras. Upon receiving an "informing notification" (Section 3.2) from an upstream camera, the connection manager appends the associated event e_i into its candidate pool for use by the Vehicle Re-identification element. Upon receiving a "confirming notification" from a downstream camera (Section 3.2), the connection manager will send a confirming notification to all the other downstream members in its MDCS. Upon receiving a "confirming notification" from an upstream camera, the connection manager annotates the associated event e_i as "matched" in its candidate pool. All matched events are ready to be garbage collected. However, to reduce false negatives, pruning of matched events are done only when the candidate pool grows too large.

4.1.4 Vehicle Re-identification

As we mentioned earlier in Section 3, the goal Vehicle Re-identification (shown in Figure 2) is to *match* each vehicle detection event (e_t) at a given camera from those in its candidate pool (Figure 7). The candidate pool data structure contains a list of detection events, $[e_1, e_2, \ldots, e_n]$, that was received from upstream cameras. This matching is done by calculating the Bhattacharyya distance [37] between the adaptive histogram of e_t and e_i. If this distance $Bhatt(e_t, e_i)$ is smaller than the pre-configured *Bhatt_threshold*, then e_i is successfully matched with e_t. The matching results (*i.e.,* e_i and e_t) will then be given to the Storage Client (shown in Figure 7) for updating the trajectory of the vehicle. The successful match will also notify the communication manager (Figure 7) to report the match to the upstream camera as part of the communication protocol (Section 3.2).

For every matched event e_i, we annotate it as "matched" in the candidate pool, which implies that it can be safely removed from the candidate pool. Notice that it is best not to do this removal immediately. The reason is that the reported matching could be a false positive and we want to make sure that eager pruning of the candidate pool does not lead to false negatives. Therefore, the pruning of the candidate pool is done only when the pool grows too large.

4.1.5 Exploration of Design Space

We conducted extensive design space exploration before we arrived at the specific implementation described in the above subsections. We summarize some of the key design alternatives that were explored.

Computational resources per camera. Initially, we experimented with a single *RPi* for each camera. However, we quickly realized that to meet the sub-100 ms latency for each subtask, we needed to add an additional *RPi* and split the computation to be performed on each frame as detailed in the above subsections. Further, the DCNN-based detection algorithm, being the most computationally intensive, required dedicating an EdgeTPU connected to the first *RPi* just for the detection task.

Choice of vision algorithms. We experimented with several vision algorithms such as YOLO [27], Deep Sort [39], and GoogLeNet [43]. However, we found them to be computationally expensive. We also experimented with an alternative

method, namely, "detect-and-track" that is often used to reduce the computational cost of tracking a vehicle during its passage through the FOV of a camera. In this method, the detection model is run at a specific frame interval (e.g., every 5 frames), and a KCF tracker [14] is used for tracking the detected vehicle(s) on the intervening frames. We found this method to be not robust enough for vehicle identification. However, with a dedicated EdgeTPU for the detection task, we were able to run the detection algorithm on every frame, and use it in tandem with the Sort tracker as described in Section 4.1.2 to get robust vehicle identification results.

Image Serialization. The raw pixels of an image coming from the camera into *RPi* 1 have to be communicated to the other computational resources attached to each camera (*RPi* 2, and storage server). "Image serialization" is the process of converting the raw image into other formats (e.g., JPEG, NumPy [33]) before transporting it among the computational resources. We experimented with these options for image serialization but both these options turned out to be computationally expensive on *RPi*.[7] Instead, we chose to keep the image in its raw form for transportation, which incurs additional communication latency and impacts the time of the Load subtask. However, this option turned out to be much better from the point of overall latency for the pipelined processing and helped us achieve the 100 ms latency bound for the subtasks.

Mapping tasks to the *RPis*. We experimented with several different mappings of the elements of the continuous processing on the two *RPi*'s before setting on the ones detailed in the above subsections. For example, it made logical sense to map all the subtasks for Vehicle Identification onto one *RPi*.[8] However, this mapping resulted in increasing the resource contention between different elements of the pipeline mapped to a single *RPi* and broke the target latency bound.

