ELECENG 4OI6

Engineering Design
McMaster University

Edge Vision Engine (EVE) Final Report

Instructor:
Dr. Aleksander Jeremic

Authors:
Christoper Cabral
Brian Lee
Julian Vincent Sedillo
James Warburton

January 25, 2023

Table of Contents

1 Introduction

Message to the Reader: For a live demo and additional info please visit our documentation site: `https://engbrianlee.github.io/EVE_frontend/#/`

1.1 Problem Statement

Conventional video surveillance involves constantly running cameras, active human operators, and requires large repositories for storage of video data. These surveillance topologies need complex systems to interconnect, power, and manage them. This results in systems which are not only inflexible and difficult to modify, but over time can become computationally expensive to the point of intractability.

Traditional security techniques also require trained personnel to actively monitor a surveillance video feed in order to detect and report anomalies. These human-based detection methods are subject to a medley of disruptions that can negatively impact performance, such as: operator fatigue, operator distractions, human bias in perception, training quality, and staff rotations. The sheer amount of attention a person needs to apply to a traditional surveillance video feed in order to detect these types of events in a timely manner is painstakingly laborious as a consequence. This is especially true when considering monitoring stations with more than a handful of simultaneous video feeds per operator.

The aforementioned shortcomings of conventional video surveillance compound upon one another, resulting in typical video surveillance systems being financially expensive to purchase, operate, maintain, and upgrade. Video surveillance as it exists is therefore a privileged practice, which is available only to those with large pools of capital. This is especially in cases concerned with sophisticated distributed video surveillance networks possessing dynamic capture and detection capabilities.

Security measures that use integrated intelligence or Machine Learning (ML) to autonomously detect, alert, and capture events of interest during surveillance are still a novelty in the security space. For example, security measures can be implemented at the software level to provide alerts when important events are observed, such as an unidentified person entering a restricted area or field of view monitored by a camera. This lag in adoptation of cutting edge methods due to the entrenchment of the sector in its previous tactics needs to be corrected. The result should be an increase in capability and decrease in cost, which should allow for greater adoptation of video surveillance systems as barriers for entry are removed.

1.2 Motivation

The ambition behind the *Edge Vision Engine (EVE)* project is to develop and showcase an efficient, semi-autonomous, and affordable alternative to commercial surveillance products, lowering the barrier to entry for a potential user. The resultant product is a platform that seamlessly automates the surveillance process and provides real-time alerts and analysis of detected events, allowing attention to be directed towards more pertinent video feeds as key events occur within those monitored spaces.

As a forerunner to an abundance of solutions that revolve around the Internet of Things (IoT), EVE lays the groundwork for redefining and re-engineering event monitoring and surveillance, the smart home, and the smart city. Most importantly, EVE enables the average household to take advantage of state of the art Artificial Intelligence (AI) and provide an affordable and autonomous surveillance and monitoring system that is low power, compact,

scalable, and accessible anywhere on the web. Commercializing the structure developed for supporting EVE is a first step in improving access and affordability to video network surveillance systems across society.

2 Objectives

The development of the project was divided into four sequential stages: Bronze, Silver, Gold, and Platinum. Each of these developmental stages are characterized by an increasing engineering effort applied to implement technical features that improved it over its predecessor. Through following an iterative engineering design approach, each sequential prototype became an increasingly complex system for addressing the issues laid out within the problem statement. Starting with the Bronze target, functionality would be incrementally added until terminating at the most capable target, Platinum.

2.1 Initial Targets

During the initial stages of its creation, the primary objective designated to EVE was to monitor the occupancy of an event space, such as a room, where a party, procession, or meeting would be taking place. Moreover, the identity of all people present in the event space was to be observed, using a machine learning model that registered the person as they enter and detected the moment they leave using their facial feature vector representing their unique physical characteristics corresponding to their human face. The unique features of each planned developmental stage are the following:

- **Bronze:** The IoT device (Rasp. Pi) is equipped with IR or proximity sensors to detect a human passing through a constrained single file entrance way. The device classifies the individual passing through as an entry or exit, then sends this information to the Amazon Web Services (AWS) Cloud. A number representing the amount of people present either increments or decrements depending on the device's decision. A web application reads this state data from AWS, and continually displays and updates this number accordingly.

- **Silver:** A camera sensor is attached to the IoT device, and captures an image showing the face of the individuals who enter the event space. The image is sent to AWS and it is appended to an array of images displayed on the web application.

- **Gold:** The camera sensor functions concurrently with the proximity sensors to discern whether a person is entering or leaving the event space. Images are captured at the time of entrance and exit and sent to AWS. A Machine Learning (ML) model also running on AWS actively parses the captured images (e.g. from a database) and obtains a feature vector pertaining to the unique characteristics of the person.

  If the person is entering, this feature vector is added to a collection for all the active occupants in the room. If exiting, the feature vector is compared to the feature vector representations of all the occupants in the room. The ML algorithm calculates the relative distances between the captured feature vector and those already in the collection, and removes the feature vector that has the closest similarity (i.e the "distance"). On the web application, the image is either added to the array when the person is entering, or its matching image is removed from the array when the person is exiting.

- **Platinum:** The machine learning model that detects the presence of a person using the camera feed is compressed and run natively on the IoT device. This requires the machine learning model to be efficiently run on significantly reduced resources, and

perform inference at near real-time. Further iterative features that may be added are implemented at this stage, if time and resources allow.

2.2 Updated Targets

Over the course of development, the milestone objectives radically changed for the better through further learning and reflection on what could make EVE more effective in addressing the problem statement. Notably, the discovery of cutting-edge compressed machine learning models that efficiently determined if a human is in frame opened up a plethora of new possibilities. Additionally, naive assumptions made in the assessment of the potential solution space were updated as research and development occurred. Major changes to the project targets are as follows:

- Removed counting ingress and egress via infrared sensor or proximity sensor due to the profound restriction it imposed on the demonstrability of the device in a live environment. These separate sensors would also have greatly limited the flexibility of the final product in terms of its use cases.
- Research of state of the art Deep Learning (DL) models yielded a Visual Wake Word (VWW) model optimized for embedded systems. The VWW model boasts fast performance, a low memory footprint, and a straightforward, generalized integration capability.
- Research of the chosen web-service host, Amazon Web Services (AWS), revealed a world of in-built services, packages, and tools to handle and manage tasks that would have otherwise needed to be built from the ground up. Beyond the task of integrating these separate resources together, this exponentially increased the features that could be demonstrated during the project lifetime.

By the time that the project had come to a head, decisions had to be made regarding existing milestones when contrasted with the new research, tools (i.e. VWW model), capability of cloud services, and desired project outcome and scalability. The targets that represent the project's developmental stages had now become the following:

- **Bronze:** Video camera interfaced with a computer. Compressed state of the art VWW model that can detect if a human is in frame is run on the device. Video feed can be captured and shown locally on the device, with data output showing if a human is detected in frame.
- **Silver:** A video camera is attached to an IoT low profile device (Rasp. Pi). Compressed state of the art VWW model that can detect if a human is in frame is run on the device. Video feed and human detection can be captured and shown locally on the device. Hardware configuration, device-camera interfacing is optimized for the chosen technology at this time.
- **Gold:** Video frame data is uploaded directly to AWS cloud service network. Video feed can be captured and shown on a web application remotely.
- **Platinum:** Additional services are integrated as back-end (off-device) computations, such as: face classification, storage, and video streaming. Services are hosted and accessible through an easily understandable human machine interface (HMI) for users to interact with is built as a web application. A data path is designed and built to incorporate

several cloud-based services to allow for face classification (i.e. identity of individual), and bounding boxes to be drawn on faces captured in real-time from VWW model running on IoT device.

It was understood from a very early point in the development of our project that the milestones were not immutable. As continual research, development, and prototyping efforts progressed, logistic and resource constraints were constantly being modified. New constraints may be introduced, or in some cases removed as decisions were made, goals and objectives changed, or new solutions discovered. Communicating the initial targets that were decided upon at the onset of the project and comparing them to the final updated targets (which met the same requirements as the Platinum standard) is an effort to stress the fluidity of iterative project development. Although it may be common in more traditional fields of engineering that are solving well-defined problems, it very rare that that software-based enterprise follows a strict "waterfall" structure. The iterative process of engineering prototyping constantly yields new insights and perspectives, resulting in a consistently evolving product until each component is well-understood, and features have been optimized to meet standards and objectives.

3 Approach

3.1 Internet of Things (IoT)

IoT will provide a scalable paradigm to connect multiple devices to the cloud. Our solution will use Amazon Web Services (AWS) IoT support services to create the following components:

- *Device Gateway*: Enables devices to securely and efficiently communicate with the Cloud over the internet.
- *Rules Engine*: Provides message processing, automation, and integration with other services (i.e. AWS-based services and HTTPS web application).
- *Device Registry*: Provide a way to register and identity unique devices.
- *Device Shadow*: Provide a means to to store and retrieve current state information for a device connected to AWS IoT. Device states are stored as a JSON object in the Device Shadow.
- *Message Broker*: Manages MQTT protocol for IoT device state management, and passes messages according to a publish-subscription model.

3.2 IoT and Cloud (AWS) Communication and Device State

In order to allow for secure communications between the IoT Devices and the Web Application for the users to see data displayed, as well as interact with the device, we will be using AWS IoT Core that provides an internet gateway that simplifies the direct connections between the IoT Devices and the Cloud. The AWS IoT core enables each IoT Device to be registered with a unique "thing" ID, whichcorresponds to a 1-to-1 mapping for the device's state (called Device Shadow in AWS IoT Core), and the Cloud services that it will be using (e.g. AWS S3 Bucket, and AWS Kinesis Stream). The gateway and unique identifies allows for an easily scalable system that reduces complexity of establishing secure internet communication.

