Bus on the edge: Continuous monitoring of traffic and infrastructure - year 2

Canbo Ye (https://orcid.org/0000-0002-8011-5881)
Christoph Mertz (https://orcid.org/0000-0001-7540-5211)
Mahadev Satyanarayanan (https://orcid.org//0000-0002-2187-2049)

FINAL RESEARCH REPORT - August 31, 2021

Contract # 69A3551747111

Contents

1. Introduction

For the purpose of safety and liability, transit buses nowadays usually have cameras installed to observe the environment around the buses, together with some other sensors like GPS and IMU. Such data sources are undoubtedly valuable to the ambitions of a resilient and intelligent transportation system.

For example, these live data streams can be analyzed and used for abnormal event detection, traffic modeling and infrastructure monitoring and thereby provide input for up-to-date detailed maps of roads and traffic. Up-to-date detailed maps areone essential component of autonomous driving, but they are also needed for trafficmanagement, planning, and infrastructure maintenance. Moreover, the mobility of transit buses also makes the freshly-captured visual data the favorable input for executing ad hoc search queries. Example applications include searching for a missing child or lost items along the roads as well as collecting some distilled images to construct a dataset for research given a specific target. Other use cases include trafficcounts, counting of parked cars, observations of road construction, pothole detection, detecting landslide precursors, measuring snow cover, or observing crossing of wildlife.However, it is a difficult task to process and analyze the live bus data. For one thing, it is impractical to send all these data to the cloud for analysis because of thebandwidth and storage constraints. In addition, the real-world collected data are extremely redundant despite their application value. Only a small fraction of the datahas the information we are interested in. This is exactly where edge computing can play an important role. Specifically, we will have an in-vehicle computer installed on the bus to have preliminary processing of the raw input in real time and only send theframes of interest to the nearest cloudlet server for further analysis. In this context, we develop an efficient data collection and analytics platform called **BusEdge**, the goal of which is to tap into the valuable but redundant bus data to achieve data refinement and analysis in an efficient manner.

The BusEdge platform benefits a lot from the strength of edge computing in termsof bandwidth scalability, enhanced privacy and low latency. The processing on the bus edge enables us to have early discard of the raw data according to our tasks or privacy requirements. The cloudlet server can provide high compute power with lowlatency and sufficient bandwidth due to the network proximity.

There are two key ideas embodied in the design of the BusEdge platform.

Firstly,the platform should be easily extensible to various applications. Our goal is to build up a comprehensive research platform to make use of the real-world traffic data. Encapsulated applications developed by different groups can be easily plugged in and deployed on the system seamlessly with the assistance of the modular design. Moreover, the system should be scalable to a large number of vehicle clients and cloudlets. We aim to have a fleet of buses in the future to provide a gigantic amount of live data for utilization and analysis. Such a large number of data bring tremendous value but also pose great challenges for the scalability of the system.

In terms of the typical applications on the platform, the coarse-to-fine object detection pipeline is one of the most general and useful applications. Object detectionis usually the first step for many downstream tasks in the context of intelligent transportation, such as HD map update, infrastructure monitoring and hazard detection. However, it usually requires a lot of development time and effort for a domain expert to create a specific object detector pipeline and deploy it on the edge to use live data.

First, training a good detector for an ad hoc target requires a huge amount of labeled data, which is not always available especially for rare objects that are not part of public datasets. Even for those categories contained in a public dataset, itis still not the best idea to directly apply an off-the-shelf pre-trained model to the real-world data due to the distribution bias of the input data. The context-specific knowledge in real-world scenes tends to have a great impact on the performance of object detectors, for which an extra training procedure is usually needed. In addition, sufficient programming and machine learning skills are expected for the domain expert to train a well-performing model. It also requires extra effort to deploy and manage the model on a running system to process the live data. Moreover, edge devices deployed in transportation environments usually suffer from limited computing power. Latency requirements and available bandwidth also need to be taken into account. These factors will impose more restrictions on the object detection models.

In our work, we develop an application called **Auto-Detectron** which integrates labeling, recursive learning and automatic model management to rapidly launch a specific object detection task upon the BusEdge platform. The goal here is to provide a general coarse-to-fine object detection pipeline and boost the development of relatedapplications. This application can also be directly used to execute ad hoc search queries on the live bus data.

Large parts of the content of this report have been published in a Master's thesis [84]. Research performed through the sponsorship of NSF and DARPA have been leveraged for this UTC research. The software developed has been open-sourced[1].

1.1 Contributions

The main contributions of this thesis are more explicitly stated as follows:

- We build up an efficient bus data collection and analytics platform called **BusEdge**, which takes advantage of edge computing and is easily extensible and scalable.
- We develop an application called **Auto-Detectron** upon the BusEdge platform to execute ad hoc object search on the live bus data by integrating labeling, learning and model management.
- We deploy and test our system and applications on an actual running transit bus for demonstration.

1.2 Outline

The remainder of this thesis is organized as follows.

- In Section 2, we introduce the background and prior work related to our proposed platform and applications.
- In Section 3, the system architecture and implementation of the BusEdge platform is described in detail. We test the system with an example application on an actual running bus for demonstration.
- In Section 4, we present the Auto-Detectron application upon our platform and explore approaches to improve its performance. Extensive experiments are conducted using real-world data to evaluate the performance of Auto-Detectron.
- In Section 5, we conclude this report and discuss future direction

[1] Code is available at https://github.com/CanboYe/BusEdge.

2. Background

2.1 Data Collection and Analytics on Vehicles

Autonomous vehicles and intelligent transportation system have received broad interest and investment over the last decade from both academia and industry. The tremendous enthusiasm in this field has also brought about the upgrading of in-vehicle sensors such as better cameras and even LiDARs, which provide opportunities for many promising applications. The sensor data from vehicles are valuable but hardto be efficiently used due to its gigantic size and the resource limits of the on-board devices.

A few companies offer commercial products that are relevant to the research of this field. Automotive companies like Tesla [12] use the in-vehicle sensors and videoanalytics to achieve vehicle autonomy. Tesla has the inference ASIC (application- specific integrated circuit) that they call the Full Self Driving (FSD) chip [13] installed on their in-vehicle computer to execute the real-time video analytics. Their per-camera networks will analyze the raw images to perform semantic segmentation, object detection and monocular depth estimation. Technology companies like Mobileye [8] use the crowd-sourcing data from vehicles to achieve low-cost update of a high- definition map. They classify the relevant data on board and then send them to the cloud for further aggregation and semantic identification. A global bank of road information will sent to their production vehicles for autonomous applications. Another relevant company is Roadbotics [10]. They use cell phone cameras to capture video of roads and then uploads the data to a server where machine learning algorithms are performed to assess the road conditions.

[43] has explored a system which uses vehicles to collect and store sensor data, and prioritizes the dissemination of data to a local server. However, this system is limitedto selective transmission due to the lack of on-board compute. In prior work [18], wehave prototyped a system also using a camera along with an in-vehicle computer to observe road conditions with computer vision algorithms. This work provides a proof-of-concept implementation, which demonstrates that the preliminary on-vehicle videoprocessing can save considerable bandwidth with little loss of detection accuracy. Wealso reimplement this application on the BusEdge platform as an example application.While that work focuses more on the development of a specific application for conceptdemonstration, in this work we aim to create an easily extensible and scalable platform actually running on a transit bus, on which various applications can be deployed and tested using the live bus data. We also attempt to develop a more general object detection pipeline on the platform to boost the development of similar applications.

2.2 Object Detection

As the typical application on our BusEdge platform, training well-performing object detectors for different devices, in real-world settings, is one of our key contributions. The coarse-to-fine detection pipeline on our platform leads to the need for both efficient detectors and computationally intensive deep models. For the ambition of Auto-Detectron, training a model with only a handful of real-world data is closelyrelated to the research on few-shot learning. Recursive model update also has some requirements in common with research on incremental leaning. We will introducesome related research background in these fields in the following sections.

2.2.1 State of the Art

Object detection is a well-studied problem in computer vision. In early years, most of the object detection algorithms were built based on handcrafted features [22, 27] with sliding window classification. With the rise of deep learning [47], CNN-based approaches have become the dominant object detection solution and represented the state of the art. In the deep learning era, object detection can be further divided intotwo genres: two-stage detection and one-stage detection.

The first line of work, pioneered by R-CNN[32], follows a coarse-to-fine detection process which will first generate class-agnostic region proposals of the potential objects and then refine and classify them into different categories. R-CNN uses selective search [71] to propose regions, extract features using AlexNet [47] and use a Support Vector Machine (SVM) [19] to do the classification. To improve the processing speed of the model, SPPNet [37] uses spatial pyramid pooling (SPP) layer to achieve shared computation. Fast R-CNN [31] integrates the advantages of R-CNN and SPPNet and enables the simultaneous training of the detector and bounding box regressor. Subsequently, Faster R-CNN [61] finally gets rid of selective search by theintroduction of the Region Proposal Network (RPN), making it the first end-to-end and near-realtime deep learning detector. Afterwards, a variety of improvements based on Faster R-CNN have been proposed including Feature Pyramid Network (FPN) [49] and R-FCN [20]. However, Faster R-CNN with a well-performing backbone like ResNet-101 [38] can still be considered as the state of the art for object detection.Likewise, we will use Faster R-CNN as the baseline detection model deployed on thecloudlet for our applications.

One-stage detection, on the other hand, pays more attention to the speed and simplicity of the model by integrating the object localization and classification to be a

single regression problem. YOLO [59] and SSD [52] are two typical frameworks for single-stage detection following this idea. RetinaNet [50] tries to improve the accuracy of one-stage models by introducing a new loss function named “focal loss”.They claim that the focal loss can lead the detector to put more focus on hard and misclassified examples during training. [83] introduces a keypoint-based model called CenterNet, which models an object as a single point and gets rid of the anchor generation commonly used in previous works. In spite of the high speed and simplicityof these one-stage model, they are still outperformed by RPN-based methods with state-of-the-art results [83].

2.2.2 Efficient Neural Networks

The object detection architectures mentioned above can have backbones of different size. Larger backbones give more accurate detection but require more computation, and vice versa. The trade-offs in accuracy, computation and memory footprint have been analyzed in [14, 15, 42, 68] in detail.

To make use of the deep learning-based object detector on mobile devices, compact backbones are indispensable considering the limited hardware resource. Early workin this field usually attempts to compress existing neural network architectures by pruning [35, 82]. Another popular method is to explore new cost-friendly operations. This includes the depthwise convolutions in MobileNetV1 [39] and inverted residual blocks in MobileNetV2 [63]. Many of these efficient models are good candidates for the backbone model in detection network when high efficiency is required.

Given the success of these manually designed compact architectures, they have been superseded by automatically-searched counterparts [21]. Neural Architecture Search (NAS) automates the network design for state-of-the-art performance. Differentiable Neural Architecture Search (DNAS) [51, 73, 80] is one of the most common techniques used in this field. It uses gradient-based methods to optimize the architectures of the convolutional neural network. In [21], the authors jointly search both architectures and training recipes on ImageNet to acquire a performant lightweight model called FBNetV3.

In the context of our BusEdge platform, compute efficiency is one of the primary requirements of the detector deployed on the bus. A one-stage detector with a lightweight backbone like SSD-MobileNet is a good option to achieve satisfactory real time performance on the bus. Meanwhile, we will also consider combining the two-stage detector with the latest compact model like FBNetV3 when developing the Auto-Detectron pipeline.

2.2.3 Few-Shot Learning

Until now, few-shot learning mostly focuses on the classification tasks, which can be divided into three categories: (a) Meta-learning-based methods attempt to usethe task-level prior knowledge to constrain the learning process to generalize to new tasks. [28, 53, 62] aim to obtain a good parameter initialization that generalizes well for new tasks with few examples. [30, 75] generate task-aware feature embeddings to cope with few-shot novel images. (b) Another line of works focuses on metric-learning. [46, 54, 60, 67, 72] try to find the most similar class of the novel target using different well-designed distance metrics. Among these, the cosine similarity-based classifier [30] outperforms many other methods due to its ability to reduce the intra-class variance. (c) Learning a task-aware generative model from base classes is another approach which uses the prior knowledge of data. [36, 77] propose methods to hallucinate new examples to augment the novel set. [77] introduce an end-to-end framework which jointly optimizes a generator and a meta-classifier. However, these approaches haveonly been used for the few-shot classification task but not for more challenging taskslike object detection.

There are some early attempts to make use of few-shot learning methods for the object detection task. [78] adopts a meta-learning model to disentangle the learning of category-agnostic and category-specific parameters. [44, 56, 81] aim to learn re-weighting vectors from the support images and apply them to either a single-stage [44, 56] (YOLOv2, CenterNet) or a two-stage object detector [81] (Faster-RCNN).Inspired by the siamese network, [26] tries to solve the problem in a matching scheme.Some fine-tuning-based approaches are explored and evaluated in [70, 76]. Ac- cording to the experimental results in [76], however, those well-designed few-shot object detectors are actually still outperformed by simple fine-tuning methods. The fine-tuning techniques under few-shot settings are further explored in [70]. They pro-pose several tricks during few-shot training and also introduce a contrastive proposalloss to help the classification, which is inspired by the work of supervised contrastivelearning [45]. In our work, we will follow the fine-tuning-based methods when training our detectors using the deficient bus data for a given target. The application of the other few-shot algorithms is a good future direction to boost the performance of our bootstrapping model.