4.2 Trajectory Storage and Frame Storage

We need a persistent store for the space-time tracks generated by the camera network as well as for the raw frames captured from the cameras (if further analyses are needed on the raw streams). Both due to the limited storage capacity and the unacceptably high computational cost for encoding/decoding the raw image frames on *RPi*, it is infeasible to host the persistent store on the compute resources associated with each camera. Instead, Coral-Pie uses a nearby Edge nodes to act as persistent stores (the Storage component shown in Figure 2). A given Edge node may serve as the persistent store for a small set of cameras in the same geographical neighborhood.

[7]For e.g., conversion to JPEG took 135 ms and conversion via the NumPy route took over 100 ms.

[8]This would involve moving the Track + Extraction in Figure 6 to Figure 5, so that all the subtasks of Vehicle Identification are on *RPi* 1.

4.2.1 Trajectory Storage

The trajectory of all vehicles is stored in one composite probabilistic graph, where vertices are detection events generated on cameras, and edges connecting vertices build up the trajectory of a given vehicle. Given the ground truth, the graph should be composed of several disjoint paths, where every path represents the trajectory of a unique vehicle, and every intermediate vertex should have exactly one incoming and one outgoing edge. In Coral-Pie, to ensure that false positives do not mask out the true positives, every vertex is allowed to have multiple incoming and outgoing edges and the weight of every edge is the confidence (*aka* Bhattacharyya distance) between two connected vertices (*aka* detection events).

The incremental construction of a vehicle's trajectory graph happens as follows: (a) Upon generating a detection event e, a new vertex v_e is inserted into the graph,whose attributes contain an index for the time interval when the vehicle appeared in the camera stream. Then the ID of v_e is added back to the JSON object of e such that v_e can be accessed from other cameras. (b) Upon finding a match between e and e_k, where e_k is an event from the camera's candidate pool, a directed edge $e_k \rightarrow e$ pointing to the newer detection e is inserted into the graph, and the weight of the edge is set to the Bhattacharyya distance between the color histogram of e and e_k. To query the trajectory of a particular vehicle, one can start at a known detection for that vehicle, *i.e.,* a known vertex in the trajectory graph, and traverse the graph using incoming and outgoing edges from that vertex. The result would be a collection of paths containing false positives, which can be further pruned by a human user or more advanced analytics in the Cloud.

In Coral-Pie, the trajectory graph server (included in the Storage in Figure 2) is implemented using JanusGraph database [18], hosted on a nearby Edge node. Each camera has a JanusGraph client (part of Storage Client in Figure 6), which sends requests to the server for vertex and edge insertions.

4.2.2 Frame Storage

Frame storage is not critical for STVT, but it is essential for a real-world smart camera system such that the user can verify and visualize the trajectory of vehicles. Keeping raw video footage may also be needed from a law enforcement perspective. As we mentioned earlier, image encoding and decoding is expensive on *RPi*. Thus after the Vehicle Identification is complete on a frame, the Storage Client (Figure 6) sends the raw video frame (using non-blocking Zero MQ message) and annotations (*i.e.,* metadata associated with the frame such as bounding boxes and tracking information) to the frame storage server designated for this camera on an edge node.

4.3 Camera Topology Server

The Camera Topology Server resides in the cloud and uses OSMnx [6] to acquire the base road map corresponding to the camera network deployment. If the camera is at a road

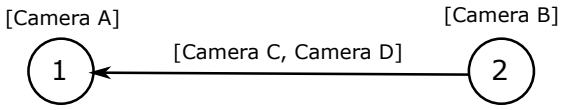


Figure 8: Cameras at non-intersections: Vertices represent road intersections. Camera A is at vertex 1. Camera B is at vertex 2. Camera C and D lie along the lane between vertex 1 and 2, and thus are assigned to the edge between these vertices. We use a list structure to indicate that Camera C is close to vertex 1, and Camera D is close to vertex 2.

intersection, the server uses the camera's latitude and longitude to locate the nearest vertex in the base map and assigns the camera to that vertex. If a camera is *not* at an intersection but along the lane, the topology server assigns it to the appropriate edge (*i.e.,* lane) using a list structure that preserves the geographical order of the cameras in that road segment. When performing a DFS mentioned in Section 3.3 to calculate the MDCS, the server checks cameras at both vertices and edges, and returns the first camera it visits in each DFS path. For *e.g.,* in Figure 8, doing a DFS from camera B returns camera D as its MDCS.