The two major components of the AWS IoT Core that we leverage to enable secure and reliable IoT communications, and track IoT Device state are:

1. MQTT Protocol
2. AWS Device Shadow

3.2.1 MQTT Protocol

The IoT Device (Rasp Pi) and AWS primarily communicate through the Message Queue Telemetry Transport (MQTT) protocol, which is an ISO standard which facilitates a publish-subscribe network protocol topology. Each IoT device is able to read and write messages, with AWS acting as the *broker* for the system. A broker under MQTT acts as the central repository for messages in the network. In regards to other communication protocols, the broker can be thought of as the master and all connected IoT devices as slaves (although this analogy is only for illustration and is not entirely correct).

The publish-subscribe topology is facilitated through the system broker. *Topics* are published to the broker, or sometimes by the broker itself. These topics are usually self-described in their subject line as a method of addressing. Topics are queried through a *subscription request* in order to read the attached message. It is therefore imperative that the broker is regarded as the single*source of truth* for the system, and by necessity needs near constant up-time so that publish requests are not dropped.

Since AWS handles the role of broker in the MQTT protocol for EVE, the cloud nature of AWS is exploited to achieve the requisite stability to support the system. By leaning on AWS Cloud Services stability, MQTT protocol allows for safe asynchronous communication between interconnected devices. This means that when simultaneous devices are interfaced with AWS, cross-talk is impossible, device level network problems or message-drops can be corrected by simple re-publish or re-subscribe (Quality of Service, QoS) routines, and indeterminate states are rendered impossible.

3.2.2 AWS Device Shadow

AWS defines a *shadow* as a JSON document that is used to store and retrieve the current state of information for *any* device connected to AWS. Each instance of a device shadow is considered a separate *shadow* object. The Device Shadow service acts as an intermediate to maintain a shadow for each device that is connected to the AWS IoT Core, and can manage large numbers of shadows simultaneously. The device shadow is readable or writeable over MQTT. This means that the device state can be queried or manually set via manipulation of the shadow. For example, toggling the camera on and off, or executing or reading the output of a subroutine can be handled by manipulating the device shadow. Each device has a shadow attributed to its inherent *device name*, which can be changed dynamically if required.

3.3 On the Edge AI (Visual Wake Word)

On the Edge AI will allow devices to perform complex pattern recognition without having to connect to more powerful servers or ancillary computing units. Computing on the edge is necessary to allow for low latency connections, meet low bandwidth requirements, reduce energy consumption and perform minimal computation server-side. The deep learning compression algorithms discussed in Section 4.1.1 will be used to fit the state-of-the-art recognition models on a Raspberry Pi.

The effect of utilizing On the Edge AI will be a class of IoT devices that can prepare or infer necessary state changes by conducting lower-frequency AI subroutines. For example, EVE runs a compressed model that allows the device to determine whether a human is in-frame.

This means that the device only needs to send data (and thus power taxing wireless network transmission circuits and amplifiers) when it *already knows* if an event of interest has occurred. By finding and running these subroutines at the frequency floor to achieve required performance, the power, bandwidth, and shared server-side computational resources are minimized.

3.4 Cloud (Serverless) Infrastructure

Our software solutions run completely in the cloud, eliminating the concern of scalability, maintenance, and upfront development costs of key services such as databases. Furthermore, our solution is able to be accessed remotely from anywhere with an internet connection.

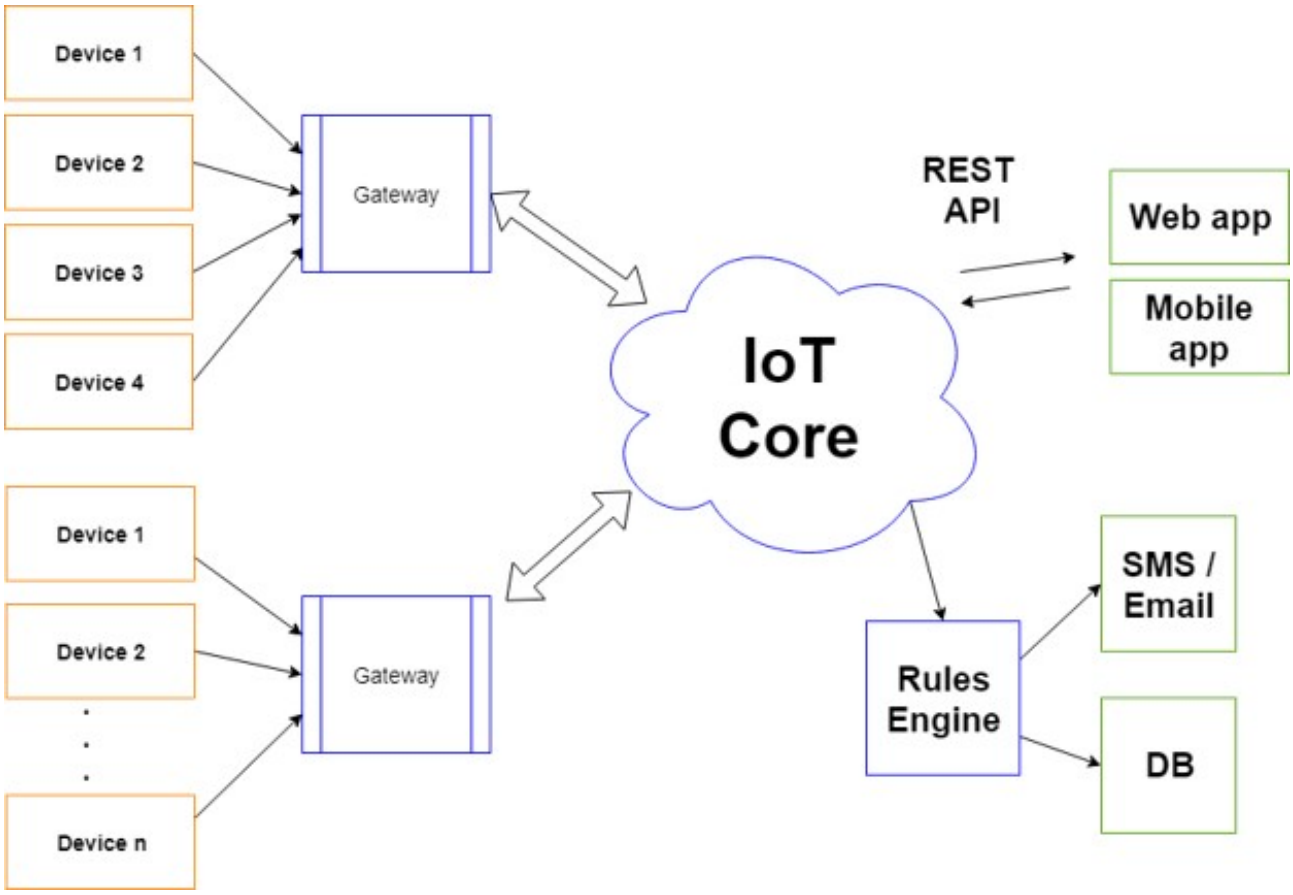

Figure 1: Original Flowchart illustrating generic IoT framework combining devices with server-less cloud architecture and data visualization on a web application.

3.5 State Estimation, Detection, and Decision-Making

For any given Field of View (FOV) or specific camera frame, EVE determines if there is a person present within the frame, and if so, at what exact time the person was detected. This is handled by a compressed neural network model running on the Raspberry Pi and functions as a Visual Wake Word (VWW). What this allows is for the camera to be refreshing at a lower frame rate, and for other power-intensive subroutines (such as network broadcast) to be turned off whilst the camera does not detect a human presence. When the algorithm detects the *transition* from having no human in frame (state 0) to having a human in frame (state 1), the device immediately captures the frame and publishes it to the cloud. This type of detection is called *Positive Edge* detection because transition from state 0 to 1 occurs along a *positive* gradient or *edge* between states.

Once the image of a detected human is published to the Cloud Server, it is fed through a face detection algorithm within AWS Rekognition to pre-process the image. First, the image is convolved with a sliding window as a *first pass*, in order to determine the best suited bounding box around potential face features. This is done using *intersection over union* and *non-max suppression* techniques. Whilst the most probable bounding box is created, the same process is iterated for the related *facial landmark* coordinates simultaneously. This overall process for building the bounding box around the subject face and labelling the enclosed feature vectors is called the You Only Look Once (YOLO) algorithm, due to its non-recursive structure. The result of the output of this algorithm is that the incoming image is now described by an appropriate bounding box around the face and the inherent feature vectors of the associated face are quantified.

As a result of completing the aforementioned process, AWS IoT will send a notification to the user over the web application and update the state of the device. The user is then allowed to take specific actions based upon the current state of the system. The bounded face with mapped feature vectors is presented, and if the face does not map sufficiently close to a face within a predefined *Collection* of feature vectors, then it is given an *Unknown* label and stored as such within the S3 Data Bucket. Through the web application, the user may access this image or a series of images with ascribed feature vectors and label them manually as a Collection, either preexisting or new. The result is that if the same person is to again enter the field of view of EVE, it will recognize that the person is not an Unknown, and in fact has been seen and classified beforehand.

The high level output of the above process flow is a system which utilizes Deep Learning models to do complex analysis on frames of a video stream that is typically left to a human operator or observer. Although this process is automated, it can also be augmented through a web application to verify compliance or accuracy and further discern people by their associated names or titles. The act of deciding if a person is *Unknown* or already *Indexed* (Known) are handled automatically. These tasks are completed an order of magnitude faster than a human operator could perform them and are appropriately and automatically stored and organized within *Collections* in the Cloud. Accessing Collections, defining Labels, and reviewing incoming video streams are made easy through the web application user interface.