2.2.4 Incremental Learning

The problem of incremental learning has a long history in machine learning. First of all, many of these works [48, 57, 66] focus on adapting the original model to additionally detect objects of new classes, which is inconsistent with our goal. In addition, there are some works [16, 24] trying to continuously update the training set

and retrain the model. They aim to continuously expand the training set with data acquired from the Internet and then use all the collected data to retrain the classifier in a semi-supervised manner. However, these works rely heavily on the abundant, diverse and preliminary sorted data from the Internet. In our case, the incremental learning is conducted under the few-shot settings for one specific object class.

2.3 Edge Computing

Edge computing is a new computing paradigm that has received more and more focus in both academia and industry over the past few years. The key idea of this nascent paradigm is to place considerable compute and storage resources at the edgeof the Internet, in close proximity to mobile devices, sensors and end users. As a complementary approach to cloud computing, edge computing is characterized by its potential to improve latency, scalability and bandwidth over the cloud-only model. Terms including “cloudlets”, “micro data centers” and “fog” have been used in the literature to refer to these small data centers. We will mainly use “cloudlets” to represent the edge-located node we use for edge computing in the rest of the thesis. The modern computing landscape with edge computing is first introduced in [64] using a three-tiered model, shown in Figure 2.1. The tiered model classifies different devices based on their available resources. Figure 2.1 shows from left to right a hierarchy of increasing physical size, compute power, energy usage and elasticity [74]. Tier-1 represents cloud servers, which provide nearly unlimited compute and storage resources with high elasticity. Tier-2 represents the smaller but still powerful data center at the edge. It provides high compute power with low latency due to the network proximity to the end devices. Tier-3 includes various IoT and mobile devices, which are constrained by their physical size, weight, heat dissipation and power consumption. Sensing and local storage are the key functionalities of Tier-3 devices.

The large-scale deployment of Tier-3 devices produces a huge amount of data which exceed the processing capabilities of these mobile devices. The emergence of deep learning over the past decade further increases the compute demand for the analysis of visual data. Considering the sufficient compute and storage in Tier-1, offloading computation to the cloud servers is the most popular solution nowadays.

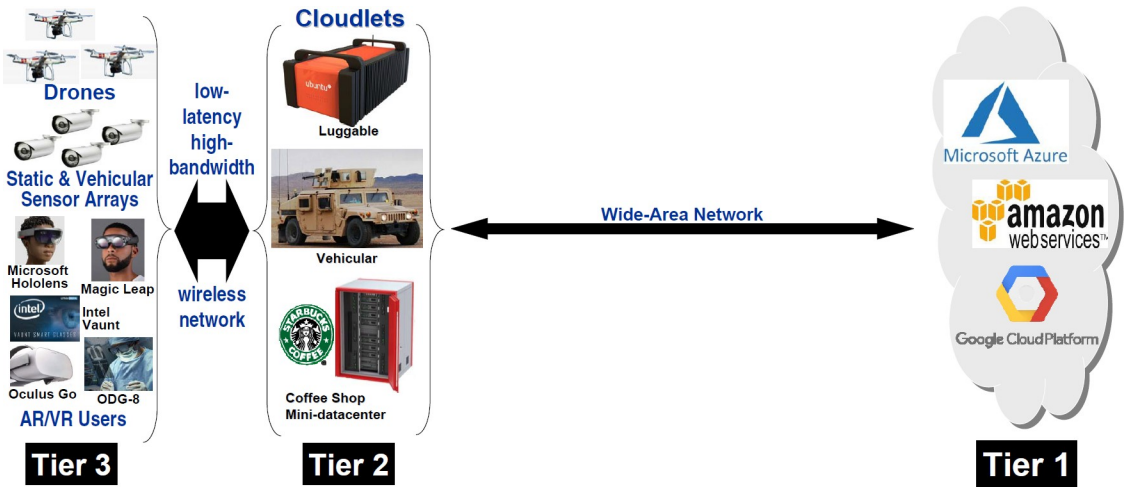


Figure 2.1: Tiered Model of Computing [74].

Several prominent cloud computing services include Amazon Web Services [1], Google Cloud [3] and Microsoft Azure [2].

However, offloading computation to the cloud has its own shortcomings. More often than not, these heavily consolidated data centers are far from Tier-3 devicesin terms of the logical network distance. This imposes significant constrains onthe applications because of the latency, throughput and network cost of offloading computation to the cloud. To solve these problems, edge computing is introducedto bring the data center closer to the edge. This provides the mobile devices withlow latency response as well as sufficient compute power. The strengths of using edge computing include low latency, bandwidth scalability, enhanced privacy, and improved resiliency to WAN network failures [65], which are important to many applications at the edge.

In the context of our system, we will have an in-vehicle computer installed onthe bus to collect and preprocess the raw data from various sensors, which can be considered as a Tier-3 device. Meanwhile, we will use a small data center situated onthe Carnegie Mellon University (CMU) campus as the cloudlet to carry out heavy video analytics tasks for the applications.

2.4 Gabriel Platform

The Gabriel platform, shown in Figure 2.2, is an application framework designed for wearable cognitive assistance using cloudlets in [17, 33]. It consists of a front- end running on Tier-3 devices and a back-end running on cloudlets. The major functionality

of the Gabriel front-end is to collect and preprocess the raw sensor data and then stream them over the wireless network to the nearest cloudlet. The Gabriel back-end on the cloudlet will distribute the incoming sensor streams to multiple *cognitive modules* to execute different tasks for the applications. The *control module*aims to manage the communication between mobile devices and cloudlets as well as the data distribution among cognitive modules. The output of the cognitive modules will be integrated by a task-specific user *guidance module* and provided to the end-users.

Even though the Gabriel platform is initial designed for wearable cognitive devices only, its design philosophy and flow control mechanism are applicable to most applications which use cloudlets and require low latency. Therefore, we will apply the flow control mechanism of Gabriel and extend it for our project.

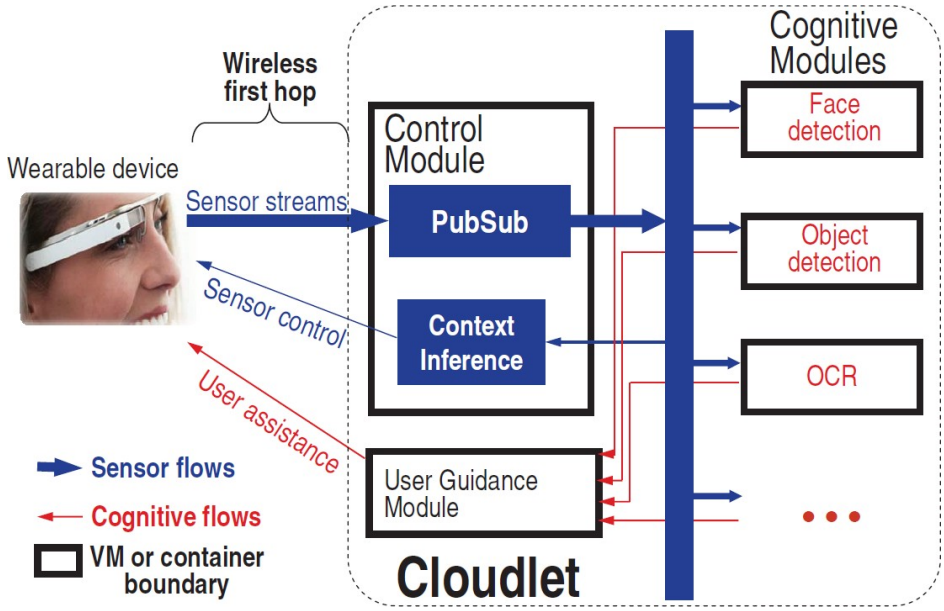


Figure 2.2: Gabriel Platform [17].

3. BusEdge Platform

This section gives a detailed description of the BusEdge platform. In Section 3.1, an overview of the system architecture is given. This is followed by the implementation details in Section 3.2. The descriptions of the hardware and the prototype platform are given in Section 3.3. Finally, we will show an example application built upon the BusEdge platform in Section 3.4 for demonstration.

3.1 System Architecture

When designing the architecture of the BusEdge platform, there are several requirements that need to be considered. The first is the scalability of the system, which means that it should be straightforward to scale the platform to perform with multiple buses and cloudlets. The second point is the extensible modular design, which aims to create a plug-and-play comprehensive platform for different applications to use the live traffic data. The third is the importance of low latency, which represents the ability to handle the most up-to-date data so that we can be responsive to different events. Moreover, the ability to cache data locally is also of great value, since it enables us to reexamine the discard piles for experiments in an offline manner.

Now let's look at the architecture of the BusEdge platform. As is shown in Figure 3.1, the BusEdge platform contains four major components: *Sensors* and *Early-Discard Filters* on the bus side as well as *Cognitive Engines* and *Sinks* on the cloudlet side.

Sensors are the data sources on the bus, which usually include video streams from multiple cameras, GPS, accelerometer and other vehicle status read from the CAN- BUS. These sensor inputs will be firstly preprocessed and then published to differentEarly-Discard Filters for data distillation.

Early-Discard Filters represent the data refinement components running on theinvehicle computer. Different tasks could share the same filtering node or have different ones. Each filter is independent and will send its outputs to the specific Cognitive Engine on the cloudlet for further analysis.

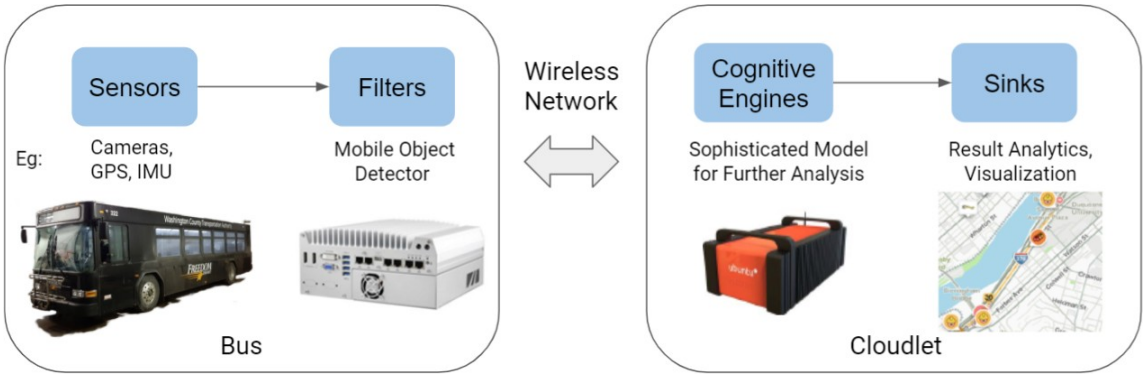


Figure 3.1: Major Components of the BusEdge Platform.

Cognitive Engines are the computer vision modules running on the cloudlet to handle and analyze the distilled data from bus clients. Each Cognitive Engine will register one type of filtered source with a given source name before launching. The flow control between the bus clients and the cloudlet server is managed by Gabriel [33], which will be introduced in Section 3.2.2.

Sinks represent the final components on the cloudlet to collect all the results from different Cognitive Engines and do the result analytics and visualization. This could be an OpenStreetMap [34] server to simply show all the detection results on a map, animage annotation tool like CVAT [4] or a toolbox for data analytics and visualizationlike Tableau [11] and Kibana [7].

The abstractions of these four components indicate how the BusEdge platform is separated. In general, the local storage and preliminary processing of the raw data take place on the bus, while the further data analytics and user interaction happen on the cloudlet. The role of the bus client is to provide high-quality filtered data in a bandwidth-efficient manner, while the cloudlet performs sophisticated data analytics and computations.

Moreover, the communication between the bus client and cloudlet server is bidirectional, making it possible for the bus clients to react to the analysis results or ad hoc requests from the server. This also enables the update of the early-discard filters on the fly. As is indicated in prior work [41], some typical vehicle-cloudlet interactions are illustrated in Figure 3.2.

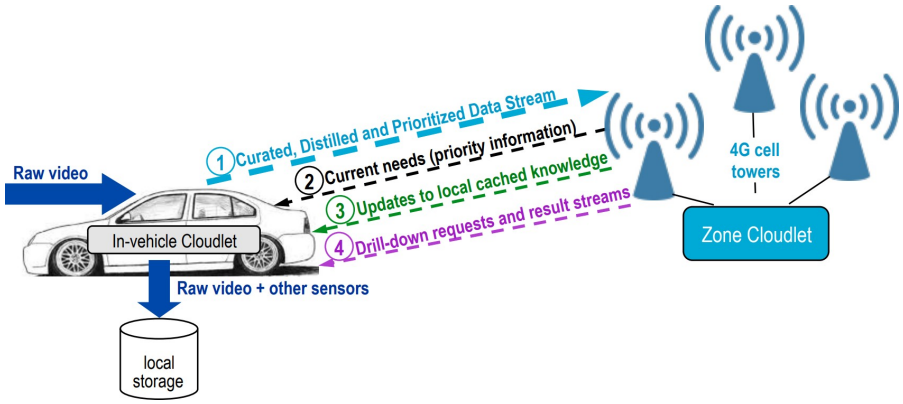


Figure 3.2: Vehicle-Cloudlet Interactions [41].

3.2 Implementation

3.2.1 Pipeline

A typical BusEdge pipeline consists of all the four components mentioned above, but the details can vary from task to task. In general, a BusEdge pipeline can use multiple sensors as input, and have one or more cognitive engines and sinks to analyze the refined data, but should only have one early-discard filter with a given "source name" to produce the filtered data unit. In other words, the source name of the filter is the keyword to define a BusEdge pipeline.