5 Performance Evaluation

We have built an end-to-end prototype of Coral-Pie as proof of concept. We have also carried out controlled experiments and evaluation of Coral-Pie using five live street cameras controlled by the campus police. The focus of the performance evaluation is to showcase the system aspects of the distributed architecture.

5.1 Prototype System

As detailed in Section 4, each camera in Coral-Pie is associated with two *RPis* and one *TPU* that serve as the dedicated computational resource per camera. The cameras that are used in the evaluation are under the control of campus police and require accessing them through an authenticated gateway server. The gateway is two network hops away on a LAN from the *RPis*, and the measured latency is 2ms. Consequently, the camera stream is fetched in a frame-by-frame manner using HTTP requests from the Avigilon server that serves as the secure gateway. The resolution of each camera frame is 1280×1024. The maximum frame rate possible from the gateway for live camera streams is ~15 FPS. We heuristically choose the parameters of the vision algorithm, namely, *minimum confidence threshold* and *max_age*. Specifically, we set the *minimum confidence to be 0.2* and *max_age to be 3*. Figure 9 shows the field of view of the 5 campus cameras and their context of interest (CoI).

The results to be presented in this section are based on controlled experiments that use frames collected simultaneously from live campus cameras. Due to the shortage of human-resources on labelling the ground truth, we limit the controlled experiments to 10000 frames from 5 cameras (i.e., 2000 frames per camera).

Figure 9: The FOV of the five campus cameras used in the evaluation study. Context of interest (CoI) shown as red polygons.

Raspberry Pi 1			
Fetch	67ms	Load	94ms
Resize	2ms	Inference	93ms
Post-Inference	1ms	RPi1_To_RPi2	1ms
Raspberry Pi 2			
Track	10ms	Feature Extraction	4ms
Communication	2ms	Vehicle-Reid	12ms
Trajectory Storage	28+30ms	Frame Storage	1ms

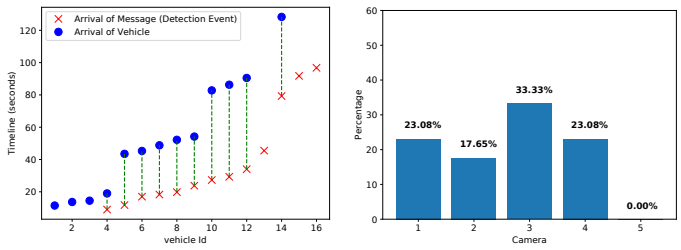
Table 1: Coral-Pie Latency Summary: Timings for all the sub-tasks that need to execute for the continuous processing of every frame on each camera. The latency number for each sub-task is the average over 2000 frames.

5.2 Microbenchmarks

Table 1 presents the measured latency for each of the sub-tasks for the continuous processing on each frame with Coral-Pie. Fetch, Load, and Inference are three costly operations on the first *RPi* which justifies our three-stage pipeline design. The high latency of Load indicates image decoding is inefficient on *RPi*. Inference latency can be further reduced by replacing Raspberry Pi 3 B+ with Raspberry Pi 4 which supports USB 3.0. Due to the nature of the pipelined architecture of Coral-Pie, the overall throughput is limited by the slowest stage in the first *RPi*, namely, Load. Still Coral-Pie is able to keep up a processing speed of 10.4 FPS with live campus camera streams, which is 5X times better than what is possible with a naive sequential execution of all the elements of the continuous processing that needs to happen on every frame.