3.6 Web Application

Our web application establishes a convenient platform for communication between the users and the cloud. User control over the state of the IoT devices (i.e. whether the device is steaming or running the VWW) as well as the control over the collected portrait images is maintained here. Moreover, the web application allows for the visualization of the data sent to the cloud from the IoT devices, such as portrait images, a live video stream, and notifications that a person is detected by the VWW.

3.6.1 React Framework

The web application was created using the ReactJS library, a framework built and maintained by Facebook. This framework was chosen for the front-end development of EVE given the rapid gains in popularity it has made since its release to the public in 2015, and its distinct component-based approach it employs in its web interfaces.

3.6.2 Material UI Library

The embellishment of the web interface was achieved with the use of the open-source Material-UI library which features React components that implement the popular Google Material Design. This open-source library was crowd sourced not long after the public release of ReactJS, introducing new React components that integrate Google's signature Material Design. The considerations behind choosing this library to style our application involved the decision to introduce users to an interface that remains familiar as well as choosing to avoid spending an excessive amount of time to implement styles in creating custom React UI elements, and manually implementing Cascading Style Sheets (CSS).

4 System Overview

The overall EVE system can be broken down into three major subsections:

1. Edge Device (IoT)
2. Cloud Services (AWS)
3. Web Application (React)

These sub-systems each specialize in particular functions that modularizes EVE in a way that enables it to be both flexible and scalable.

4.1 Edge Device

The Edge (IoT) device boasts many features that enable it to be a ideal platform to measure and acquire human-specific information regarding an environment of interest. Each edge device is inexpensive and widely available (Raspberry Pi hardware), has low power requirements, is non-invasive due to its small form factor, is and connected to the cloud wirelessly. The camera connected to the edge device is non-unique, and can be swapped out for a variety of comparable alternatives, each depending upon the desired resolution and frame-rate requirements of the user.

4.1.1 Visual Wake Words and Deep Learning Compression

The true power behind the IoT devices comes from the highly compressed deep learning model that runs natively on the edge device with low memory footprint, compact size, and rapid inference rate. Support for the concept of *Interference on the Edge* was supported from several IEEE papers reporting the effectiveness of deep learning compression methods used on edge computing devices.[8][4][1] These techniques are rising in popularity due to their efficient resource allocation compared to traditional deep learning techniques. This cost-effectiveness is realized through reductions in energy consumption, space, computational needs, and high-throughput communication. An important fact to note is that the deep learning models are able to the run on the edge computing devices only after they have been fully trained. Training deep neural networks is still heavily constrained to high performance computing clusters with massive parallel computation capabilities.

Much of the research that enabled efficient inference on the Raspberry Pi comes from Dr. Song Han's work, particularly within his seminal paper, *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*,[6] which presents algorithms that reduce model size by up to 500% without affecting performance.

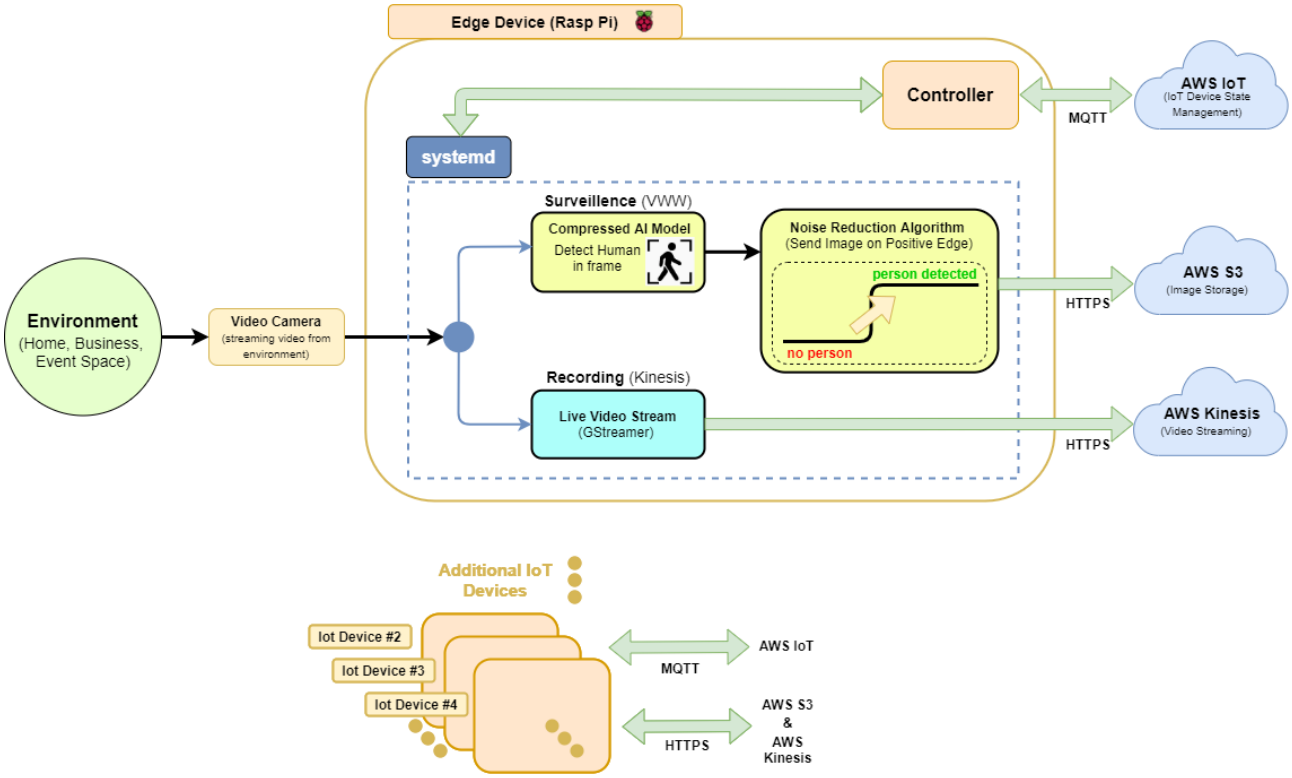

Figure 2: System Diagram of the IoT Device (Edge) Sub-System, illustrating the various state the data-flow from Environment, where the service (daemon) being run is controlled via the systemd Linux service manager.

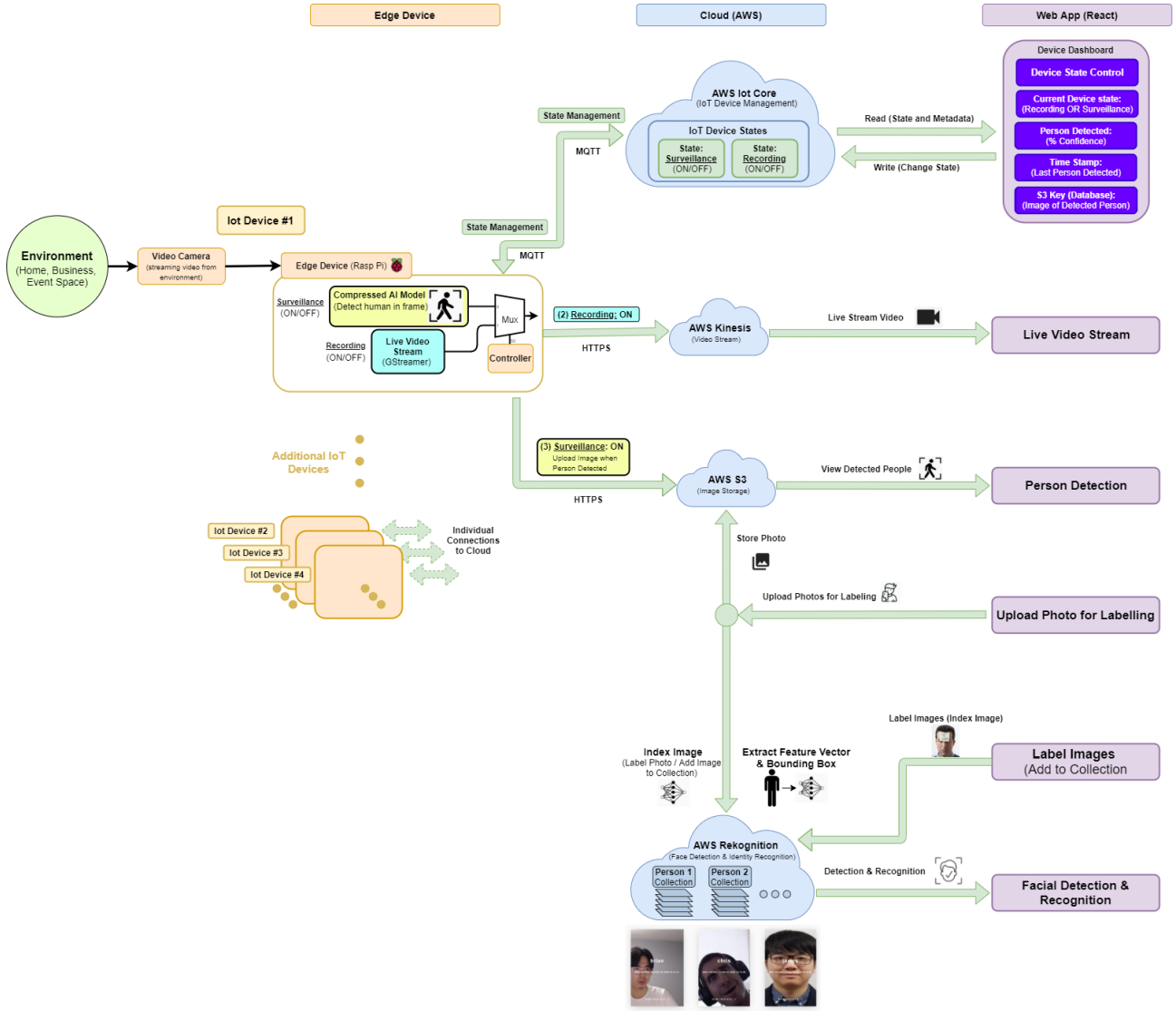

Figure 3: System Diagram of the Edge Vision Engine (EVE) illustrating the data-flow from Environment to IoT Edge Device, AWS Cloud Infrastructure and Services, and React Web Application.