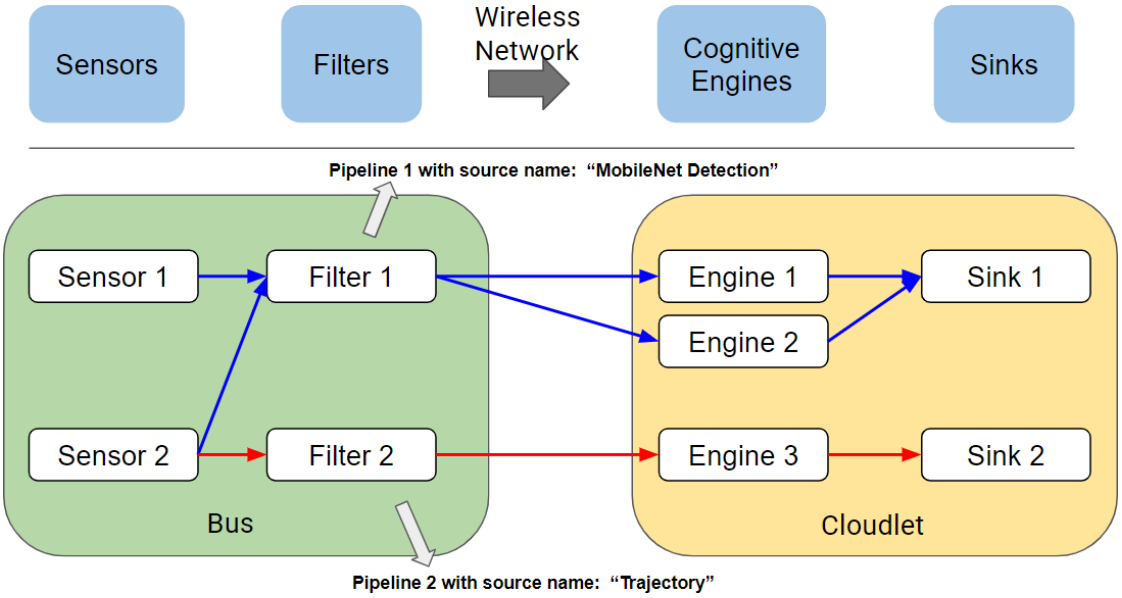


Figure 3.3: An Example of the BusEdge Pipelines.

For one thing, this pipeline design enables usto reuse filtered data to reduce the compute and network demand on the bus, which is of great importance to the scalability of the bus client. The early-discard filter usually carries out the data refinement in a general and coarse manner, the output of which can be used by various applications. For another thing, this is required by the flow control mechanism of Gabriel, which will be introduced in Section 3.2.2. This reduces the complexity of cognitive engines and also allows cognitive engines to knowwhat to expect in input frames and what results the client expects back.

Figure 3.3 shows a use case where we implement two pipelines on the BusEdge platform, which are distinguished by the color of the arrows. Each pipeline should register its own source name corresponding to a specific early-discard filter. For each early-discard filter, however, data could be captured by different sensors and itsoutput could be transmitted to one or multiple cognitive engines on the cloudlet. As is shown in Figure 3.3, pipeline 1 with the source name "MobileNet Detection" uses inputs from two sensors (e.g. camera and GPS) and sends filtered messages to two different cognitive engines on the cloudlet. For example, we can have a general trafficsign detector as the early-discard filter and have multiple cognitive engines to processits output for different tasks. On the other hand, pipeline 2 with the source name "Trajectory" has only one sensor input (e.g. GPS) and one cognitive engine, which isactually constantly sending the GPS location of the bus for live trajectory tracking on the map.

3.2.2 Flow Control

The flow control of our system is managed by Gabriel. Gabriel is an application platform initially designed for wearable cognitive assistance using cloudlets in [6, 17, 33]. Two key ideas were embodied in the design of Gabriel: (a) the ability to use disparate legacy code bases to speed up the development of applications and (b) an end-to-end flow control mechanism for timely delivery of requests even during network congestion. These two ideas are exactly in line with the needs of the BusEdge platform, making it a reasonable choice to apply the design of Gabriel to our project. We will explain the flow control mechanism of Gabriel briefly next and also illustrate its role in the BusEdge platform.

As is shown in Figure 3.4, there is a Gabriel client on the bus side to gather refined data from multiple early-discard filters, and a Gabriel server on the cloudlet side to distribute messages to different cognitive engines. The wireless communication

between the Gabriel client and server is achieved via the WebSocket Protocol. Gabriel will manage the flow control for every individual pipeline. The fundamental target of Gabriel flow control is to avoid starvation and saturation for each pipeline and enable the engines to process the most up-to-date data.

Specifically, Gabriel's flow control is based on the token mechanism. When a Gabriel client sends a message to the server, this consumes a token for the source that produced this message. When the first cognitive engine finishes processing this message, the client gets back the token that was consumed. After a client consumes all of its tokens for a source, it will only send a new message after it receives a token back from the server [6]. This ensures that the offered load of the clients will not exceed the maximum processing capacity of the cloudlet server and the cognitive engines can always process the newest data. This will help to avoid pipeline saturation and guarantee low latency.

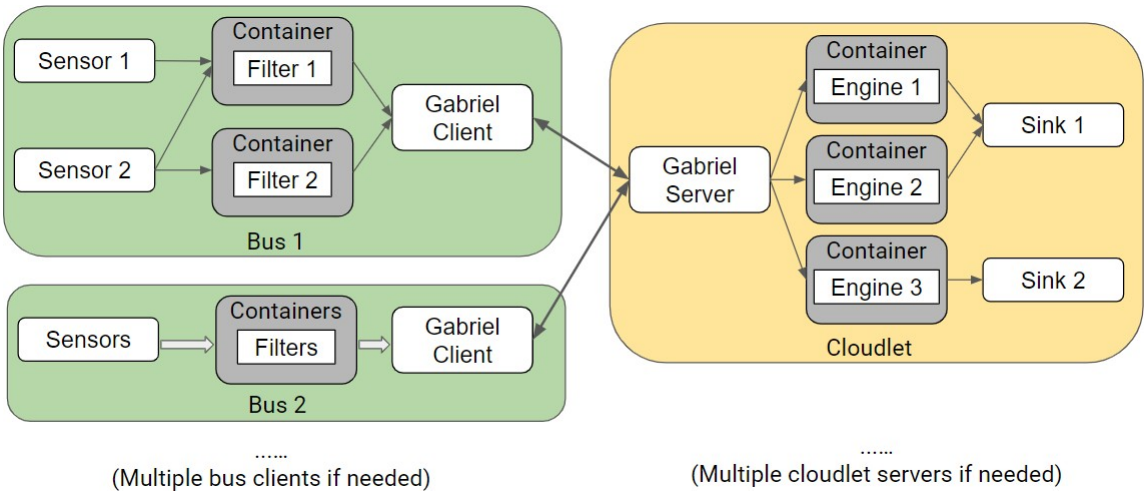


Figure 3.4: System Architecture of BusEdge.

3.2.3 Scalability and Extensibility

Modular design is another important feature of the BusEdge platform to realize extensibility. The flow control of the pipeline is transparent to the developers and they only need to develop their own early-discard filter and cognitive engine. Each early-discard filter and cognitive engine run in its own Docker container both on the bus client and the cloudlet server. The developer can also choose an existing source of early-discard filter if they share the same distillation requirements with some othertasks. In general,

when the developer wants to deploy his codes onto the BusEdge platform, all he should do is to connect his Docker container with the Gabriel server on the cloudlet and register an existing source or create a new early-discard filteron the bus. We isolate the functionality modules including cognitive engines and early-discard filters into different Docker containers because we want to enable rapidfunctionality expansion and avoid dependency conflicts. Figure 3.4 illustrates the system architecture details of a typical BusEdge use case. We can observe that all of the early-discard filters and cognitive engines are encapsulated into their own Dockercontainers.

Additionally, it can be observed from Figure 3.4 that the system can be expanded to multiple bus clients and cloudlet servers. The ability to handle multitenancy is indispensable to scale our system to a fleet of buses in the future. The challenge of multitenancy mainly lies in how to reduce the clients' offered load to the wireless network and how to allocate and schedule the cloudlet resources to minimize queueingand impacts of overload. Our solutions are to filter out futile frames, reuse the filtered source in each client and also set a threshold of task load for both the bus clients and cloudlet servers. In addition, the token-based flow control mechanism hasenabled each pipeline from multiple clients to transmit data in a way that the cloudletengines can process the most up-to-date data. However, we also admit that this strategy requires extra attention from the developer when deploying applications. Oneimprovement in our future work will be to apply the application-aware adaptation techniques proposed in the Scalable Gabriel [74].

3.2.4 Bus Client based on ROS

The implementation details of the bus client are shown in Figure 3.5. We use the Robot Operating System (ROS) [69] to manage the data acquisition, distributionand storage. The advantages of using ROS are that it provides us with a series of convenient data preprocessing packages and its publish-subscribe-based messaging protocol fits our modular architecture very well. In addition, it will be very convenientfor a developer in the robotics community to expand or update the system usingROS because of its popularity and wide application. Another detail we can see in Figure 3.5 is that not all the early-discard filters are encapsulated into the Docker containers. This is because some simple filter nodes are provided for common usages or experiments, such as sending the GPS trajectory at a given rate or sendingpreprocessed images at a fixed interval, which are not necessary to be isolated into containers.

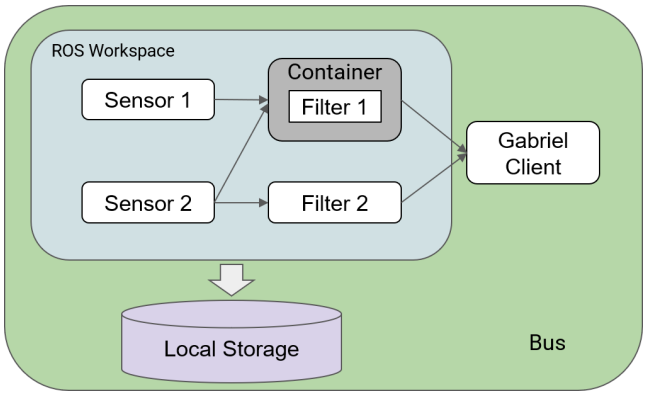


Figure 3.5: Architecture Details of the Bus Client.

It is also worth noting that in addition to real-time processing, we will also store all recent sensor data locally because these raw data could be valuable to many tasks for reexamining. The sensor inputs are recorded in the ROS-bag format. We could download this locally cached data when the bus has returned to the garage and is connected to WIFI. Using ROS-bag to store the raw data has several advantages. Firstly, it generates a compact file containing data from all the sensors tagged with timestamps. With multiple cameras and other sensors like GPS and IMU on the bus, it is usually cumbersome to sort and synchronize all the data, but ROS-bag providesa compact and flexible solution. Moreover, ROS-bag brings great convenience to the offline experiments because it enables us to play back the data according to the collected timestamps as if in real time.

Figure 3.6 illustrates the arrangement of the ROS workspace on bus clients in more depth. You can observe that more flexibility is provided inside the ROS workspace. Besides the raw data streams, we will also provide a few useful preprocessed input sources for the filters. All of these input sources are published with ROS topics andcan be used the same way as the raw data streams. This preprocessing step can greatly reduce the amount of computation we need on the resource-limited bus device. The *Preprocessing* node includes several common filtering and image processing operations. It will firstly synchronize different sensors' data with timestamps and then carry out deduplication and conditional filtering on the raw data. The cameras on the bus will generate a lot of duplicate images when it is stopped at bus stops or crossroads, which does not bring much new information. Thus, we will compare consecutive frames and remove repeated images. Additionally, some low-quality images resulting from motion blur, poor weather or illumination conditions should also be removed. This is inspired by the discussion of reducing the offloading workload in [40].

The Laplacian of Gaussian (LoG) filter is used here to detect the blurred images and image subtraction is applied to filter out those nearly identical images.Besides, we also support common filtering based on the frame rate, captured time and location such that the user can focus on only the data of interest. Here is an example to show the effect of the *Preprocessing* module: Given ROS-bags (5 frames per second) of a route of with 90627 image frames in total, there are only 7007 framesleft after we remove the duplicate and blurry images and reduce the frame rate to 1 Hz.

The *Feature Extractor* node will feed the preprocessed images to a pretrained Convolutional Neural Network and only publish the extracted feature vectors to the following filters for various downstream tasks. Popular compact models like MobileNet [39, 63] and FBNetV3 [21] are available options. The motivation is that the feature representation learned from a large training set is generalized enough to be directly transferred to novel tasks, as is indicated in [76]. For any tasks requiring a CNN backbone on the bus client, such extracted features can be shared and used. This module can greatly reduce the compute demand and benefit the extensibility of the bus client.

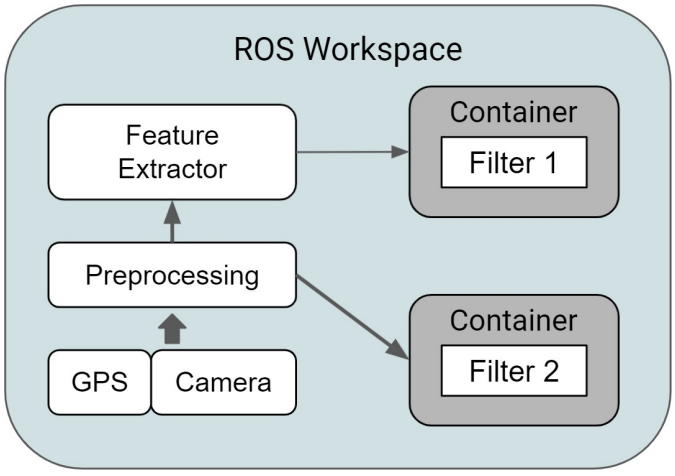


Figure 3.6: Implementation inside the ROS Workspace.