5.3 Effectiveness of Communication Protocol

We first want to show that the proposed communication protocol in Coral-Pie meets expectations. In other words, when a vehicle arrives at a camera, the detection event sent by the upstream cameras should be ready in the camera's candidate pool for re-identification. This is illustrated in the results shown in Figure 10(a), wherein arrival of vehicles at Camera 1 is shown by blue dots and the arrival of the corresponding "informing" message of the communication protocol from the upstream cameras is shown by red markers. The informing message arrives well ahead of the vehicle arrival event at the camera. We start and stop the whole camera system simultaneously in the experiment, so the first several vehicles do not have detection events sent by the upstream cameras and the last several vehicles are still on their way to the camera's field of view. The point of this experiment is to show that the communication protocol in



(a) (b)

Figure 10: (a) Shows the vehicle arrival time and corresponding message (detection event sent by upstream cameras) arrival time on Camera 1. The green dashed line indicates the interval between two arrival times. The stepped structure is caused due to traffic lights. (b) Shows the percentage of spurious detection events in each camera's respective candidate pool.

Coral-Pie will not be the cause for any false negatives of vehicle trajectory[9].

The second experiment concerns the number of spurious "informing messages" received by each camera due to the communication protocol. Ideally, such messages go to exactly the next downstream camera to construct the space-time trajectory for a vehicle. Without an oracle to guide the communication protocol does the next best thing to use the topography and send such messages to the cameras in the MDCS. Except for the specific downstream camera that gets to re-identify the vehicle, such messages are spurious for all the others in the MDCS. Such messages result in spurious entries in the candidate pool, increasing the latency for the re-identification algorithm and also potentially causing false positives of any vehicle re-identification algorithm which may not be 100% robust. To measure the amount of such spurious entries, we first isolate the computer vision errors (*e.g.,* false positives from vehicle identification also result in redundant events in downstream camera's candidate pool) by manually labeling the ground truth from 2000 consecutive frames on each camera and accounting the "unmatched"' detection events (at the end of the experiment) in the candidate pool as "redundant". As shown in Figure 10(b), the percentage of redundant events in each camera's candidate pool is low (as a comparison broadcasting such messages to all the five cameras results in over 83% redundant events). Furthermore, most of these "redundant" events are caused by specific vehicles that have not yet reached the downstream camera's FOV at the end of the controlled experiment; therefore, they may not be redundant in the long run.

5.4 Fault Tolerance

Coral-Pie is self-healing in the presence of failures, because the topology server automatically detects failures and recreates the new camera network topology and disseminates it to the affected cameras. Upon a camera failure, the camera topology server and the communication component on each camera work together to reconfigure a camera's MDCS and communication channels on the fly. To study the healing time, *i.e.,* the time during which there could be a surge in false negatives,[10] we conduct a simulation-based study. We simulate 37 cameras deployed around the campus and kill 10 randomly chosen cameras successively to measure the time that it takes for all affected cameras to get the correct topology update. Figure 11 shows the results for two settings of the heartbeat message interval, 5 seconds and 2 seconds. Heartbeat interval determines how quickly the camera topology server can detect the camera failure. So a low heartbeat interval leads to fast failure recovery and less variance in the recovery time (the red line in Figure 11). From Figure 11, we also see that Coral-Pie takes at most twice the heartbeat interval to recover from camera failures.

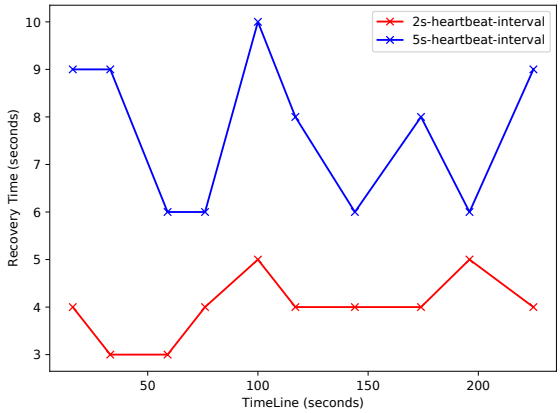


Figure 11: Recovery from Failures: x-axis is the timeline; Camera failures are marked with an "x" along the timeline. y-axis is the recovery time. The graphs show the recovery time for all affected cameras to get the topology update from the topology server.