These techniques, among others, are currently being used throughout the industry to enable the use of large deep learning models on mobile and embedded devices and is still an active area of research.

An implementable version of a compressed deep learning model tuned for human recognition was submitted by Dr. Song Han's research group during the Visual Wake Word (VWW) competition at the Conference for Computer Vision and Pattern Recognition (CVPR) in the summer of 2019. The concept of a Visual Wake Word is constrasted with well understood Verbal or Audio Wake Words that are in use everyday, such as: "OK Google", "Hello Siri", or "Alexa", which awakens the device from a low power state, and enables the user to input more complex commands. Dr. Han's group won the VWW competition with a computer vision model that is able to detect the existence of a human (or humans) in a frame using their highly compressed deep learning model, so that after binary detection (human or not), more computationally expensive operations can be performed. This framework forms the basis of how our IoT device operates. The results acquired from the VWW that runs on the Raspberry Pi is transmitted to AWS for further processing, as well as alerts the user on the web app for them to take addition actions.

In line with Dr. Han's research[6] which utilizes *Pruning* and *Quantization* to compress deep learning models, however these tasks are perform manually (or in a customized manner), and must be further refined and fine-tuned to a particular hardware platform. The question arises, what if there was a way to automatically perform these tasks given a target task or optimization criteria (i.e. memory, energy/power, latency constraints) and physical hardware (e.g. Raspberry Pi, Arduino, microcontroller, mobile phone hardware platforms). This very problem has been addressed through several research developments for both automating the compression as well as hardware optimization, both as a result of Dr. Han's research group. Two major developments were made: AutoML for Model Compression (AMC) which was fine-tuned for Mobile Devices[7], as well as Hardware-Aware Automated Quantiziation (HAQ), with mixed Precision[5]. The latter of the two techniques proved to be more successful in automating quantization for a variety of hardware platforms. Both of these techniques utilize a goal directed form of machine learning, called deep reinforcement learning to optimize the desired targets.

Another question remains: "How do we address that fact that each deep learning model will perform differently on different hardware?"

This performance depends upon the deep learning architecture, as well as its computational and resource requirements. When performing training, what if we could automatically determine which model architecture would be best suited for your hardware platform. Once again, we can utilize a deep reinforcement learning technique, called Neural Architecture Search (NAS), which automatically designs effective neural network architectures, however the computational efficiency of NAS becomes prohibitive to search architectures on large-scale tasks. This is primarily due to the fact that NAS was formulated as a reinforcement learning problem, which are usually very resource expensive and inefficient. This is often overcome by providing a proxy-task that is easier to compute, but may not be directly comparable to the desired task that the architecture should be optimized to solve.

Typically NAS involves two phases, model search, and model tuning. In the search phase, best architectures after limited training are selected. In model tuning, the selected architectures are trained fully, and a further refined choice can be made. Thankfully, Dr. Han's group made improvements to the NAS pipeline, through introducing ProxylessNAS[2] that allowed for efficient deep learning architecture determination, without sacrificing optimization through a proxy-task. Overall, the current state of the art is done by using shared computation and moving away from controller and reinforcement learning approaches

to favour oneshot or evolutionary methods. In addition to ProxylesssNAS, Differentiable Architecture Search (DARTS) has been proposed, which owes its efficiency to constantly relaxing the architecture representation, allowing for efficient search of a given architecture using gradient descent.[3]

Finally, the resulting compressed, efficient, low memory footprint VWW model that runs on the Raspberry Pi was created by a deep learning pipeline that combines two major steps:

1. ProxylessNAS: Automation of selecting optimal deep learning architecture that is fast/ efficient for a hardware platform.

2. HAQ: Automated compression of deep learning model using automatic quanitization with mixed precision.

EVE incorporates the predictive capabilities of a highly tuned human-detecting deep learning model, and combines it with the results of intelligent model compression that not only shrinks the size of the deep learning model, but the model itself has been optimized for the Raspberry Pi hardware, as a part of the VWW competition.

4.1.2 Human Surveillance and Recording

With any given environment that a user would like to monitor if a human is likely to enter, there are two operational states that the Raspberry Pi can operate in: Surveillance (VWW) and Recording (Kinesis Video Stream). These separate states need to operate seemlessly within the device, since they are both attempting to access the same shared resource (video camera) and also be able to be controlled via a user through a web-app. The solution to this is to utilize the built-in Linux service (or daemon) manager called *systemd*. Calls to the service manager is done via the controller, which responds to device state changes that are initiated from the AWS IoT Core, and performs the appropriate action to satisfy the state change. As previously mentioned in the Approach section, the AWS IoT Core possesses a state manager for each device, called the Device Shadow, which is able to send/receive messages to and from the IoT device using the MQTT protocol.

There are three possible states that the Raspberry Pi can take:

1. All daemons/services are OFF

2. Surveillance (VWW) is ON, and Recording (Kinesis) is OFF

3. Recording (Kinesis) is ON, and Surveillance (VWW) is OFF

Both Surveillance and Recording cannot be ON at the same time since only one daemon/service can access the camera at a time. The controller has internal logic to ensure that this cannot occur. When the controller first starts, it will delete (reset) the current device state, and update it with the current state of the device by checking what services are currently running. This eliminates the possibility that the controller is out of sync with the actual state of the Raspberry Pi. Every time that the state of the IoT Device is changed (as well as when the controller is first initiated), it will publish to AWS IoT regarding the current state of the Raspberry Pi. Whenever a change to the state of the Raspberry Pi occurs, which usually will happen from a user driven action from the web app, the Raspberry Pi will send the current state to the AWS IoT Core via the MQTT protocol, and AWS IoT will update that given device's *Device Shadow*.

It is pertinent to clarify that the web app only communicates directly with AWS IoT Core, the web app does not communicate directly with the IoT Device. As a result, when the user requests that the state of the device be changed, AWS IoT Core makes a request to the IoT device stating that it *desires* the IoT device to be in a different state. The IoT device will then try to satisfy the request (respecting the fact that Surveillance and Recording operate in a mutually exclusive manner), and then if the IoT device is able to accomplish this task, it will respond with an update to the AWS IoT Core indicating what the *reported* or confirmed state actually is. This ensures that the request made by the AWS IoT Core has been successfully enacted. Moreover, this allows for the AWS IoT Core to act as a source of truth for the React Web App.

4.1.3 Noise Correction and Positive Edge

In order to reduce the amount of noise produced from the camera auto-focusing, and thus momentarily blurring the input image, causing the deep learning VWW model to be unable to process the human in frame, a simple de-noising algorith was produced. Instead of setting the focus manually and potentially causing other cascading issues, a simple detection heuristic was developed in software to correct the noise. Three parameters are utilized to remove false negatives in the VWW model. These parameters can easily be adjusted and/or removed depending on any modifications made to the physical hardware attached to the Raspberry Pi. The flexibility of the software-based solution allows for the IoT Device to interface with a variety of sensors that may better suit the a given environment, power-constraints, cost considerations, frame-rate, or other desired factors.

Our noise reduction algorithm parameters are as follows:

1. Notification Interval
2. Minimum Detection Confidence
3. Minimum Interval Percentage

The *Notification Interval* pertains to how many frames are in a given interval between notifications send to the AWS IoT Core. This is user defined, and for demonstration purposes, this was set to 1 second, thus the notification interval was set to 15, since the camera's effective frame rate was 15fps. The *Minimum Detection Confidence* is the minimum percent confidence returned by the VWW model if a person is detected in frame, set to 80% (or 0.80), as a general heuristic. Finally, the *Minimum Interval Percentage* is the minimum number of frames in a given *notification internal* that show that a human was detected in frame, which was set to 30% (or 0.30). Therefore, at least 30% of the frames in a given notification interval must have had a percent confidence of 80% or higher for the IoT Device to alert AWS IoT Core (and consequently the Web App) that a human was detected in frame.

In order to coincide with the IoT paradigm of devices that have requirements that are: low power, low bandwidth, and efficient memory and computations, we only transmit if a human was detected in frame on a rising or positive edge. That is, we have decided that if there is a detection, we only send a photo to AWS S3 Bucket (and likewise displayed on the Web App), when the VWW model goes from *Not Detected* to *Person Detected*. This accomplishes the saving of bandwidth, computational time, and by extension, power consumption of the Raspberry Pi.

4.2 Cloud (AWS)

Integrating the IoT Devices and Web Application is accomplished via Amazon Web Services (AWS). The typical barrer to entry for households, consumers, and small businesses from having robust, affordable, scalable, and flexible surveillance and video systems are the data storage, service integration, maintenance, and up-front capital cost of hardware and servers. Cloud services offered through AWS solves not only these problems, but provides built-in scalable features, as well as a robust selection of online services that are affordable pay-as-you-go (or pay what you use), that have convenient application program interfaces (APIs) and libraries to easily integrate your hardware and software solutions. All of these services and solutions are highly customizable, completely secure end to end. Therefore all the data that is transmitted from the IoT device and transmitted to the Web App is encrypted and secure on the AWS servers. All long as each external node (i.e. Raspberry and React Web App) are secure, and all AWS API calls are done on the external nodes without taking security shortcuts, then user data integrity and security will be held to a high standard. AWS cloud services cost far less, both in the up-front capital cost, as well as long term maintenance, data integrity, system upgrades, flexibility (both in technology stack, and physical hardware connections), rental space, and training of technical staff to maintain the systems. AWS simplifies all of these features, and makes it possible for EVE to be not only affordable to the average household, but easily scaled for business and enterprise solutions, both for short and long term applications.