3.3 Hardware and Prototype Platform

In this section, we will illustrate the information of the hardware we use in our project, which includes the sensors and devices installed on the bus. We will also describe the transit bus on which we installed all the devices and deployed our system for on-going testing. Figure 3.7 shows some pictures of the hardware installed on the bus, the specifications of which are introduced as follows.

Figure 3.7: Pictures of the Hardware. (From left to right: bus computer; exterior camera; interior camera; GPS and network antenna.)

Bus Computer is the major computing and storage device on the bus to acquire data from multiple sensors and carry out the preliminary data analytics. All computation is performed on the CPU. The technical specifications of the bus edge computer arelisted in Table 3.1.

Brand	Safety Vision
Model	RoadRecorder 9000
CPU	Intel Core i7-8700t @ 2.40GHz
RAM	16 GB
Storage	5 TB
Power Input	9-48V DC
Dimensions	289 × 118 × 250 mm

Table 3.1: Technical Specifications of the Bus Computer.

Cameras. We install four waterproof exterior cameras at the four corners of the bus and one interior camera behind the windshield of the bus. The technical specificationsof the cameras are listed in the Table 3.2 and Table 3.3

GPS and Network Antenna. We use Mobile Mark's LTM501 Series Multiband MIMO (multiple-input-multiple-output) antenna, which contains five separate anten-nas, all in one compact antenna housing. The five antennas include two LTM/Cellular antennas, two dual-band WiFi antennas, and one GPS antenna.

Photos of the transit bus and the corresponding test platform are shown in Figure 3.8. You can find the exterior cameras at the four top corners of the bus, with two front cameras looking backwards and the other two rear cameras looking forwards.

Brand	Safety Vision
Model	37 series IP camera
Image Sensor	1/2.8” CMOS
Highest Resolution	1920 × 1080
Focal Length	2.8mm, 4.0mm
Field of View	H: 84.0°, V:43.3°, D:99.4°
Video Compression	H.264, Motion JPEG
Infrared Illuminators	4
Maximum IR Distance	30 m
Water Ingress Protection	IP67

Table 3.2: Technical Specifications of the Exterior cameras.

Brand	Safety Vision
Model	43 series IP camera
Image Sensor	1/3” CMOS
Highest Resolution	1920 × 1080
Focal Length	2.8mm
Field of View	H: 80.0°, V:44.0°, D:93.5°
Video Compression	H.264, Motion JPEG
Infrared Illuminators	10
Maximum IR Distance	15 m

Table 3.3: Technical Specifications of the Interior Camera.

The interior camera is installed behind the windshield of the bus and looks forwards. There is an electronics cabinet that houses recording equipment and the automatic annunciation system inside the bus, in which we install and power our bus computer. The bus is a metro commuter running between downtown Pittsburgh and Washington County, and it makes two round-trips every workday. We can access the bus computer remotely using the cellular network when it is running or upload the raw data from it via WiFi when it returns to the garage.

3.4 Experimental Results

After we set up the test platform on the transit bus, we can carry out live experiments on some example applications. LiveMap, first introduced in prior work [18], is a typical application on our system which uses a coarse-to-fine detection pipeline to look for the target object along the roads. We reimplement it on the BusEdge platform, the pipeline of which is shown in Figure 3.9. A lightweight detector MobileNet-SSD [39, 52] is deployed on the bus to execute the early-discard filtering. The distilled images will be tagged with the GPS information and sent to the cloudlet. More accurate results are obtained by a heavier model Faster RCNN [61] deployed on the cloudlet and visualized on an OpenStreetMap tile server (as is shown in Figure 3.10).

Figure 3.8: Pictures of the Transit Bus and the Test Platform.

For the live experiment on the bus, we adopt several kinds of traffic signs asthe targets. The early-discard filter on the bus is a single-class general traffic sign detector, only making send-don't send decisions [23]. The cognitive engine will detect the traffic signs and classify them into six classes [9]: stop, yield, do-not-enter, pedestrian-warning, speed-limit and prohibition signs. Both of the detectors are trained on the Mapillary Traffic Sign Dataset [25]. We evaluate the average precision

at IOU=0.5 (AP50) of these two detectors on the Mapillary test set. The general traffic sign detector yields 50.3% precision and the detector on cloudlet achieves an mAP50 of 69.0% over all the six categories. The per-class AP50 of the cloudlet detector is also shown in Table 3.4. The bus drives from Washington County to downtown Pittsburgh with the application running. We can monitor the real-time trajectory of the bus as well as the detection results on the map server display. Figure 3.10 shows the qualitative result from LiveMap using the live bus data. The blue line on the map illustrates the entire route of the bus. The icons of different traffic signs are marked along the bus trajectory according to the GPS information of the images. Users can click on the icon to view the image and bounding boxes to verify the detection results.

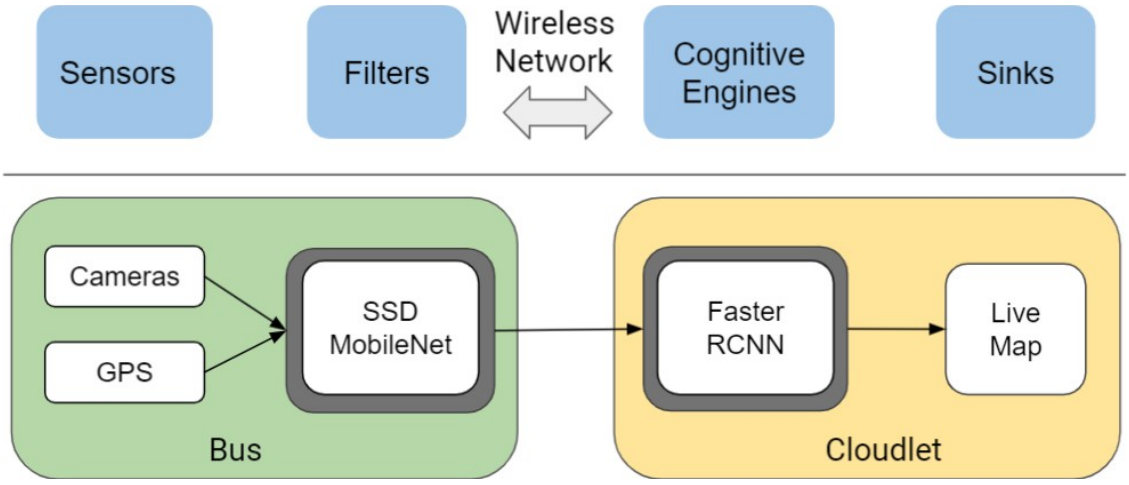


Figure 3.9: The Pipeline of LiveMap.

	stop	yield	pedestrian	speed-limit	prohibition	do-not-enter
AP50	0.86	0.79	0.82	0.53	0.52	0.64

Table 3.4: Per-Class AP50 of the Cloudlet Detector on the Mapillary Test Set.

	Precision	Recall	FPS	Total Bytes	Frame Fraction
Early Discard	—	67.5%	4.15	420 MB	17.5%
Cognitive Engine	78.9%	59.2%	15.8	29 MB	1.13%

Table 3.5: Quantitative Results of the Detection Pipeline.

To evaluate the performance of the LiveMap pipeline, we manually go over andlabel the entire route[2] for the target traffic sign. The quantitative results are illustrated in Table 3.5. We can see that the early-discard filter on the bus can save a huge amount of bandwidth by reducing the transmission frame fraction to only 17.5% with 420 MB in total. The cognitive engine on the server achieves a fairly good detection precision of 78.9% with a modest drop (-8.8%) in image recall from the distilled images.

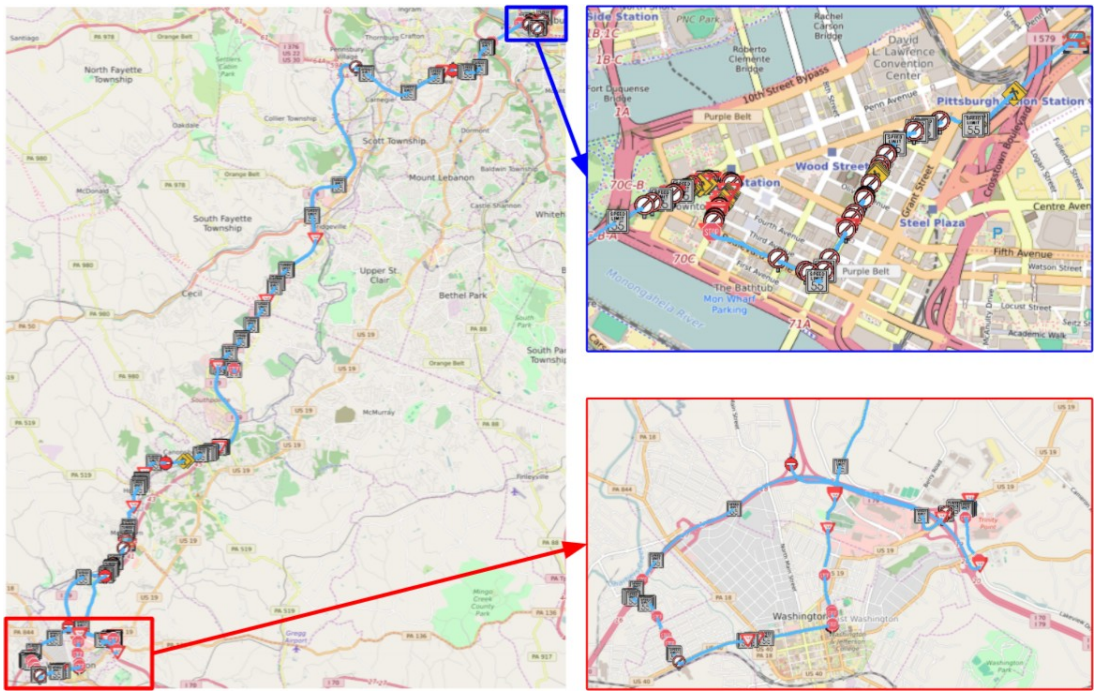


Figure 3.10: Qualitative Result of LiveMap Using Live Bus Data.

This again verifies our conclusion in prior work [18] that preliminary in-vehicle video processing can already save a lot of bandwidth without much loss of detection accuracy

[2] [1]This is the same dataset as the test set *Cloudy* constructed and used in Section 4 and will bedescribed in detail in Section 4.4

4. Auto-Detectron

For the object detection pipeline on BusEdge, we will often have a lightweight object detector deployed on the bus to execute the early-discard filtering and a more sophisticated model on the cloudlet server to have further data analytics. The biggest challenge here is how to rapidly obtain a well-performing detector for both the bus edge and the cloudlet given an ad hoc query object.

In our work, we propose an application called **Auto-Detectron** upon the BusEdge platform which enables users to easily deploy and run an object detection task for a given target using the live bus data. It integrates labeling and recursive learning to acquire object detectors and then automatically launches and updates them on our platform for continuous data filtering and analytics.

4.1 Overview

As is shown in Figure 4.1, the basic idea of Auto-Detectron is straightforward. For anad hoc search query, the user should first provide us with a few (5 to 10 shots) labeledimages for bootstrapping. Then the system will automatically train and deploy an initial object detector for filtering on the system. These distilled real-world images will then be provided to the user for manual labeling. With the new labeled data, better models will be recursively updated for the given task. This is achieved by retraining or fine-tuning the model with a growing training set. A detector with favorable performance is expected to be obtained after several update iterations.

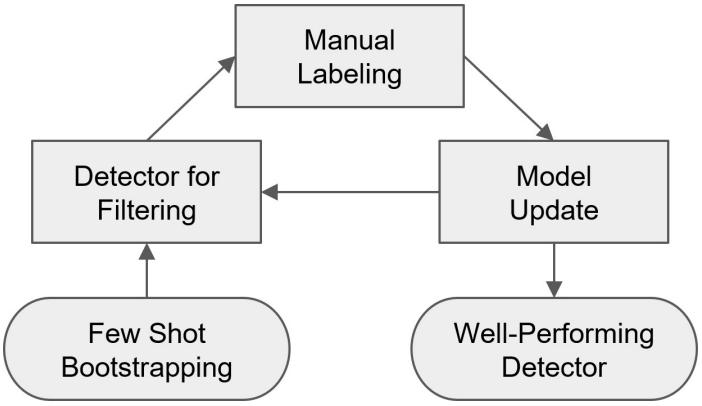


Figure 4.1: Core Idea behind Auto-Detectron.

The user can then use the new created object detection pipeline to search for the query object with the live data stream. The idea behind this work is inspired by [29], which also develops an interactive labeling system with recursive and scalable learning at the edge. Compared with their work, the Auto-Detectron application aims to perform object detection on real-world data instead of only classification on public datasets. Our goal is to develop a general object detection pipeline in the context of the BusEdge platform.