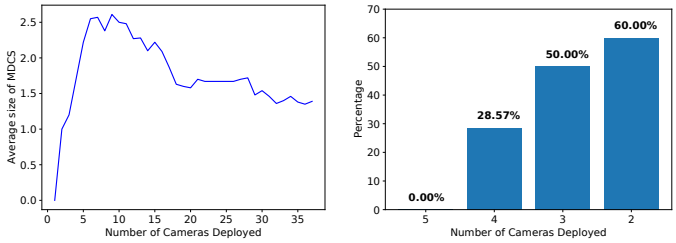
5.5 Scalability

Coral-Pie is scalable by design. Scaling up the size of the camera network would make Coral-Pie perform even better for several reasons. Consider cameras at every intersection. In this case, given that the MDCS for any camera accounts for the moving direction of a vehicle, the size of MDCS will just be 1, so there will not be any redundant entries in the candidate pools of any of the cameras.

Figure 12(a) shows the average MDCS size as a function of the size of the camera network. Firstly, we notice that no matter how many cameras are deployed, the MDCS size is always finite, which justifies our claim in Section 3.2 that the communication cost in Coral-Pie is bounded. Secondly, with an increasing number of cameras, the average size of MDCS drops, which means that each camera needs to forward the detection events to potentially fewer downstream cameras. So, increasing the number of cameras reduces the

[9] Inadequacy of vision algorithms could still lead to false negatives as we observe in Section 5.6.

[10] For e.g., when a vehicle is leaving from an upstream healthy camera of the failed camera during the healing time, the system notifies the old MDCS for the healthy camera that includes the failed camera, potentially resulting in missing a part of the space-time trajectory.



(a) (b)

Figure 12: (a) Shows the average size of MDCS for the cameras with an increasing size of camera network. This result generated through simulation, wherein we incrementally deploy 37 cameras (in random order) to the campus network and measure the size of MDCS for each camera. (b) Shows the percentage of redundant events in candidate pool of Camera 5 with a decreasing number of active cameras. This result was generated by successively deactivating Cameras 4, 3, 2 in the campus camera network which was our experimental platform consisting of 5 live cameras.

camera	recall	precision	F_2-score
1	1	0.89	0.98
2	1	0.93	0.99
3	0.95	0.71	0.89
4	1	0.85	0.97
5	1	0.83	0.96

Table 2: Event Detection Accuracy: The accuracy numbers ("1" is a perfect score) are based on the collection of 2000 frames from each campus camera simultaneously, which is roughly 2-minute of video recording for each camera. "Recall" relates to false negatives; "precision"' relates to false positives; F_2-score is a composite score that favors minimizing false negatives.

communication burden on each camera. Further, the number of redundant events in a camera's candidate pool also drops, which makes the computation on each camera more effective. So in Coral-Pie, with the camera network scaling up, the workload on each camera will decrease, which bodes well for the scalability of the system. On the other hand, when the density of the camera decreases, the size of MDCS potentially increases for each camera. In Figure 12(a), when only 10 cameras are deployed, the average MDCS size is roughly 2.5. Given that in reality, a vehicle will reach exactly one downstream camera, the additional 1.5 informing messages become redundant entries in the downstream cameras' candidate pool. To see the effect of decreasing the density of cameras in a real-world deployment, we successively deactivate Cameras 4, 3, 2 in the campus camera network. As a consequence, in Figure 12(b), we can see that the percentage of redundant entries in Camera 5' candidate pool increases from 0% to 60%. The ultimate impact of such redundant entries is a potential increase in the false positives of the vehicle re-identification algorithm that may not be 100% robust.

5.6 Application Level Evaluation

As detailed in Section 4, the proof of concept end-to-end system uses off-the-shelf models for the computer vision techniques and comparison metrics (such as MobileNetSSD V2 COCO, Sort tracker, adaptive color histogram, and Bhattacharyya distance). *False positives* and *false negatives* are useful application-level indicators for evaluating the goodness of an end-to-end system for space-time vehicle tracking. More specifically, we adopt F2-score [38] to evaluate the accuracy of the vehicle trajectory generated by Coral-Pie, which emphasizes reducing false negatives [36]. The rationale is that for surveillance video query tasks, minimizing false negatives is more crucial than false positives [5, 36]. Table 2 shows that Coral-Pie's vehicle identification pipeline achieves very high accuracy (recall, *i.e.,* false negatives; precision, *i.e.,* false positives; and F_2-score). Furthermore, four of the five cameras do not have any false negatives (a "recall" score of 1, *i.e.,* no missing vehicle detection events).