All the services we utilize on AWS fall under the Free-Tier promotion, which provides an account to access of the most common AWS services with very generous limits at no cost for 12 months. The major services that we integrated into EVE were:

1. AWS IoT Core
2. AWS S3 Buckets
3. AWS Rekognition
4. AWS Kinesis Video Streams

We also experimented with Functions as a Service (FaaS), or Serverless Cloud architecture by trying out: AWS GreenGrass, AWS Lambdas, combined with AWS CloudFormation, but we found that these we more complex than alternative solutions, and may incur a cost based upon usage.

For a given AWS Free-Tier account, we utilized AWS IoT Core to register several Raspberry Pi devices, which are referred to as IoT "Things". The AWS Software Development Kit (SDK) for Python had to be installed on every registered Raspberry Pi for it to establish a secure connection with AWS IoT Core and other AWS services. In order to enable secure data transfer and communication between each IoT Device and AWS, we had to generate Certificate Authority (CA) on AWS that provided access permission for each IoT Device. Furthermore, AWS Access Key and AWS Secret Key pairs had to be generated for the web application to interface with AWS. Secure communications from each IoT device to different AWS services utilizes the AWS IoT SDK, as well as Boto3 which allows for direct calls to other services such as AWS Rekognition, AWS S3 Bucket, and AWS Kinesis Video Stream. During prototyping for each of the services, we also installed AWS Command Line Interface (CLI), which allowed for direct function calls to AWS services without having to write a formal script. This allowed for flexible changes and modifications to different AWS

resources as we build, tested and integrated the AWS services with the IoT devices and React Web App.

Overall, each of the IoT Devices have a direct connection to AWS Iot Core using AWS IoT SDK (written in Python) which follows the MQTT protocol. The Raspberry Pi controller will establish a connection with AWS IoT Core, and receive commands and send updates to and from AWS IoT Core. Each IoT Device has a specific representation of its active state in AWS IoT Core, which is referred to as its Device Shadow. The Device Shadow is a JSON document that is used to store and retrieve current state information for a device. We can use the Device Shadow to get and set the state of a device over MQTT or HTTP (but we utilized a MQTT connection following the pub/sub model interacting with topics). The Device Shadow service uses MQTT topics to facilitate communication between applications and devices, which fall into the operations of: `/shadow/update`, `/shadow/get`, and finally `/shadow/delete`. Each operation has a response, that is either `accepted`, or `rejected`. For instance, we may receive the response for a `/shadow/update` as either: `/shadow/update/accepted` or `/shadow/update/rejected`. Devices are able to publish to a topic, when they wish to provide information to all the subscribers listening to that topic. When a change has occurred between the current state, and an updated state, this generates a Shadow delta, which confirms that the state has changes, and then can be transmitted to each subscriber on that given topic.

Each update that is recorded in AWS IoT Core, can then be transmitted to the React Web App after a `/shadow/update/accepted` has occurred. By establishing the AWS IoT Core as the "Source of Truth" for each Device State (via AWS Device Shadow), multiple simultaneous React Web Apps will all be synchronized to the same device states regardless of when the web apps were started, or the actions taken by each web app. Thus AWS IoT Core simplifies state management in a way that allows for scalability for both IoT Devices, and React Web Applications.

The remaining calls to AWS resources such as AWS S3 Buckets, AWS Kinesis and AWS Rekognition are done via HTTPS since these operations involve larger bandwidth requirements (such as transmitting an image, or video data stream). Each Iot Device is able to directly make calls to AWS S3 Bucket or AWS Kinesis using either the Boto3 API to AWS (for sending images to S3 Bucket), or AWS CLI (for streaming video data to AWS Kinesis via GStreamer application). A separate AWS S3 Bucket and AWS Kinesis Video Stream is allocated/built in AWS for each Iot Device. The cost for these resources are primarily dependent on their frequency of use. IoT Devices that are not consuming resources or actively interacting with AWS services will not incur additional costs, allowing for intelligent scalability without waste.

Lastly, interactions between AWS S3 Buckets, and AWS Rekognition are initiated by the React Web Applications using existing AWS API calls. The AWS security credentials are stored on the React Web Server, which allows for seamless access to AWS resources, and call internal AWS functions such as: `compare_faces`, `create_collection`, and `index_faces`. Each face that is labelled, consequently has their feature vector extracted by being indexed and added to a AWS Rekognition collection. In order for this indexed face to be referred to the image in the S3 bucket that created it, when each image is indexed, the `external_image_id` of that indexed face will be set to the unique link/identifier in the S3 Bucket that the indexed face came from. This allows for each image that is obtained from the VWW model (Surveillance state on the Raspberry Pi) to not only be classified (by comparing feature vectors), but enables the Web App to reference the source images that were used to generate the feature vectors that are in the collection that had the highest similarity.

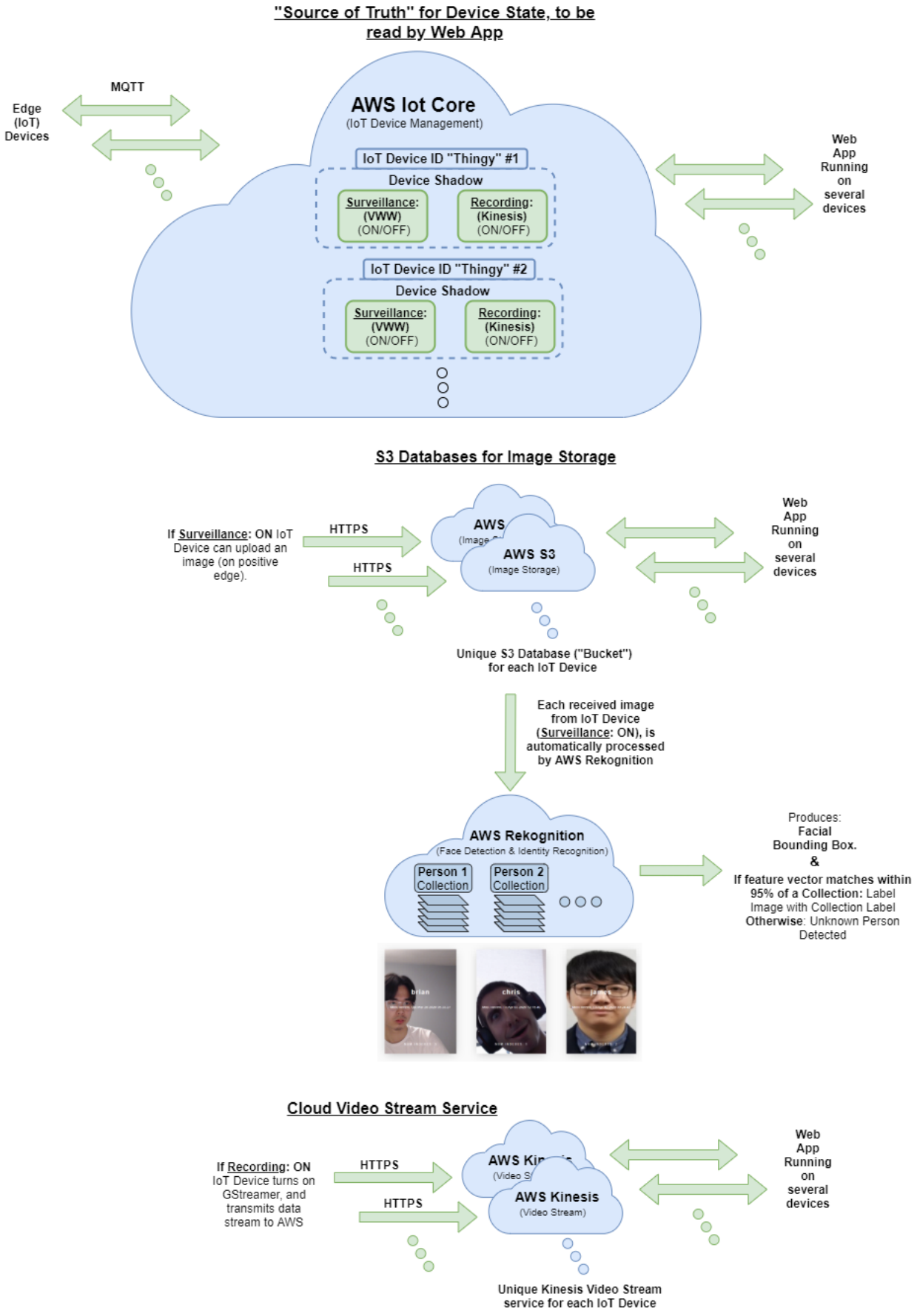

Figure 4: System Diagram of the Cloud Services (AWS) illustrating internal data flow, and interfaces with IoT Devices and Web App.

4.2.1 Face Detection

Although the source code and specific deep learning models are not explicitly disclosed for AWS Rekognition, the current cutting edge deep learning face detection methodologies are well understood. It is highly likely that AWS Rekognition is utilizing various implementations of what is understood in deep learning literature, and leveraging their massive image data sets and computing clusters to train highly accurate deep learning models. First off, to solve face detection a more general problem that needs to be solved in object detection, as a face is just a certain class of object that we want to detect. For object detection, we can perform classification, where given an image, we determine its class (e.g. dog, cat, person, human face, etc). We can compound classification with localization, where given an image, we determine a *single* class, and place a bounding box around that object corresponding to the location the object is present. While detection, in general, is when given an image, determine *multiple* classes and bounding boxes around each object of interest.