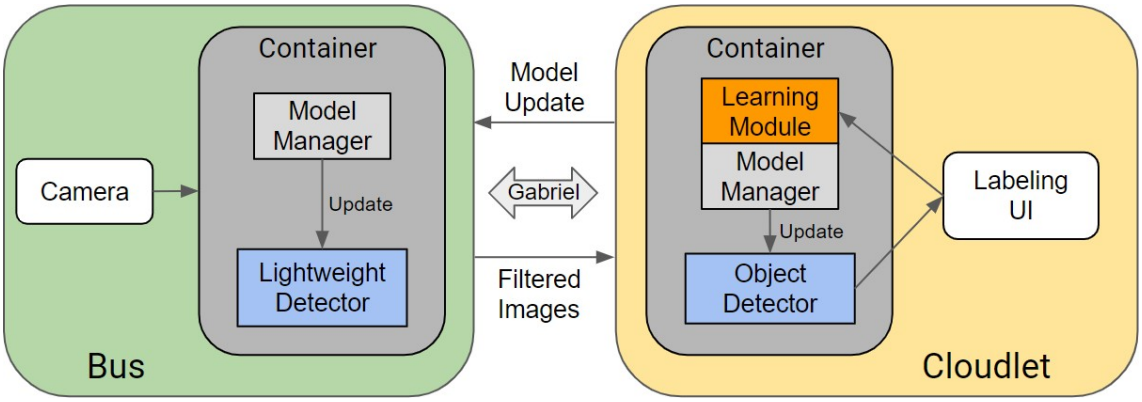


Figure 4.2: Overview of Auto-Detectron.

Figure 4.2 further illustrates the Auto-Detectron pipeline in the context of the BusEdge platform. A lightweight object detector is deployed on the bus for coarse filtering, the output of which will then be further refined by a heavier object detection model on the server. We will initialize these two detectors from some off-the-shelf pre-trained models and update their parameters every time we receive labeled data from the users. The model update is managed by a learning module on the server, which will adopt different learning strategies for both detectors taking into accountthe computing and network resources of different devices. All of the learning and update procedures are transparent to the user. The user only needs to provide some images for bootstrapping and then continuously do the annotation on the labeling UI.The rest of this section is organized as follows. Section 4.2 describes the imple- mentation of Auto-Detectron, including the model selection, learning strategies and pipeline design. Two improvement methods are proposed in Section 4.3 to enhance the performance of the detection pipeline. A dataset is constructed in Section 4.4 for the subsequent experiments. In Section 4.5, extensive experiments are carried out to evaluate the performance of each individual detector as well as the entire application pipeline.

4.2 Implementation

4.2.1 Problem Statement

In order to realize the desired functionalities of Auto-Detectron, there are three major components that require careful design and experimental verification.

Lightweight detector on the bus. As the early discard filter of the pipeline, the detector on the bus should lay more emphasis on the recall of the query object to avoid omission. Meanwhile, detection precision is also valuable to maximize the bandwidth saving. The limited computing and network resources on the edge impose constraints on the complexity of the detection model and update strategies.

Deep object detector on the cloudlet. The detection model on the cloudlet determines the final accuracy of the entire detection pipeline, which plays an important role in reducing the user's labeling workload. Thanks to the sufficient compute power on the cloudlet, the detection model can pursue much higher performance with the assistance of more sophisticated models and well-designed learning strategies.

Pipeline designs. The Auto-Detectron pipeline aims to integrate labeling, learning and automatic model deployment on the BusEdge platform. Bandwidth-efficient communication between the bus edge and the cloudlet is vital to satisfy the scalability and low end-to-end latency of edge computing. The learning and model management should happen in the background and be transparent to the users. No explicit user action is needed to trigger training and deploy a new model.

4.2.2 Model Selection and Learning Strategies

Detection on the bus

For the selection of the object detection model on the bus, model efficiency and lightweight update strategies are the two key factors in our decision. The ideal policyis to have a pretrained lightweight object detector and only retrain a minimal part of it during the recursive update.

One-stage object detection models like YoLo [58], SSD [52] and CenterNet [83] are quite popular for the applications on mobile devices. However, the design of heatmap prediction at the end of these one-stage models brings difficulties to efficientretraining

and update. On the one hand, the large size of the last fully connected layer makes it impractical to execute rapid fine-tuning and transmit it over the internet during iterative update. On the other hand, we cannot directly replace the last prediction layer with a standalone classifier like SVM due to the absence of the proposal generation.

Conversely, we can benefit from the proposal generation network of the two-stage detectors. Figure 4.3 shows a simplified network architecture of the two-stage object detection model. The straightforward box classification head in two-stage approaches makes the replacement of the final classifier possible. In order to meet the efficiency requirements, compact backbones like MobileNet [39, 63] and FBNet [21, 73, 80] areall good candidates, but we should also take into account the efficiency problems of the RPN and ROI head in the two-stage architecture.

As is shown in the left diagram of Figure 4.4, we will use a modified Faster R-CNN architecture with FBNetV3-A [21] as the backbone and corresponding lightweight RPN and ROI head [73] as the proposal feature extractors. We adopt most of the architecture design and training settings in [5].

The major modification here is to replace the last fully connected layer for classification with a support vector machine (SVM) classifier. During the recursive learning process, we will only retrain and update the SVM classifier at the end of the model but freeze all the other parameters of the detector. Retraining a SVM for model update has several advantages over fine-tuning when the training set is small:
1) much shorter training time, on the order of seconds; 2) fewer hyper-parameters to determine for training; 3) small size for transmission and reloading. These features bring benefits to the bus client in terms of low latency and bandwidth saving.

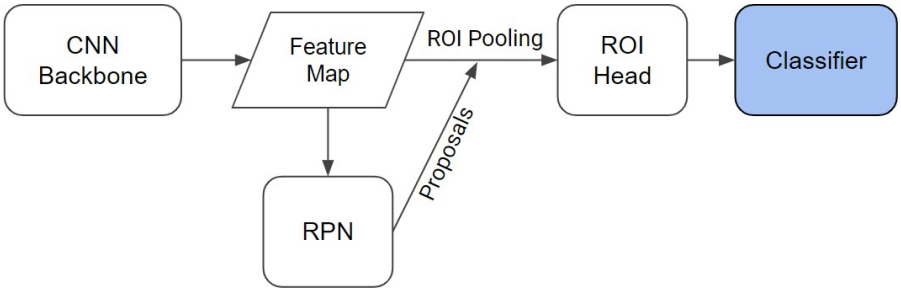


Figure 4.3: Typical Architecture of a Two-Stage Object Detector.

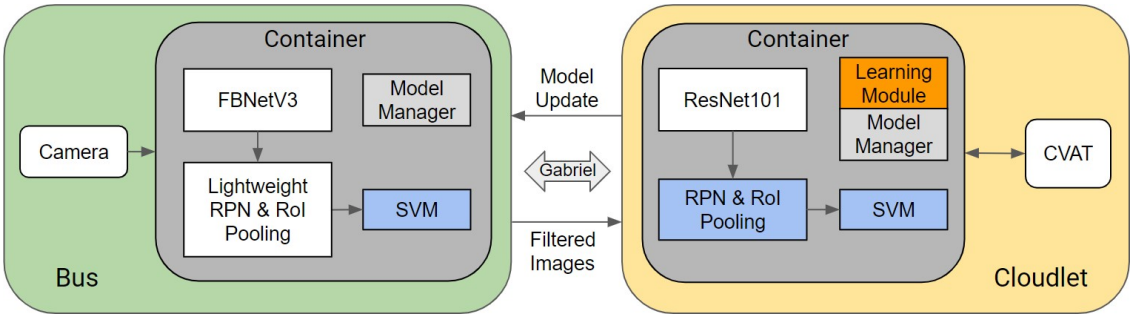


Figure 4.4: Implementation Details of Auto-Detectron.

Detection on the cloudlet

For the heavier detection model running on the cloudlet server, the Faster R-CNNwith ResNet-101 will be deployed because of its state-of-the-art object detection performance, as is shown in the right diagram of Figure 4.4. However, the update strategies of the cloudlet model require further discussion and experiments.

Likewise, the classification head of the model can be replaced with an SVM classifier because of the two-stage architecture. We adopt this modification here because we want to update the model more frequently to acquire the best performancewith the freshly-labeled data. With an always up-to-date SVM classifier, the users can have a better labeling experience due to the low-latency positive feedback.

Nevertheless, fine-tuning the deep neural network from time to time is still a complementary learning strategy. Even though fine-tuning a deep model is vulnerableto overfitting in the absence of sufficient labeled data, we can still manage to train a better model with the assistance of some few-shot learning tricks [70, 76], which is significant to raise the performance upper bound of the detector. We can also trigger fine-tuning of the model less frequently to satisfy the training time and also wait for more training data. Having the fine-tuning running in the background is another option to cope with the long training time. In our implementation, we will combine the frequent SVM training and periodic model fine-tuning together to update the object detector on the cloudlet. The detailed learning strategy will be explored and discussed through experiments in Section 4.5.2.

4.2.3 Pipeline Details

Figure 4.4 illustrates the details of the Auto-Detectron pipeline. Because of the selection of bus detection model and update approach, we will only update the SVM classifier of the bus model during the entire labeling and learning process. However, different update strategies will be adopted on the cloudlet considering the size of the labeled set and model efficiency. The entire training process will be achieved by a learning module running in the background. The communication between the bus andcloudlet is managed by Gabriel flow control, including the transmission of distilled images and new SVM model. For the annotation tools, we will use the Computer Vision Annotation Tool (CVAT) [4]. The user only needs to register the detection pipeline and then manually label the images shown on the CVAT UI.

4.3 Improvement Methodologies

With the two selected models deployed on the Auto-Detectron pipeline, we are able to basically achieve the desired functionality, but the detection performance is still far from satisfactory in our preliminary experiments (which are listed as the baselinein Section 4.5). Therefore, we will first explore possible approaches for improvementin this part.

4.3.1 Positive Mining

We notice that the deficiency of positive examples is one important factor limiting the performance of the model. For one thing, the training set during the first few update iterations is quite small, providing only a few shots of positive samples. For another thing, the frozen RPN of the model tends to generate proposals of the base classes in its original pretrained dataset instead of our target class, which will do harm to both the training and inference process.

In order to retrieve more positive samples during the forward pass, our insight is to rescue the low objectness positive anchors that are suppressed by the frozen RPN. This is inspired by a training trick of few-shot learning in [70]. The method is to double the maximum proposal number of non-maximal suppression (NMS) during proposal generation and halve the number of sampled proposals of the final RPN output. The intuition here is to bring more foreground proposals for the target instances and meanwhile discard more background proposals during sampling.

Moreover, we can make use of data augmentation with multiple training epochs

when training the SVM classifier. We can traverse the training set for multiple times and apply data augmentation to the images to construct a larger training set for SVM. This can help the detector to cope with the variations in illumination, weather and viewpoint of the bus data to some extent. However, this method should only be used when the training set is small (e.g. less than 100 positive images in our experiments), because it will increase the training time of SVM greatly with a larger dataset.

4.3.2 Utilize Hard Negatives

During our preliminary experiments, we also find that the false positives are the main source of errors during the entire training process. Nevertheless, we can easily obtain the false positive results from the manual annotation. These hard negatives can play an important role in improving the detection precision. In the original implementation of Faster R-CNN, the knowledge of these hard negatives cannot be used in training because it will only look at the images containing positive labels. Ouridea is to propel the model to lay more emphasis on these hard negative examples. The naive method is to enlarge the training set with the hard negatives and disable the filtering of images without annotations during training. Another strategy is to add a new class for the hard negatives in classification. This modification will propelthe model to sample more proposals around the previous false positive results in RPN and increase the inter-class spacing between the target objects and the distracting ones during training.

4.4 Dataset

To evaluate the performance of our system with real-world data, we have collected data using the cameras on the transit bus and manually labeled them. We term this dataset "BUS"[3][1]. We adopt pedestrian warning signs as the target object. We will also include some other classes when evaluating the end-to-end system performance in Section 4.5.3, but they don't need detailed bounding box annotations, so we will skip them in this section.

[3] The dataset is available at https://www.kaggle.com/albertye/busedge-pedestrian-signs

	Train	Test			
	Sunny	Cloudy	Snowy	Rainy	*Total*
Entire Route	7007	7974	1548	4736	14258
All Signs	966	1435	106	704	2245
Pedestrian	117	120	29	92	241

Table 4.1: Statistics of the BUS Dataset.

As is mentioned in Section 3.2.4, we deploy a preprocessing module right after the raw video stream to remove repeated or blurred images. The images after the preprocessing module will become the raw data of each route to construct our dataset.The preprocessing steps include: 1) Deduplication of nearly identical frames; 2) Removal of blurry frames; 3) Reduction of frame rate to no higher than 1 Hz. Such a preprocessing module can save a lot of annotation efforts without much loss of usefulinformation. Detailed statistics of our BUS dataset are shown in Table 4.1. Four routes in different weather conditions (sunny, cloudy, snowy, rainy) are selected to be the image sources (illustrated as the "Entire Route" row). One of them (sunny)is used for training and the other three routes (cloudy, snowy, rainy) are for testing. To boost the evaluation of different approaches when selecting detection models and update strategies, we also build a subset out of the entire route based on whether any traffic sign appears in the images (illustrated as the "All Signs" row).

(a) Pedestrian signs with different shapes and colors.

(b) Similar traffic signs which might confuse the detector.

Figure 4.5: Some Cropped Images from the BUS Dataset.

The annotation statistics of our target class are also listed in the last row of Table 4.2.There are several reasons why we select the pedestrian warning sign as the target

object: 1) Sufficient number of occurrences along the route; 2) Diverse variety in shapes and colors, as is shown in Figure 4.5a; 3) A lot of similar signs which might confuse the detector, as is shown in Figure 4.5b.

In addition, it should be noted that we need different construction of training and test sets for different evaluation tasks. For the end-to-end performance evaluation of the system, the images of the entire route with its original timestamp order will be used for both training and testing, because we want to obtain the system performance under live running conditions. Therefore, we will use the entire route of the sunny day with 7007 images as the training data (termed "*Sunny* ") and use the cloudy route with 7974 images for testing (termed "*Cloudy*").