On the other hand, for Vehicle Re-identification, Coral-Pie achieves an overall 0.71 F_2-score, an indication that there are more false positives and false negatives in the generation of the space-time trajectories. This result is attributable to the off-the-shelf algorithms used for signature extraction (in Feature Extraction) and distance function (in Vehicle Re-identification), indicating that the end-to-end system could benefit from more sophisticated vision techniques from domain experts. This is where the "pluggable" nature of the end-to-end system design becomes valuable.

In spite of the overall 0.71 F_2-score, during our experiments, it was observed that vertices have at most 2 redundant outgoing edges (*i.e.,* false positives), implying that fairly minimal further effort (*e.g.,* by security personnel) is needed to further prune the trajectories.

It should be emphasized that this application level evaluation is just to show that even with off-the-shelf vision algorithms the end-to-end system we have built provides acceptable results for generating space-time trajectories for vehicles.

6 Discussion

Despite the fact that we used STVT as an exemplar driver application to develop the distributed architecture and implementation of Coral-Pie, there are several artifacts that are re-usable for other large-scale camera network applications. Firstly, the distributed architecture described in Section 3 is general and suitable for Geo-distributed camera applications, with Cloud responsible for the membership management, Device with a dedicated pipeline for real-time camera processing and Edge for stream storage. This architecture offers a good balance across cost (*i.e.,* resources for computation, network, and storage), performance, manageability, and scalability. Secondly, the combo "the communication component and the camera topology server" in Coral-Pie can be directly used by other camera applications that require horizontal coordination between cameras. Since the topology is represented as a graph, the camera topology can be extended to support more advanced topology search as needed. Thirdly,

the vehicle identification pipeline construction using a combination of detection and tracking in Section 4.1.2 is general for many camera applications that require real-time object detection and tracking on limited-resource devices. Finally, lessons from our implementation experiences (e.g, avoiding serialization overhead) on using limited capability platforms such as *RPi* for camera processing would be of value to other similar endeavors.

We recognize that privacy is a major concern for smart camera systems. However, privacy concerns are out of the scope of this paper.

7 Related Work

VDBMS (abbrev., video data base management system) such as LightDB [13], VStore [40] and Optasia [22] allow queries on archived videos. However, they are analyzing the videos at query time, which implies a longer latency to acquire the trajectory of the suspicious vehicle compared to Coral-Pie where the trajectory is created on the fly. Furthermore, it will be more challenging for VDBMS to answer the queries for multiple vehicles simultaneously. VDBMS usually requires significant computation and storage resources to facilitate query processing (since they store the same video chunk in different formats optimized for different query types [40]), which makes them inherently Cloud-based solutions. In addition, streaming a large-scale camera network to the Cloud is not practical. Finally, today's VDBMS assumes that each video is independent [13]. Without high-level support, it is not easy (if not impossible) to achieve vehicle tracking across multiple camera streams in those systems as they stand today.

More recently, video analysis frameworks have emerged (such as Noscope [20], Focus [15] and Videoedge [16]) that feature processing at ingestion time, similar to Coral-Pie. While these frameworks focus on the partition of the video analysis tasks and the placement of these tasks across Edge and Cloud, they lack the support for the persistence of the video data and results, and the coordination between cameras which is essential for tracking vehicles across cameras.

iLAND [32] proposes a middleware architecture targeting reconfiguration for real-time systems. Using a video surveillance application as a driver, iLAND demonstrates how kernels of such applications (e.g., video compression) can be delivered as a service (with different implementations) for reconfiguration. The focus of Coral-Pie, namely, horizontal coordination for a camera network and its real-time management, is not addressed by iLAND.