Typically we start with localization of objects, and then build up to detection. When doing classification with `n` classes we get a `n + 1` dimensional softmax outputs that corresponds to the confidence of each class being present in the image. The extra output comes from the implicit *"none of the above"* class. To do localization we add 4 additional outputs, `b_x, b_y, b_w, b_h` which corresponds to a bounding box around the detected object. We use the convention that (0, 0) corresponds to the upper left corner of the image and `b_x, b_y` will be normalized coordinates between `[0, 1]`. Similarly `b_w, b_h` are normalized coordinate between `[0, 1]` that is the width and height of the box. Thus our output for classification with localization is `classification_with_localization = [p_c, b_x, b_y, b_w, b_h, c_0, c_1, c_2]`, where `p_c` corresponds to the probability that there is a detected class (i.e. if the object is class 0, 1, 2). Low confidence means that the object is the *"none of the above"* class.

4.3 Web Application

The Web App is the interface that allows the user to interact with our design. Much of the details have been explained above and thus this section will simply show the actual UI of the Web App rather then dive into the implementation details. For a live demo and additional info please visit our documentation site: `https://engbrianlee.github.io/EVE_frontend/#/`

Please see Figures 5, 6, 7, and 8 for the FrontEnd of EVE, showing different user interfaces provided by the React Web App.

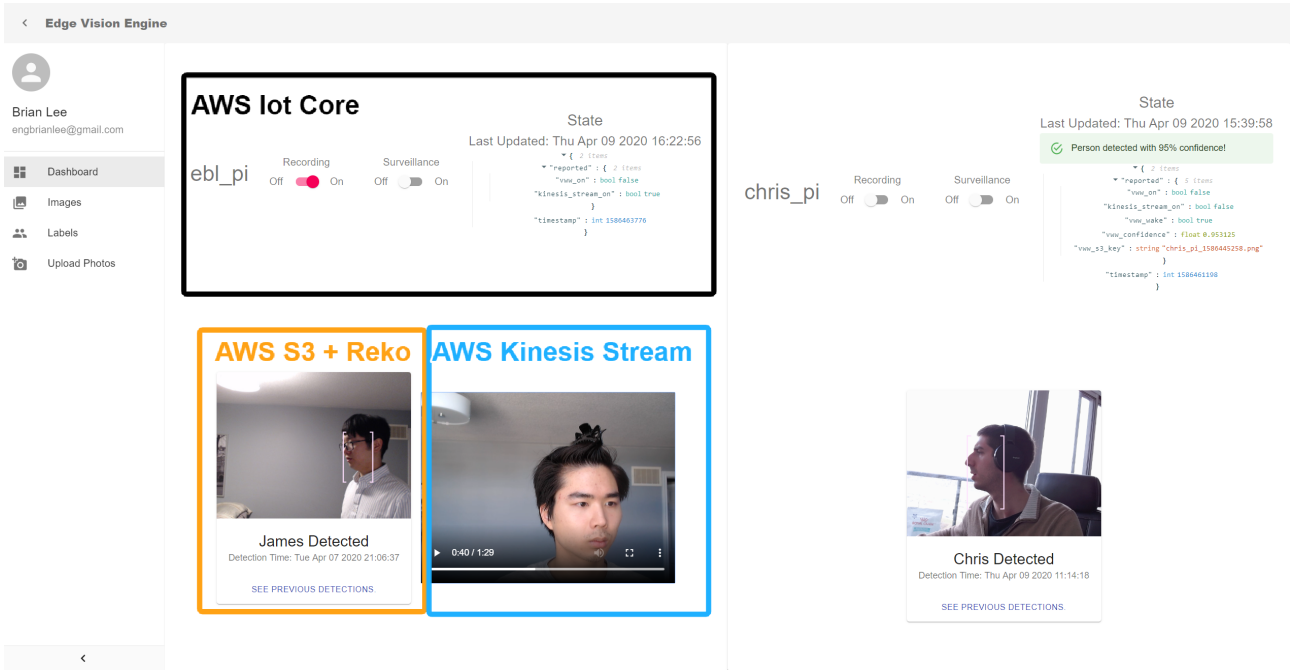

Figure 5: Web App Frontend: User interface allowing them to control IoT Device, and read data from different AWS services Surveillance (AWS S3 + Reko) and Recording (Kinesis Stream)

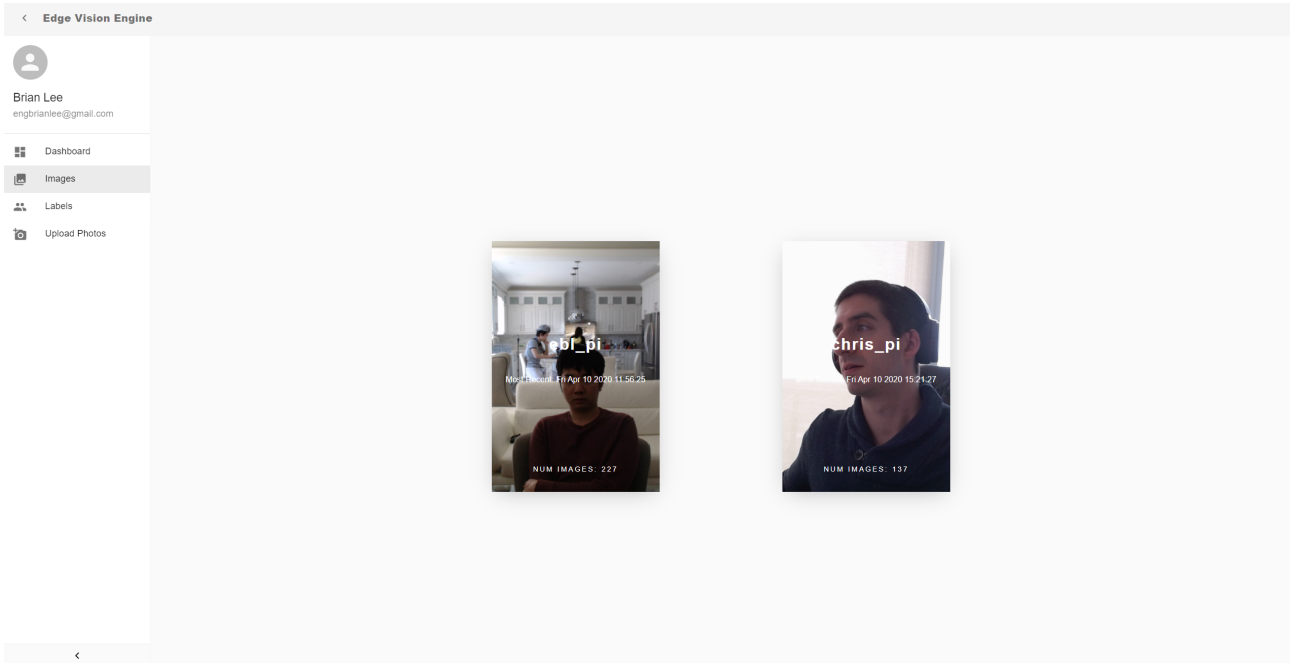

Figure 6: Web App Frontend: Album listing of edge devices

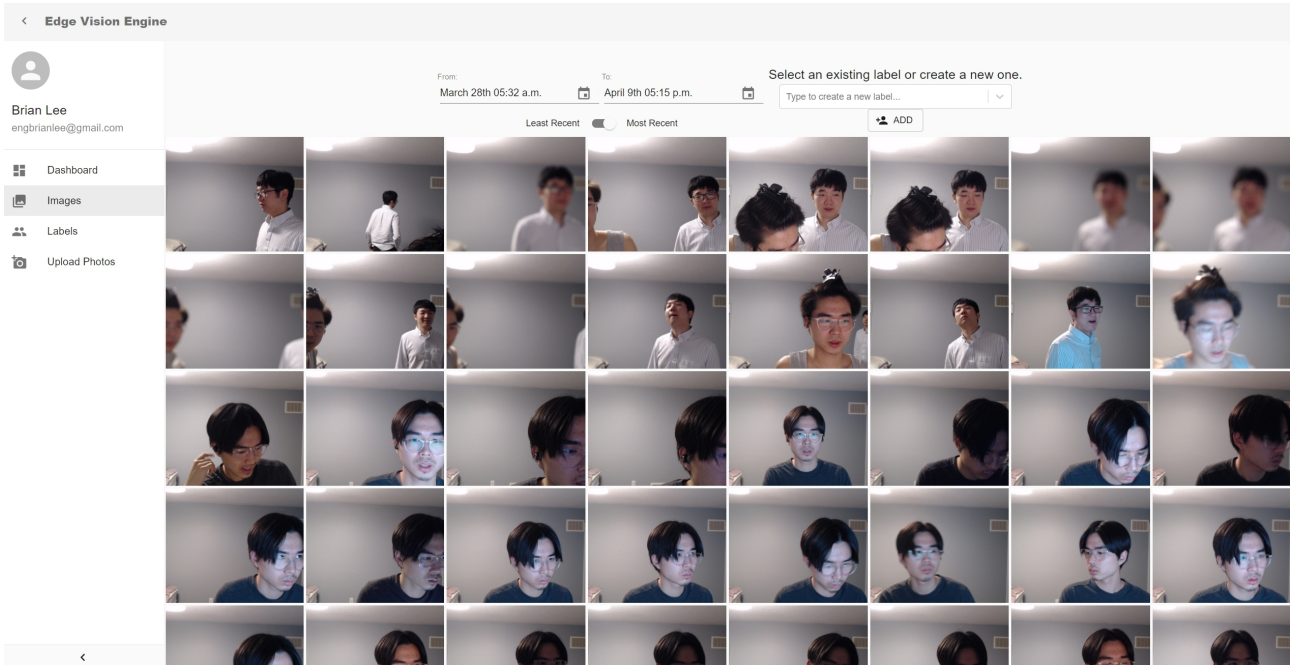

Figure 7: Web App Frontend: List all previous detections. You can filter by date as well as select images to label.