For the evaluation of each individual detection model, the subset of general traffic signs can be used. We will use the merged set with 2245 images as the test set (termed "*Merged*"). In terms of the recursive training, we will use the 115 positive images in the training set. In order to conduct repeated experiments to obtain convincing results, we will shuffle the training set with three different random seeds to have three runs.

In addition, it is worth noting that the "Hard Negative Augmented" method requires a special training set with hard negative samples added, which are actually the output of the previous model as the learning is progressing. To acquire these intermediate results and also keep the consistency of the training data, we use the best baseline model in Section 4.5.2 to process the raw training data and then manually label the false positives (obtain 274 extra images with hard negative class). During training, these hard negatives will be randomly sampled and used to enlarge the training batch of each update iteration. The arrangement of the training set and test set for different evaluation tasks is shown in Table 4.2

Evaluation Tasks	Training	Test
Methods w/o Hard Negatives	117	*Merged*
thods with Hard Negatives	117 + 274	*Merged*
System Evaluation	*Sunny*	*Cloudy*, *Merged*

Table 4.2: Dataset Arrangement for Different Evaluation Tasks.

4.5 Experiments

In this section, extensive experiments are performed to evaluate the performance of the proposed system. The prerequisite for the success of the system is to obtain well-performing detectors both for the bus and the cloudlet. These object detection models need to be able to achieve continuously improving accuracy with recursive update and acquire favorable performance after only several iterations. Subsequently,we can apply the selected models and update strategies to our proposed system pipeline and conduct end-to-end evaluation of the entire system.

4.5.1 Evaluation of Bus Detector

For the evaluation of the detector on the bus, there are three questions we aim to answer: 1) How is the performance of the selected lightweight object detector in our challenging settings; 2) Whether the proposed methods can improve the detection performance of the model; 3) Where is the performance bottleneck and the source ofthe detection errors.

Figure 4.6 illustrates the Average Precision at IoU 50% (AP50) of different approaches on the *Merged* test set as the labeling and learning are progressing. The halo around each curve is the standard deviation across three experimental runs. The curves in this chart can indicate the overall performance of the detectors during the update. We will trigger the learning process every 10 new positive images. In order to learn more about the effects of the modifications, we also provide the precision-recall curves of these models at different update steps in Figure 4.7. We will explain these results and analyze the performance of different approaches in the following.

Baseline. As is mentioned in 4.2.2, we will use lightweight Faster R-CNN with FBNet backbone as our bus detector and only replace the classifier with an SVM. As is shown by the red solid curve in Figure 4.6, the accuracy of the model can be improved during the training and achieve an AP50 of around 0.25 in the end, which is not bad given the limited resources and challenging detection settings on the bus. This also indicates the effectiveness of the simple update strategy on the bus detector, which only retrains the final SVM classifier in each iteration.

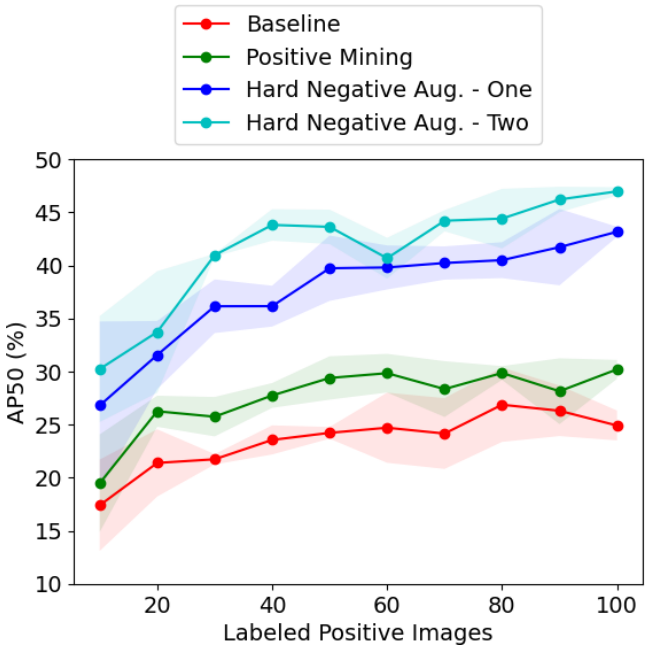
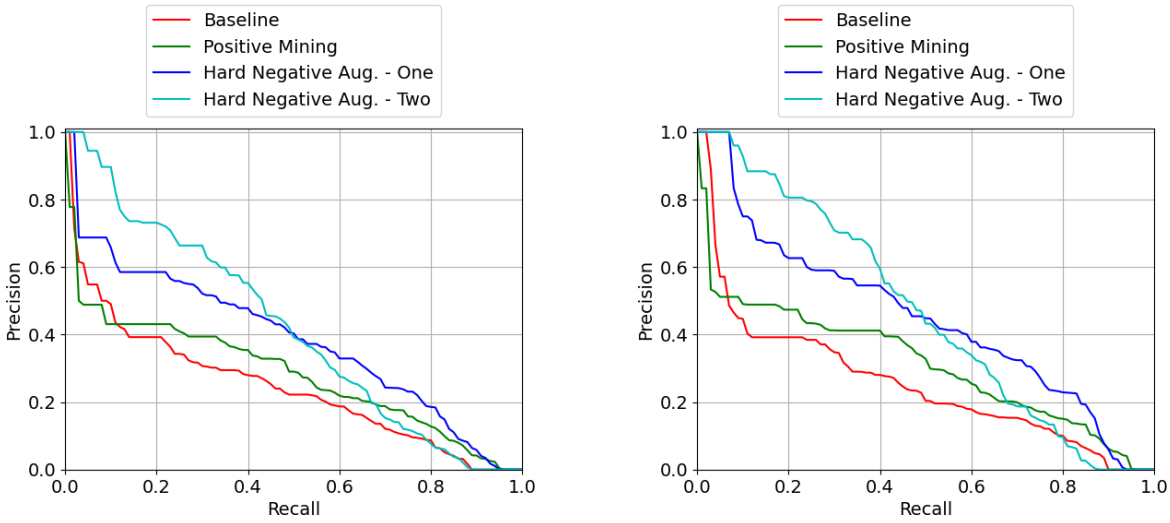


Figure 4.6: AP50 Curves of Different Approaches During Update.



(a) At Update Step 5th (b) At Update Step 10th

Figure 4.7: Precision-Recall Curves of Different Methods during Update.

Positive mining. The green solid line in Figure 4.6 shows the performance of the modified model with positive mining. We can see that it outperforms the baseline by about 5% over the whole process of recursive learning. The precision-recall curves in Figure 4.7 indicate that the accuracy gain mainly comes from the higher recall of the query objects, which is favorable considering the recall requirement of the bus detector.

Hard negative augmented. We term the two different hard negative augmented approaches *Hard Negative Aug.-One* and *Hard Negative Aug.-Two* to indicate the category number of the classification head. Both methods are based on the implementation with positive mining added already. As is shown in Figure 4.6, both modifications can achieve a huge performance enhancement of 10%-15% over the other methods, especially when more training data are provided. This is reasonable considering that the hard negative augmented training set is two times bigger than theoriginal one. Even though all these added images do not contain positive examples, the added hard negatives can guide the model to distinguish distracting proposals better instead of letting the diffusive backgrounds dominate the model training.

For the choice of category number, we can learn from Figure 4.6 that the overall accuracy of the two-class method is higher. However, the PR curves in Figure 4.7 indicate that the accuracy gain of the two-class method is mainly from reducing false positives while the one-class method achieves a considerable boost for boththe recall and precision. Considering that higher recall is more important for thebus detector, the *Hard Negative Augumented-One* method will be the best option here.

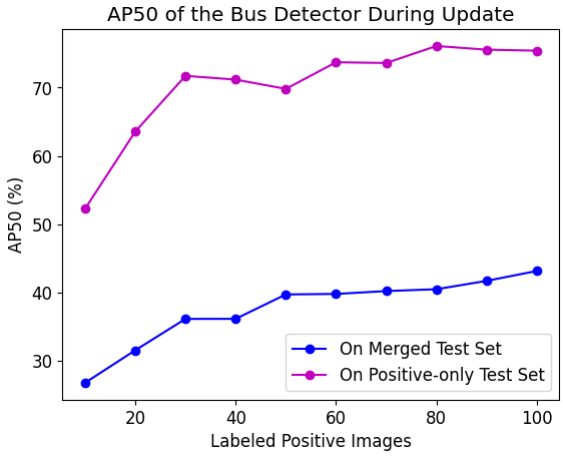


Figure 4.8: Detection Performance on Different Test Sets.

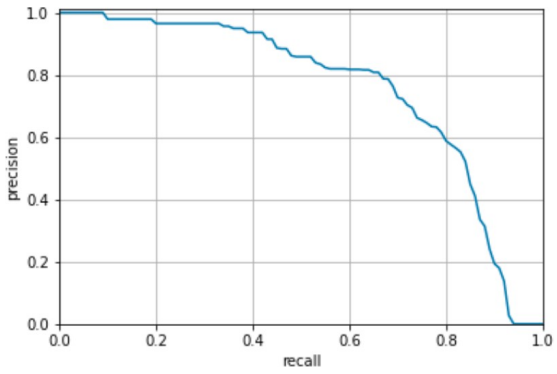


Figure 4.9: Precision-Recall Curve on the Positive-Only Test Set.

Performance bottleneck. During the experiments, we find that the AP50 of the detector hovers around 45%, even though we have applied both modifications and updated the model with more labeled images. It seems that the performance of the model has reached a plateau when trained with fewer than 100 labeled images. To explore the performance bottleneck of the model, we also test the models of our selected method (i.e., hard-negative augmented with one class) on a test set containingonly the positive images. Figure 4.8 compares the AP50 curves on the original *Merged* test set and the positive-only test set. Figure 4.9 shows the precision-recall curve of the model at step 10 on the positive-only test set. We can observe that there is a huge performance gap on the two test sets, which indicates that the false positives are still the major source of the errors although we have added hard negative to the training set. The limited capacity of the model is one possible reason, considering that we only retrain the SVM classifier but freeze all the other network parameters.

4.5.2 Evaluation of Cloudlet Detector

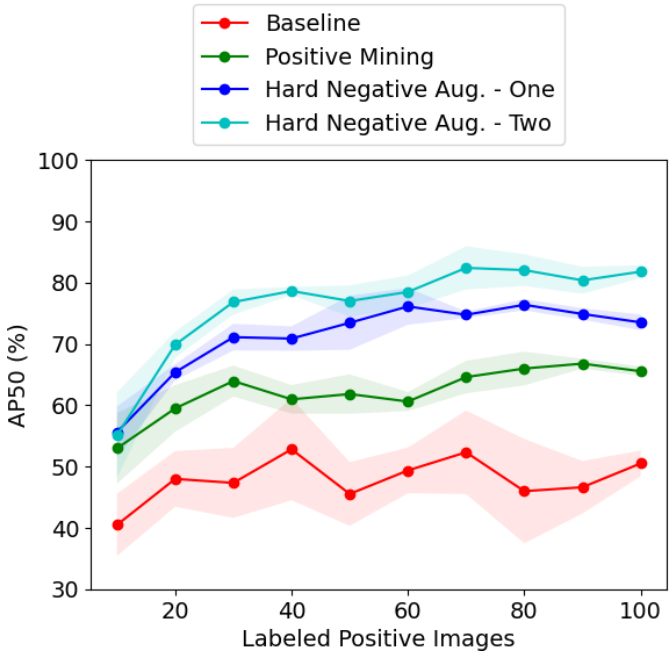
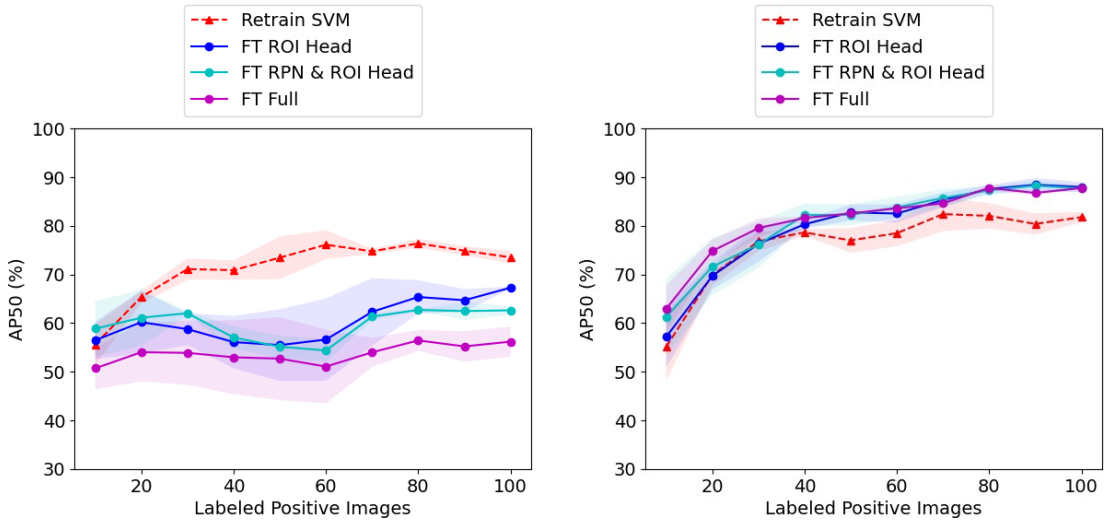


Figure 4.10: AP50 Curves of Models When Retraining Only SVM.