Though the focus of our work is on the systems side, it is important to be abreast of the strides being made in computer vision for incorporation into our end-to-end solution. In this sense, our adaptation of off-the-shelf vision algorithms is largely inspired by Tang's work [30] which won the NVIDIA AI City Challenge 2018. Their work shows very high accuracy, but some of their expensive DCNN models cannot work properly on limited resource computational platforms. STTR [41] shows the possibility of tracking all vehicles at the edge over time mathematically. However, their work does not involve any real video analytics implementation. Space-Time Vehicle Tracking [42] emulates the Edge computing platform using virtual machines from the Cloud. For every camera, they assign a 4-core, 16GB virtual machine as its computation resources, while *RPis* used in Coral-Pie have only 1GB memory and are much more affordable. On the other hand, they found the network bandwidth limitation and extra administrator cost of processing live camera streams during their Cloud emulation, which motivates our work. Kestrel [25] is a multi-camera vehicle tracking system for both static cameras and cameras on mobile devices, where mobile cameras handle the ambiguity of vehicle trajectory constructed from static cameras. However, it is not a real-time system for tracking all vehicles and it processes all the static camera streams in the Cloud, which creates a network burden for large-scale camera systems.

8 Conclusion

We present an end-to-end distributed system, Coral-Pie, for automatically generating the space-time tracks for *all* vehicles *all* the time using a geo-distributed camera network. The system architecture is *scalable by design*, and extensible via *pluggable* computer vision modules. Elements of the architecture include continuous processing of every frame on each camera for vehicle identification, vehicle re-identification, efficient inter-camera communication protocol to aid re-identification, frame storage and trajectory storage, and camera topology server for self-healing the camera network. Guided by timing of the sub-tasks in the continuous processing, we present an efficient latency conscious pipelined implementation of the continuous processing on low-cost dedicated computational resources associated with each camera. The implementation of the distributed architecture spans the cloud-edge-device continuum to accomplish a scalable and cost-effective placement of the application components. We conduct controlled experiments of the prototype system using live streams from five campus cameras and simulations to benchmark the system and showcase the key attributes of the design including efficient communication, fault tolerance, and scalability. We also show using off-the-shelf computer vision algorithms that we can get acceptable results at the application level in terms of false positives and false negatives.

Acknowledgments

This work was funded in part by NSF Awards NSF-CPS-1446801 and NSF-CNS-1909346 and a grant from Microsoft Corp. We would also like to thank our shepherd Paulo Ferreira whose insightful comments and guidance helped us to improve our paper.

References

[1] Alasdair Allan. 2019. *Benchmarking Edge Computing — Comparing Google, Intel, and NVIDIA accelerator hardware.* Retrieved May, 2019 from https://medium.com/@aallan/benchmarking-edge-computing-ce3f13942245

[2] Majed Alwateer, Seng W Loke, and Wenny Rahayu. 2018. Drone services: An investigation via prototyping and simulation. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 367–370.

[3] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-time video analytics: The killer app for edge computing. *computer* 50, 10 (2017), 58–67.

[4] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. 2016. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)*. 3464–3468. https://doi.org/10.1109/ICIP.2016.7533003

[5] David C. Blair. 1979. Information Retrieval, 2nd ed. C.J. Van Rijsbergen. London: Butterworths; 1979: 208 pp. Price: $32.50. *Journal of the American Society for Information Science* 30, 6 (1979), 374–375. https://doi.org/10.1002/asi.4630300621

[6] Geoff Boeing. 2017. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems* 65 (2017), 126–139.

[7] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*. Springer, 169–186.

[8] J-K Chang, Seungteak Ryoo, and Heuiseok Lim. 2013. Real-time vehicle tracking mechanism with license plate recognition from road images. *The Journal of Supercomputing* 65, 1 (2013), 353–364.

[9] Elvia. 2019. *IP Camera Bandwidth Calculation.* Retrieved Apr, 2019 from https://reolink.com/ip-camera-bandwidth-calculation/

[10] RASPBERRY PI FOUNDATION. 2019. *Raspberry Pi 4.* Retrieved June, 2019 from https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

[11] Google. 2019. *Models built for the Edge TPU.* Retrieved Oct, 2019 from https://coral.withgoogle.com/models/

[12] Google. 2019. *USB Accelerator.* Retrieved Oct, 2019 from https://coral.withgoogle.com/products/accelerator

[13] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2018. Lightdb: A dbms for virtual reality video. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1192–1205.

[14] João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. 2014. High-speed tracking with kernelized correlation filters. *IEEE transactions on pattern analysis and machine intelligence* 37, 3 (2014), 583–596.