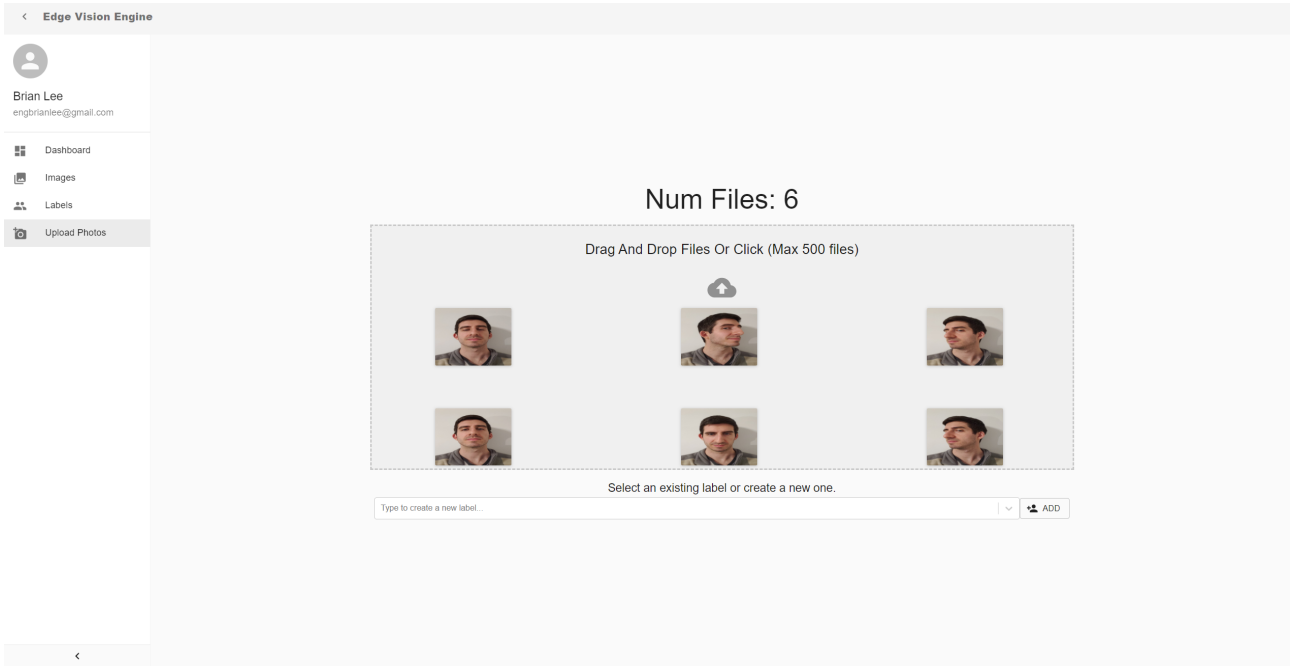

Figure 8: Web App Frontend: Allow user to upload images for labelling.

4.4 Engineering Design

The key focus of EVE was to develop a product and service that offers a high quality experience, that not only demonstrates a proof of concept, but is compelling and refined enough for our own personal use. As such, instead of building every component from the ground up, we utilized existing deep learning models and AWS cloud services in order to integrate and build a finished product. Despite having all of these components, the entire data-flow, decision-making process, and integration of Linux services, Python code, Video streaming applications, AWS API calls and resources is not a trivial task. This process forms the basis of software and hardware engineering, where existing products and services need to inter-operate in a complex fashion, requirements and constraints need to be considered, and data constantly transformed in a secure and scalable manner from one platform to another.

The breakdown below shows which elements are designed compared with what was implemented from a existing solution:

4.4.1 Edge Device (Raspberry Pi)

Designed

1. Controller (built in Python)
2. Services built for each state (Surveillence/VWW or Recording/Kinesis)
3. Noise Reduction Algorithm (built in Python)
4. Positive Edge Code to determine when to transmit image to AWS S3 Bucket (built in Python)
5. Python Wrapper for AWS SDK (establish secure authenticated connection to AWS, built in Python)
6. Design of Different States, and Metadata for each State

Implemented

1. Deep Learning Visual Wake Word (VWW) Model
2. GStreamer (to consume video stream data and transmit to specific AWS Kinesis)
3. systemd - Linux daemon/service manager
4. AWS CLI and Boto3 API calls to connect to AWS services

4.4.2 Cloud (AWS)

Designed

1. Data path and architecture to integrate different AWS Services together (AWS Iot Core, AWS S3 Buckets, AWS Kinesis Stream, AWS Rekognition)
2. Architecture and design to relate S3 Bucket images to feature vectors stored in AWS Rekognition (this solution does not natively exist in AWS, need to design algorithm(s), data flow, an API calls)

Implemented

1. AWS IoT Core Service, Device Shadow (MQTT broker), and APIs
2. AWS S3 Bucket Services and APIs
3. AWS Rekognition Service and APIs
4. AWS Kinesis Video Stream Service and APIs

4.4.3 React Web App

Designed

1. ReactJS Code to build entire website from scratch
2. Data flow, web application structure, integration of AWS services with web app views
3. Design of Different States, Metadata for each State to be obtained using GET and UPDATE commands
4. Device Dashboard (UI and Code) to Read Device State and Metadata, and make Write operations
5. Live Video Stream - wrote code to interface with video stream data from AWS Kinesis, and display in the web browser
6. Display of detected person with either `<NAME OF PERSON> Detected` or `Unknown Person Detected`, with bounding box obtained from AWS Rekognition
7. User Interface to Upload Photos for Labelling (drag and drop interface, with code to handle images and make secure API calls to AWS S3 Buckets to store photos)
8. User Interface to Label Photos (make API calls to AWS Rekognition from images stored in AWS S3 Buckets to index image and obtain feature vector, create collections according to label, and assign S3 Bucket Image ID to feature vector create 1-to-1 mapping between feature vector and AWS S3 Bucket).
9. User Interface to view existing images. Images can be acquired from VWW Model from Raspberry Pi that is stored in an S3 Bucket specific to each IoT Device, or images can be manually uploaded from user with a label (put into a collection), that is used to classify images.

Implemented

1. Material UI - React components to format visual appearance of UI elements and web components (reduce complexity of CSS).

5 Critical Problems Solved

5.1 Connecting Raspberry Pi to University Wifi Network

By default, the McMaster University Wifi network does not allow Raspberry Pi's to connect to it. This would be a critical issue during the ECE Expo since we would need each of the devices to be actively communicating with AWS. An alternative would be to set-up a separate Wireless Access Point for the IoT Devices to connect to (such as a remote hotspot). A more cost-effective method was to set-up and configure a Linux `wpa_supplicant` which enables the Raspberry Pi to connect to the encrypted McMaster wireless network. This would utilize a student's MacID WPA authentication to authenticate the Raspberry Pi.

5.2 Controlling IoT Device (Rasp Pi) from AWS

We needed a means to robustly start different programs and services on the Rasp Pi that could interface with AWS Iot Core, as well as report the current state of the Iot Device (Surveillance or Recording) to AWS IoT Core in a manner that the Web App would be able to read and write to. We went through many iterations of how to accomplish this. First we had attempted to use built-in or suggested AWS services that automate or control what is run on an IoT device from commands made on AWS. Investigations were made using services such as AWS GreenGrass, as well as AWS Lambdas. We found that it was comically easy to be overwhelmed with the plethora of options and seemingly simple to implement services that AWS offered. However, we found that many options were far more complicated, with more moving parts and potential avenues for failure, as well as more difficult to modify, if we simply utilized every service that AWS offered for our platform.

Instead, after trying out multiple avenues, we discovered that we could avoid using AWS lambdas and AWS GreenGrass altogether, and instead develop a controller program that interfaces with the built-in Linux daemons/services manager called: *systemd*. This controller, written in Python, would use a variety of Python libraries to make calls to *systemd* manager via *systemctl* commands in originating from Python code. This reduced the overall complexity of our systems, and also gave us a deeper understanding of how the Raspberry Pi interfaces with AWS, as well as a deeper understanding of systems programming, daemons/services, and operating systems as a whole. The controller script was also easily modifiable to use AWS authentication for secure cloud communication, data transfer, and quickly modified to use different AWS services (e.g. AWS S3 Buckets, AWS Iot Core, AWS Kinesis).

5.3 Detecting a Human in Frame

Our original conception for detecting and classifying human faces was to utilize a camera situated at eye-level, the individual would be prompted to look at the camera, a picture would be taken and undergo a series of data processing steps (e.g. image segmentation) for later deep learning classification and feature vector extraction. The goal is to reduce the amount on on-board computing, and reduce how much data is being transmitted (for latency concerns). However, after discovering Dr. Song Han's research, the deep learning compression pipeline, and the VWW model, we quickly pivoted from this approach, and brainstormed how to intelligently leverage the VWW deep learning model that would demonstrate powerful inference at the end (as we suddenly had a platinum milestone in sight), and set new objectives that took advantage of these new capabilities.

The quality of the VWW deep learning model is incredible, and significant testing has yielded that little to no false positives are produced from observing the running model process different humans in frame in various backgrounds and conditions. If there are no people in frame, the model will not output a detection. However, we did immediately notice many false negatives, that is, if there was a person in frame, we would sometimes see the occasional: "no person in frame". From a deep learning point of view, and from the unbiased nature of the VWW model, there should be no reason why we get this distribution of error. We deduced that this must be a hardware issue. After some investigation, we realized that due to the auto-focus of the camera we were using, it would cause the image to become very blurry depending upon how the subject entered or exited the camera frame. As a result, when a blurry image was captured, it would fail to output a detection when a human entered into frame, or have a very low confidence. We could remedy this issue if we were to set the focus manually, or disable the auto-focus such that the focus was only for a fixed distance, however we were worried that this would introduce further complexity to our hardware system. It was much simpler, more

robust, and easier to modify, by writing a simple detection heuristic in software to correct the noise.