For the evaluation of the detector on the cloudlet, there are also three aspects we aimto explore: (1) How is the performance of the FRCNN-SVM model on the challenging bus data; (2) Whether the proposed methods can improve the detection performanceof the model; (3) The comparison between different learning strategies, including only retraining SVM and fine-tuning more layers of the neural network.

Retraining only SVM. We first follow the same update strategy as on the bus, which is only retraining the final SVM classifier of the model. The AP50 curves of different approaches are shown in Figure. 4.10. We can see that the curves look quitesimilar to the previous experiment on the bus models. Both the modifications can improve the detection performance. The positive mining method is more helpful at bootstrapping and the hard negative augmented methods can achieve larger performance improvement with more labeled data. The final AP50 of the best model can reach 80% by only retraining the SVM. This indicates the excellent performance of the pre-trained feature extractor. However, we can see that the accuracy curves of this simple update strategy do not rise steadily but have fluctuations. Moreover, these models all reach a performance plateau early during the update. It will be good to explore if other methods can solve these problems.



(a) Hard Negative Augmented with One Class

(b) Hard Negative Augmented with Two Class

Figure 4.11: AP50 Curves of Different Models and Update Strategies.

Fine-tuning more layers. When exploring the effect of fine-tuning more layers, we will by default apply the two proposed modifications, i.e., positive mining and hard negative augmented, but the suitable class number of the classifier remains to be explored.

The AP50 curves of the model under different update strategies and class numberare shown in Figure 4.11. If we compare the two diagrams in general, it is easily noticed that the one-class method is outperformed by the two-class method to agreat extent

when using fine-tuning as the learning strategy. This is because thehard negative samples are actually treated the same way as the common background instances in the one-class method during the fine-tuning. The RPN will not distinguish these hard negatives with the other background instances. All these samples havethe same weights during training. In this case, those easily classified negatives will comprise the majority of the training samples and dominate the gradient. The two-class method, however, will propel the model to sample more hard negatives during proposal generation and also space the proposals with different labels apart in the projection space during backpropagation. Therefore, the two-class method is more favorable when fine-tuning the cloudlet model.

We can also see in Figure 4.11b that the overall accuracy of the fine-tuning strategyis higher than that of only retraining SVM, especially when more labeled data are provided. In terms of the number of layers to be fine-tuned, we can see from 4.11b that the three options arrive at a similar AP50 at the end of training, but fine-tuning more layers works better at the beginning. Taking into account the detection performance, the generalization ability and training time of the model, we opt to unfreeze theentire model except the backbone (i.e., train the RPN and ROI head) when fine-tuning.

Discussion. As is discussed above, both update strategies of fine-tuning and retrain- ing SVM have their own pros and cons. The fine-tuning method can provide a better and more stable detection performance but it requires more training time. In our experiments, it takes 5-10 minutes to execute each update step for fine-tuning[4], but retraining SVM requires less than half a minute.

However, having an SVM as the classifier will slow down the inference speed on the cloudlet. The prediction of linear SVM is to calculate the inner product of the support vectors and the input feature, which has time complexity of $O(N_{sv} \times D)$, with N_{sv} the number of support vectors and D the dimension of features. The SVM model will become heavier with more training data. We try two implementations of SVM in our experiments. The first one is to run the inference of SVM on CPUs using Scikit-learn [55] and the other is on GPUs using ThunderSVM [79]. Figure 4.12 shows the inference time of different methods. The red solid curve shows the inference time when we execute SVM on CPUs while the cyan curve shows the inference time on GPUs. On the one hand, the SVM model consumes an increasing prediction time when using CPUs only. It requires five times more inference time (374 ms/frame) than the fine-tuning method (61 ms/frame) when 100 positive images are provided. On the other hand, using parallel

[4] We set a constant number of fine-tuning iterations for each update step in our experiments. We find that this configuration can achieve a performance close to that obtained by increasing the iteration number linearly.

computing on GPUs can achieve a stable predictiontime (196 ms/frame) but it is still much slower than the end-to-end Faster R-CNN detector.

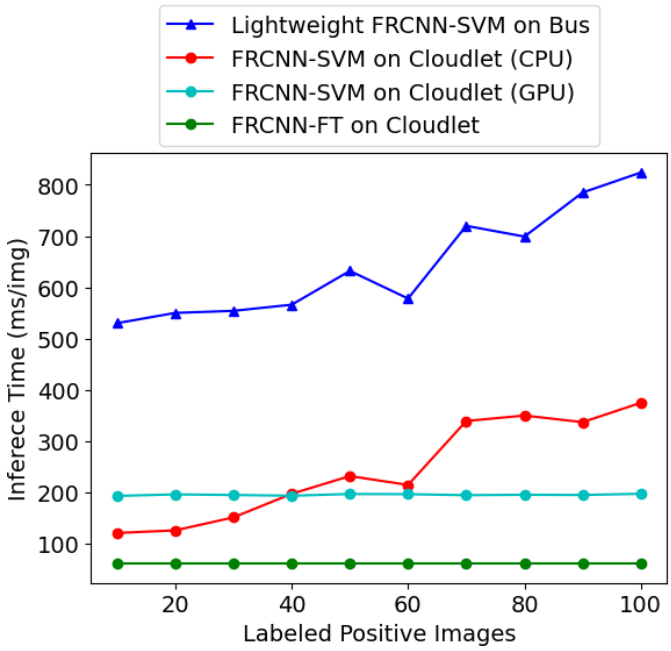


Figure 4.12: Inference Time Comparison of Different Methods.

In conclusion, retraining SVM is more favorable when low-latency feedback is emphasized while fine-tuning does a better job in terms of detection accuracy and computation efficiency. In our implementation, we combine the two update strategies together. Specifically, we will update the SVM every time we receive new data but also fine-tune the model in the background. The newest model trained with more labeled data will be adopted as the cloudlet detector. We will disable the trainingof SVM when sufficient data are provided (e.g. 60 positive images) considering the performance plateau and slower inference time. In this case, we will have a better detector trained by fine-tuning if the annotation progresses slowly and will also be responsive to new annotations in the first few update iterations.

4.5.3 System Evaluation

Test on the BUS Dataset

To evaluate the performance of the Auto-Detectron pipeline, we will use the entire route *Sunny* as the input data for training and labeling. It is worth mentioning that the system is meant to run online but we are testing it offline here to conductquantitative

experiments with a labeled dataset.

We first download 10 shots of pedestrian sign images from the Internet and use them to bootstrap the system. Then the system starts to perform object detection on the bus video. We can see the distilled images on the labeling UI (as is shownin Figure. 4.13). A new model will be automatically trained and updated on the pipeline if sufficient labeled data are provided. In our experiments, we update themodel every 10 new positive images.

Figure 4.13: Schematic Diagram of Auto-Detectron from the User Perspective.

Throughout the entire route of Sunny, we end up annotating 128 images in total and obtaining 50 positives from them, from which we also acquire five pairs of objectdetectors (not including the bootstrapping model). Table 4.3 gives a summary of the entire image filtering process. Though this detection pipeline is just initialized by a few shots of web images and still under periodic training during the filtering process,the bus detector still achieves a great bandwidth saving with a good recall of 58.1%. The cloudlet detector can also recognize most of these true positives. We also notice that there are more false positives at the first few update steps, which leads to the relatively low overall accuracy of the pipeline. This is reasonable considering the lackof training data at the beginning.

	All Images	True Positive	Precision	Recall
Entire Route	7007	117	—	—
Sent by Bus	846	68	8.03 %	58.1 %
Sent by Cloudlet	128	50	39.1 %	42.7 %

Table 4.3: Quantitative Results of the Labeling and Learning Process on *Sunny*.

To evaluate the performance of the learning module, we test each individual detector using the *Merged* test set. Figure 4.14 shows the AP50 of each detector during the update. We also provide the precision-recall curves of these models on Figure 4.15. We can learn from the AP50 curves that both the bus and cloudlet detectors can achieve improving accuracy with growing training data and have a fairly good performance with only 60 labeled images provided in the end.

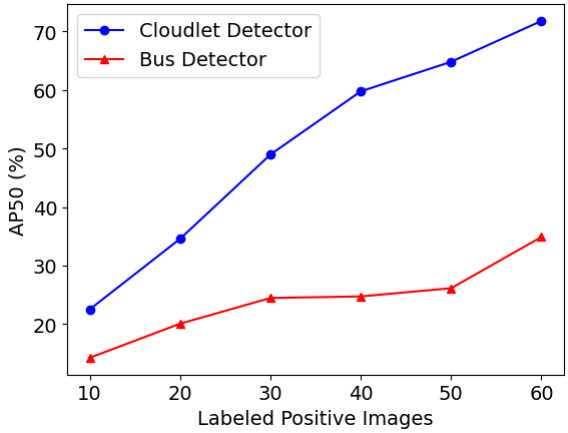
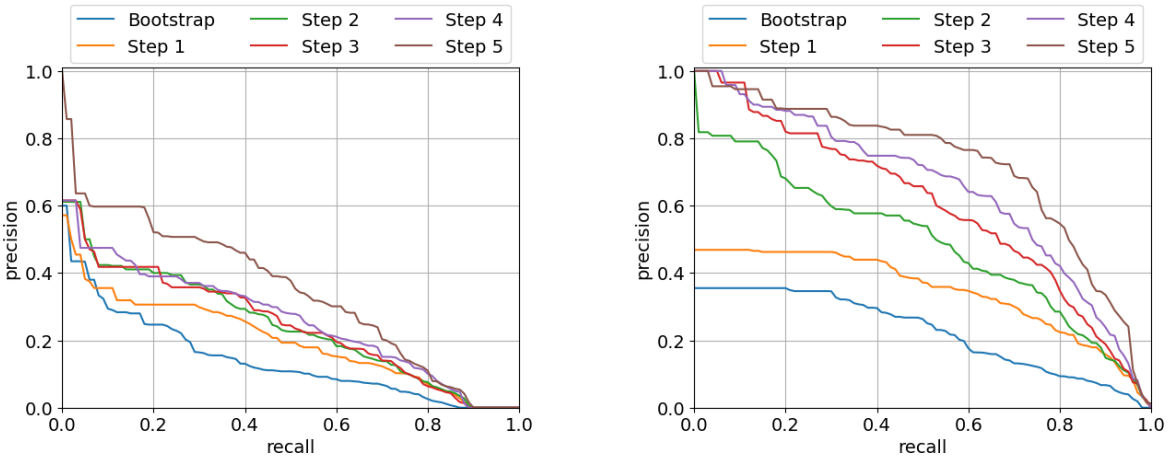


Figure 4.14: AP50 of the Detectors During Recursive Training.

To further evaluate the end-to-end performance of the newly deployed detection pipeline, we continue to test it on the entire *Cloudy* test set. Without any further labeling or learning, we will only use the final coarse-to-fine detection pipeline to process the *Cloudy* route. The quantitative results are illustrated in Table 4.4. For comparison, the results from previous LiveMap is also provided in 4.5. We can see that the newly deployed bus model can achieve much more bandwidth saving (from 420 MB to 168 MB) with a similar recall compared with the LiveMap pipeline. Moreover,the final recall of the new detection pipeline is 60.8%, with only 5.0% difference fromthe coarse filtering but with a huge precision gain of 72.7%. Both the precision and recall rate are better than the LiveMap pipeline. This indicates the success of the online training model on the real-world bus data compared to the model pretrained only on a public dataset.



(a) Bus Detectors (b) Cloudlet Detectors

Figure 4.15: Precision-Recall Curves of the Detectors During Recursive Training.

	Precision	Recall	FPS	Total	Bytes	Frame Fraction
Early Discard	12.2%	65.8%	1.59	168	MB	8.13%
Cognitive Engine	84.9%	60.8%	16.4	22	MB	1.09%

Table 4.4: Quantitative Results of the Auto-Detectron Detection Pipeline on *Cloudy*.

	Precision	Recall	FPS	Total Bytes	Frame Fraction
Early Discard	—	67.5%	4.15	420 MB	17.5%
Cognitive Engine	78.9%	59.2%	15.8	29 MB	1.13%

Table 4.5: Quantitative Results of the LiveMap Detection Pipeline on *Cloudy*. (Same as Table 3.5.)

More Experiments on Different Classes

In order to demonstrate the versatility of the Auto-Detectron pipeline, we also conduct quantitative experiments on more classes. Similarly, we will first bootstrap the application using some web images of the target and then use the *Sunny* routefor labeling and training. We don't need to go through the entire training route but can stop labeling when we are satisfied with the performance of the detector. Subsequently, we can test the newly deployed detection pipeline on the entire Cloudy route. Images sent by the bus and the cloudlet detectors will be saved, from whichwe can manually count the true positives and calculate the detection precision.

As is shown in Figure 4.16, we include four more object classes that are common along the road. They are stop signs, speed-limit signs, garbage cans and traffic cones.

Figure 4.16: More Classes for System Evaluation.

The quantitative results of these additional experiments are listed in Table 4.6. The "Train" column illustrates the number of images we have labeled during training and how many of them are actually positive. This can indicate the positive rate and how much labeling effort is needed during training. It can be observed that we only need to label tens or a few hundreds of images to yield good detection performance. One potential problem here is that it might require much time to wait for positive images or there may not even be enough positive examples along the route, when the target object is quite rare. This problem can be solved if we can have a fleet of buses running at the same time to acquire abundant live traffic data. Reexamining the old cached data on the bus is also a feasible solution to enlarge the training set.