[15] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. 2018. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 269–286.

[16] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 115–131.

[17] Intel. 2018. *Intel Neural Compute Stick 2 (Intel NCS2).* Retrieved Nov, 2018 from https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html

[18] JanusGraph Authors. 2019. *JanusGraph.* Retrieved Oct, 2019 from https://janusgraph.org/

[19] Dave Johnson. 2017. *Create a Lightweight Raspberry Pi System with Raspbian Lite.* Retrieved Feb, 2018 from https://thisdavej.com/create-a-lightweight-raspberry-pi-system-with-raspbian-lite/

[20] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1586–1597.

[21] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge computing for autonomous driving: Opportunities and challenges. *Proc. IEEE* 107, 8 (2019), 1697–1716.

[22] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 57–70.

[23] James C Miller, Matthew L Smith, and Michael E McCauley. 1998. *CREW FATIGUE AND PERFORMANCE ON US COAST GUARD CUTTERS.* Technical Report.

[24] NVIDIA. 2019. *JETSON NANO Bringing the Power of Modern AI to Millions of Devices.* Retrieved June, 2019 from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[25] Hang Qiu, Xiaochen Liu, Swati Rallapalli, Archith J Bency, Kevin Chan, Rahul Urgaonkar, BS Manjunath, and Ramesh Govindan. 2018. Kestrel: Video analytics for augmented multi-camera vehicle tracking. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 48–59.

[26] U. Ramachandran, H. Gupta, A. Hall, E. Saurez, and Z. Xu. 2019. Elevating the Edge to Be a Peer of the Cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 17–24.

[27] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242* (2016).

[28] Mahadev Satyanarayanan. 2017. Edge computing for situational awareness. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 1–6.

[29] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.

[30] Zheng Tang, Gaoang Wang, Hao Xiao, Aotian Zheng, and Jenq-Neng Hwang. 2018. Single-camera and inter-camera vehicle tracking and 3D speed estimation based on fusion of visual and semantic features. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 108–115.

[31] Ying-li Tian, Lisa Brown, Arun Hampapur, Max Lu, Andrew Senior, and Chiao-fe Shu. 2008. IBM smart surveillance system (S3): event based video surveillance system with an open and extensible framework. *Machine Vision and Applications* 19, 5-6 (2008), 315–327.

[32] Marisol García Valls, Iago Rodríguez López, and Laura Fernández Villar. 2012. iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *IEEE Transactions on Industrial Informatics* 9, 1 (2012), 228–236.

[33] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22.

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

[35] Jianyu Wang, Jianli Pan, and Flavio Esposito. 2017. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*. 1–6.

[36] Shibo Wang, Shusen Yang, and Cong Zhao. 2020. SurveilEdge: Real-time Video Query based on Collaborative Cloud-Edge Deep Learning.

[37] Wikipedia contributors. 2019. *Bhattacharyya distance — Wikipedia, The Free Encyclopedia.* Retrieved Oct, 2019 from https://en.wikipedia.org/w/index.php?title=Bhattacharyya_distance&oldid=918888046

[38] Wikipedia contributors. 2019. F1 score — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=F1_score&oldid=928547822. [Online; accessed 12-January-2020].

[39] Nicolai Wojke and Alex Bewley. 2018. Deep Cosine Metric Learning for Person Re-identification. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 748–756. https://doi.org/10.1109/WACV.2018.00087

[40] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[41] Zhuangdi Xu, Harshit Gupta, and Umakishore Ramachandran. 2018. Sttr: A system for tracking all vehicles all the time at the edge of the network. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM, 124–135.

[42] Zhuangdi Xu, Sayan Sinha, Shah Harshil S, and Umakishore Ramachandran. 2019. Space-Time Vehicle Tracking at the Edge of the Network. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*. ACM, 15–20.

[43] Linjie Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. 2015. A large-scale car dataset for fine-grained categorization and verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3973–3981.

[44] Quan Yuan, Haibo Zhou, Jinglin Li, Zhihan Liu, Fangchun Yang, and Xuemin Sherman Shen. 2018. Toward efficient content delivery for automated driving services: An edge computing solution. *IEEE Network* 32, 1 (2018), 80–86.