The other columns in Table 4.6 show the performance of the detection pipeline on the *Cloudy* route. For the bandwidth efficiency, we can see that the bus detector on all the four experiments can reduce the transmitted data to a great extent, and meanwhile they can still achieve a fairly good accuracy. In terms of the performance of the cloudlet detector, it can achieve a very high precision with rare omission of the true positives sent by the bus. This again demonstrates the favorable performance of the detectors trained with only a modest number of live bus data. In addition, we also notice that the detection of stop signs and garbage cans achieves a much better precision either on the bus or cloudlet. Possible reasons include: 1) Fewer similar and confusing objects along the route; 2) The pretrained dataset contains similar classes. In conclusion, the experiments on more classes verify the performance and versatility of the Auto-Detectron application. The user can easily perform an accurateand efficient object detection pipeline on the BusEdge platform to execute an ad hoc search query.

	Train	Test on *Cloudy* Route (with 7974 images)					
	Positive/ Labeled	Sent by Bus	True Positive	Precision	Sent by Cloudlet	True Positive	Precision
Stop	50/71	173	119	68.8%	118	113	95.8%
Speed Limit	100/394	737	124	16.8%	111	76	68.5%
Garbage Can	130/225	510	318	62.4%	307	285	92.8%
Traffic Cone	40/77	545	132	24.2%	108	78	72.2%

Table 4.6: Experimental Results for More Classes. “Sent by Bus” shows the results after early-discard filtering from the bus and “Sent by Cloudlet” shows the images provided to the user for annotation.

5. Conclusion and Future Work

We propose a system architecture, BusEdge, that uses edge computing to achieve efficient data collection and analytics on transit buses. It provides a scalable and extensible platform for many promising applications to make use of the live traffic data captured by the cameras on the bus. It uses the in-vehicle computer on the bus to perform preliminary video analytics for data distillation and uses the cloudlet to carry out more sophisticated computer vision tasks. Furthermore, we implement and deploy this system on a metro commuter bus running between Pittsburgh and Washington County for live demonstration. We also use it as a platform to collect data and test new applications.

Coarse-to-fine object detection is the most typical application upon the BusEdge platform, but it requires a lot of effort to obtain a good object detector given a specific target. In order to boost the development of similar applications, we present Auto-Detectron, which integrates labeling, recursive learning and automatic model management to rapidly launch a specific object detection task upon the BusEdge platform. Moreover, various model modifications and update strategies are explored to enhance the performance of the detection pipeline. Extensive experiments are conducted using the real-world bus data to evaluate the performance of the application.

Several potential areas of future research and continuation exist for BusEdge. First of all, the system has room for improvement. To enhance the scalability of the system, some application-aware adaptation techniques [74] can be adopted to avoid and cope with the overload of the cloudlet in extreme conditions. The poor network conditions on a running bus also require extra consideration to achieve stable vehicle-cloudlet communication. More applications can be implemented on the system to extend the value of the platform and the live bus data. In addition, various advanced computer vision algorithms can be explored and applied in the Auto-Detectron application. For example, the object detector based on few-shot learning algorithms might enhance the performance of the bootstrapping model. It is also worthwhile to explore if we can make use of some self-supervised or semi-supervised methods to reduce or even get rid of the manual labeling efforts required for updates.

Bibliography

[1] Amazon Web Services. https://aws.amazon.com. 2.3

[2] Microsoft Azure. https://azure.microsoft.com. 2.3

[3] Google Cloud. https://cloud.google.com. 2.3

[4] CVAT. https://github.com/openvinotoolkit/cvat. 3.1, 4.2.3

[5] D2Go. https://github.com/facebookresearch/d2go. 4.2.2

[6] Gabriel Codebase. https://github.com/cmusatyalab/gabriel. 3.2.2

[7] Kibana. https://www.elastic.co/kibana/. 3.1

[8] Mobileye. https://www.mobileye.com/. 2.1

[9] HD Map Update. https://mscvprojects.ri.cmu.edu/2020teamg/project/.3.4

[10] Roadbotics. https://www.roadbotics.com/. 2.1

[11] Tableua. https://www.tableau.com/. 3.1

[12] Tesla. https://www.tesla.com/, . 2.1

[13] Tesla Autonomy Day. https://www.youtube.com/watch?v=Ucp0TTmvqOE, . 2.1

[14] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmarkanalysis of representative deep neural network architectures. *IEEE Access*, 6: 64270–64277, 2018. 2.2.2

[15] Manuel Carranza-García, Jesús Torres-Mateo, Pedro Lara-Benítez, and Jorge García-Gutiérrez. On the performance of one-stage and two-stage object detectors in autonomous vehicles using camera data. *Remote Sensing*, 13(1):89, 2021. 2.2.2

[16] Xinlei Chen, Abhinav Shrivastava, and Abhinav Gupta. Neil: Extracting visual knowledge from web data. In *Proceedings of the IEEE international conference on computer vision*, pages 1409–1416, 2013. 2.2.4

[17] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, et al. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017. (document), 2.4, 2.2, 3.2.2

[18] Kevin Christensen, Christoph Mertz, Padmanabhan Pillai, Martial Hebert, and Mahadev Satyanarayanan. Towards a distraction-free waze. In *Proceedings of the20th International Workshop on Mobile Computing Systems and Applications*, pages 15–20, 2019. 2.1, 3.4, 3.4

[19] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. 2.2.1

[20] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural informationprocessing systems*,

pages 379–387, 2016. 2.2.1

[21] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, et al. Fbnetv3: Joint architecture-recipe search using neural acquisition function. *arXiv e-prints*, pages arXiv–2006, 2020. 2.2.2, 3.2.4, 4.2.2, 4.2.2

[22] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision andpattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005. 2.2.1

[23] Hunter Damron and Christoph Mertz. Traffic sign detection and localization onthe edge for hd map updating. *CMU RISS Journal*, 8:58–62, 2020. 3.4

[24] Santosh K Divvala, Ali Farhadi, and Carlos Guestrin. Learning everything about anything: Webly-supervised visual concept learning. In *Proceedings of the IEEEConference on Computer Vision and Pattern Recognition*, pages 3270–3277, 2014. 2.2.4

[25] Christian Ertler, Jerneja Mislej, Tobias Ollmann, Lorenzo Porzi, Gerhard Neuhold, and Yubin Kuang. The mapillary traffic sign dataset for detection and classification on a global scale. In *European Conference on Computer Vision*,pages 68–84. Springer, 2020. 3.4

[26] Qi Fan, Wei Zhuo, Chi-Keung Tang, and Yu-Wing Tai. Few-shot object detection with attention-rpn and multi-relation detector. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4013–4022, 2020. 2.2.3

[27] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE trans- actions on pattern analysis and machine intelligence*, 32(9):1627–1645, 2009. 2.2.1

[28] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on MachineLearning*, pages 1126–1135. PMLR, 2017. 2.2.3

[29] Shilpa George, Haithem Turki, Ziqiang Feng, Deva Ramanan, Padmanabhan Pillai, and Mahadev Satyanarayanan. Integrating labeling and learning for edge computing. Unpublished manuscript, 2021. 4.1

[30] Spyros Gidaris and Nikos Komodakis. Dynamic few-shot visual learning without forgetting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4367–4375, 2018. 2.2.3

[31] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015. 2.2.1

[32] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceed- ings of the IEEE conference on computer vision and pattern recognition*, pages580–587, 2014. 2.2.1

[33] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedingsof the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81, 2014. 2.4, 3.1, 3.2.2

[34] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive computing*, 7(4):12–18, 2008. 3.1

[35] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 2.2.2

[36] Bharath Hariharan and Ross Girshick. Low-shot visual recognition by shrinkingand hallucinating features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3018–3027, 2017. 2.2.3

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactionson pattern analysis and machine intelligence*, 37(9):1904–1916, 2015. 2.2.1

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 2.2.1

[39] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 2.2.2, 3.2.4, 3.4, 4.2.2

[40] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. The case for offload shaping. In *Proceedings of the 16th International Workshop onMobile Computing Systems and Applications*, pages 51–56, 2015. 3.2.4

[41] Wenlu Hu, Ziqiang Feng, Zhuo Chen, Jan Harkes, Padmanabhan Pillai, and Mahadev Satyanarayanan. Live synthesis of vehicle-sourced data over 4g lte. In *Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, pages 161–170, 2017. (document), 3.1, 3.2

[42] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*,pages 7310–7311, 2017. 2.2.2

[43] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a dis- tributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 125–138, 2006. 2.1

[44] Bingyi Kang, Zhuang Liu, Xin Wang, Fisher Yu, Jiashi Feng, and Trevor Darrell. Few-shot object detection via feature reweighting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8420–8429, 2019. 2.2.3

[45] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised con- trastive learning. *arXiv*

preprint arXiv:2004.11362, 2020. 2.2.3

[46] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015. 2.2.3

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classifica-tion with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012. 2.2.1

[48] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017. 2.2.4

[49] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125,2017. 2.2.1

[50] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Fo- cal loss for dense object detection. In *Proceedings of the IEEE internationalconference on computer vision*, pages 2980–2988, 2017. 2.2.1

[51] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architec- ture search. *arXiv preprint arXiv:1806.09055*, 2018. 2.2.2

[52] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 2.2.1, 3.4,4.2.2

[53] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018. 2.2.3

[54] Boris N Oreshkin, Pau Rodriguez, and Alexandre Lacoste. Tadam: Task dependent adaptive metric for improved few-shot learning. *arXiv preprint arXiv:1805.10123*, 2018. 2.2.3

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 4.5.2

[56] Juan-Manuel Perez-Rua, Xiatian Zhu, Timothy M Hospedales, and Tao Xiang. Incremental few-shot object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13846–13855, 2020.2.2.3

[57] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conf. on Computer Vision and Pattern Recognition*, pp. 2001–2010, 2017. 2.2.4

[58] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016. 4.2.2

[59] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 2.2.1

[60] Mengye Ren, Eleni Triantafillou, Sachin Ravi, Jake Snell, Kevin Swersky, Joshua B

Tenenbaum, Hugo Larochelle, and Richard S Zemel. Meta-learning for semi-supervised few-shot classification. *arXiv preprint arXiv:1803.00676*, 2018. 2.2.3

[61] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: To- wards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015. 2.2.1, 3.4

[62] Andrei A Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pas- canu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. *arXiv preprint arXiv:1807.05960*, 2018. 2.2.3

[63] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang- Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceed- ings of the IEEE conference on computer vision and pattern recognition*, pages4510–4520, 2018. 2.2.2, 3.2.4, 4.2.2

[64] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009. 2.3

[65] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The seminal role of edge-native applications. In *2019 IEEE International Con- ference on Edge Computing (EDGE)*, pages 33–40. IEEE, 2019. 2.3

[66] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. Incremental learning of object detectors without catastrophic forgetting. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3400–3409, 2017.2.2.4

[67] Jake Snell, Kevin Swersky, and Richard S Zemel. Prototypical networks for few-shot learning. *arXiv preprint arXiv:1703.05175*, 2017. 2.2.3

[68] Luis Miguel Soria, Francisco J Ortega, Juan A Alvarez-Garcia, Francisco Velasco, and Damian Fernandez-Cerero. How efficient deep-learning object detectors are? *Neurocomputing*, 385:231–257, 2020. 2.2.2

[69] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. URL https://www.ros.org. 3.2.4

[70] Bo Sun, Banghuai Li, Shengcai Cai, Ye Yuan, and Chi Zhang. Fsce: Few- shot object detection via contrastive proposal encoding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7352–7362, 2021. 2.2.3, 4.2.2, 4.3.1

[71] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WMSmeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013. 2.2.1

[72] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080*, 2016. 2.2.3

[73] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, SainingXie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. Fbnetv2: Differentiable neural

architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages12965–12974, 2020. 2.2.2, 4.2.2, 4.2.2

[74] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai,and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 152– 165, 2019. (document), 2.3, 2.1, 3.2.3, 5

[75] Xin Wang, Fisher Yu, Ruth Wang, Trevor Darrell, and Joseph E Gonzalez. Tafe-net: Task-aware feature embeddings for low shot learning. In *Proceedings ofthe IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1831–1840, 2019. 2.2.3

[76] Xin Wang, Thomas E Huang, Trevor Darrell, Joseph E Gonzalez, and Fisher Yu. Frustratingly simple few-shot object detection. *arXiv preprint arXiv:2003.06957*, 2020. 2.2.3, 3.2.4, 4.2.2

[77] Yu-Xiong Wang, Ross Girshick, Martial Hebert, and Bharath Hariharan. Low-shot learning from imaginary data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7278–7286, 2018. 2.2.3

[78] Yu-Xiong Wang, Deva Ramanan, and Martial Hebert. Meta-learning to detectrare objects. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9925–9934, 2019. 2.2.3

[79] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. ThunderSVM:A fast SVM library on GPUs and CPUs. *Journal of Machine Learning Research*, 19:797–801, 2018. 4.5.2

[80] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision andPattern Recognition*, pages 10734–10742, 2019. 2.2.2, 4.2.2

[81] Xiaopeng Yan, Ziliang Chen, Anni Xu, Xiaoxi Wang, Xiaodan Liang, and Liang Lin. Meta r-cnn: Towards general solver for instance-level low-shot learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9577–9586, 2019. 2.2.3

[82] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Haibin Lin, Zhi Zhang, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. Resnest: Split-attention networks. *arXiv preprint arXiv:2004.08955*, 2020. 2.2.2

[83] Xingui Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019. 2.2.1, 4.2.2

[84] Canbo Ye. BusEdge: Efficient Live Video Analytics for Transit Buses via Edge Computing. Master's thesis, Carnegie Mellon University technical report CMU-RI-TR-21-46, July 2021. 1