The deep learning model (VWW) performs a simple classification, and provides two (2) classification scores: *Person_Score*, and *No_Person_Score*. For each frame, we pass the image through the deep learning model, and run our detection heuristic. After several iterations, we decided upon the following process:

1. First determine which score is higher with a function *Evaluate_Scores*, and obtain the confidence.
2. We have a *is_detected* if the confidence is greater than the *Minimum_Detection_Confidence*.
3. Store this result in our buffer to use later (where we later count the positive hits in our buffer).
4. When we have processed the total number of frames in a given time interval (given by: *Notification_Interval* number of frames), send the detection decision to AWS IoT. We check if the last *Notification_Interval* number of frames are greater than or equal to: *Minimum_Interval_Percentage* detection rate (this frame count was stored in our buffer).
5. If decided there is a detection, additionally send a photo on the rising edge going from not detected to detected (save computational resources).

5.4 Resource Reduction for IoT Devices and AWS

The purpose of the Surveillance state of operation (when it is running the Visual Wake Word or VWW), is to allow the Raspberry Pi to be running in a reduced or low power state for intelligent surveillance that can take place over a longer period of time. By comparison, the video streaming feature consumes far greater resources than the VWW, both regarding on-board memory and processing power, as well as bandwidth and communication requirements in constantly transmitting frames to AWS Kinesis. In order to further reduce the amount of resources utilized by the Raspberry PI when running the VWW, we modified the noise reduction algorithm to only transmit an image to AWS S3 database on a *positive edge*. That is, if VWW detects an individual/human in frame, and the previous state of the VWW was *No Person Detected*. Therefore, if a person continually remains in camera frame while VWW is running and constantly detects a person, then it will only send a single picture for the first time that it had detected the individual.

We realized that it was critical that we limited the number of pictures that were sent to AWS S3 Bucket, since the first time that we enabled the Surveillance (VWW) to send a picture to the cloud, we ended up sending hundreds of photos in the span of a minute, and we realized that our data stores would not only overflow, but it would be impossible for us to analyze or display these photos to a user. Our first iteration on this was to reduce the frequency at which we transmitted photos (e.g. send 1 photo update every second if a person was detected), however we found that this was also too frequent. After considering things from a potential user's perspective, we wanted the information they read on the Web App to be maximally meaningful. Furthermore, we wanted alerts and notifications from the surveillance to help prompt a user action. One way to accomplish this was by only sending an update to the AWS IoT, and transmit a picture, on the rising edge of a person being detected. This way, the user would know that when they received an alert, that the previous state was that there was no person recognized in frame.

5.5 Training and Utilizing Neural Network for Facial Classification

In addition to learning about compressed deep learning models, extensive research yielded that amount of data required to train robust facial classification deep learning model would require an exorbitant amount of resources, with training sets on the order of tens or hundreds of millions of faces, massive GPU computing clusters, expertise in training and optimizing the models, and application program interfaces (APIs) to utilize and integrate the models into our existing design. After researching online for existing resources, it appeared that AWS Rekognition offered many of the capabilities we were looking for. This was contrasted with were easier and simpler Python-based methods that we could natively run on a Raspberry Pi, however they lacked the computational power to be able to pull images simultaneously from multiple cloud-based databases, or run the extracted image data through multiple deep learning classifiers. Our main goal was to extract a feature vector that classified a human face in an image, but AWS Rekognition also provides robust deep learning models that also auto-generate facial bounding boxes, as well as APIs that can compute the comparison of facial feature vectors, and result the percent confidence between two compared vectors.

Integration of AWS Rekognition as difficult at first, since we originally wanted to compute facial recognition on a live video stream, and draw a bounding box and classficiation in real-time. However we realized that the interface for AWS Kinesis did not offer what we desired, and we had already integrated that service as our video solution, and did not want to toss our the baby with the bath water, and return to the drawing board. Any real-time video monitoring would require expensive online services that did not come with the AWS Free-Tier that we were enjoying. Instead we compromised, and decided to have repositories images taken from each Raspberry Pi when the Surveillance program was running. and store them in S3 Buckets. AWS Rekognition would perform analysis after the fact, and utilize collections of labelled images that were either pre-built in advance, or built after an person was detected. It turned out that performing facial recognition as a post-processing step was far more flexible, and gave the user greater decision-making power over the data they received. Even more surprising was the fact that the AWS Rekognition APIs worked so quickly, that to the user the bounding box and identity classification of the person detected appeared to occur in real-time. The only downside was that our video streaming only offered real-time capabilities, and did not have any machine learning analysis.

5.6 Mapping Resources across Services in AWS and React Web App

Acquiring images from the Raspberry Pi (IoT device) was a straight forward process. After the Raspberry Pi detected a person on a rising edge, it would use its AWS authentication keys to transmit the image via HTTPS to an AWS S3 Bucket. However, once the image containing a person was in the S3 Bucket, how would we be able to label the image with an ID pertaining to the identity of the person captured? Moreover, after an image has been labelled, how do we classify the identity of new images that are captured based upon our labelled images? One method would be to build a separate AWS database that is designed for storing metadata (called AWS DynamoDB). However this added significant complexity, and needed to be well-designed to easily integrate with newly added S3 Buckets.

This problem was solved from the React Web Application side by having a feature that allowed the user to upload custom images that are of high enough quality that they can be used as the ground truth for labels. Furthermore, we created an interface that allowed the user to custom label any image, whether it be user uploaded, or actively acquired from the Raspberry Pi during Surveillance (VWW). When the user labels an image, the process is more involved, where the image stored in an S3 Bucket is indexed into a collection that shares the

same name as the label. For instance, if we labelled an image with the name "Brian", then it would index that image (extract its feature vector) using AWS Rekognition deep learning models, and place the feature vector into a collection with the name "Brian". Any additional images that are labelled with same name "Brian" will be added to this same collection.

When new images come in, and we wish to classify the identity of this image, the Web App automatically calls AWS Rekognition to extract the feature vector of that image, and compare it to every feature vector in every collection. This process classifies the image, however what if we want to see which images it has the closest similarity to? Well, this was solved when we first label an image. The `external_image_id` of a feature vector is set to the URL of the original image in the S3 Bucket that was used to generate that feature vector. Therefore, when classifying the identity of an image, the collection of feature vector that has the closest similarity, will also possess the metadata corresponding to the pictures that compose that collection. The Web App will use the URL's stored in the AWS Rekognition to then extract the original S3 images, and use the bounding boxes from the feature vectors to dynamically draw a box around each of the returned faces. The list of these faces will appear when an image is classified by AWS Rekognition according to one of the collections. Lastly, the classified image will also have it's own bounding box dynamically drawn since it too has had its feature vector generated from the AWS Rekognition deep learning models.

6 Conclusion

The Edge Vision Engine (EVE) boasts a scalable and affordable intelligent surveillance and human detection system, through the use of compressed deep learning models running on low power edge devices. The scalability is achieved jointly through the paradigm of modular IoT devices that are able to inter-operate seamlessly with reliable and secure cloud services that are inexpensive compared to traditional alternatives. The low memory footprint of the compressed deep learning model allows EVE to perform inference at the edge, significantly reducing latency, communication, and bandwidth requirements of our system, and is power efficient. A React Web App allows the user to receive updates and alerts about their environment of interest, whether it be the home, office, business security, or major event space. The web app can be viewed anywhere, as long as an internet connection exists. Our team is pleased to present a final product that is a complete end to end solution, and in its current iteration, is perfect for personal use. Further work can yield a product that may meet business and enterprise needs.

We were able to meet all of our project milestones, from bronze to platinum, and we were able to quickly pivot our project direction from our original targets upon discovery of Dr. Song Han's research and the compressed deep learning model pipeline. The robust VWW deep learning model enables accurate detection of a human in frame, and intelligently sends data to the cloud (on a rising edge of detection), for further classification. Integration of IoT devices was accomplished via AWS IoT Core, as well as utilizing existing AWS services and APIs. The performance that the VWW deep learning model achieves is typically in excess of 95% confidence. These images are processed server side using AWS Rekognition to both classify the identity of the human detected, as well as produce a bounding box around the feature vector of the human face. At all times, AWS IoT Core acts as a source of ground truth for each IoT device, and relays this information to the user via the React Web App. The web app dashboard enables read and write operations for the user control the IoT Device between Surveillance (VWW) and Recording (Kinesis video stream). These alerts allow the user to take actions in real-time to go from low latency and resource Surveillance mode (using VWW), to a deep-dive via a live video stream using Recording (Kinesis).

The user can provide sample labelled images through the web app as a source of truth for the identity of images being classified. In practice, the accuracy of the AWS Rekognition deep learning models can perform classification of images from the edge device with exceptional accuracy, upwards of 98% to 99% on average. Furthermore, this performance can be achieved with only a single labelled picture as the source of truth! All of the functionality achieved by the EVE system is scalable, secure end to end, and the user only pays for the services that they use. Although our group did not have to pay for any AWS cloud services due to the 12-month Free-Tier promotion offered to new AWS accounts.

Message to the Reader: For a live demo and additional info please visit our documentation site: `https://engbrianlee.github.io/EVE_frontend/#/`

7 Future Work

To bring EVE closer to a commercial product that meets the needs for households, businesses, and enterprises, the entire deployment of EVE to the Edge Device (Raspberry Pi) would have to be automated. Furthermore, the allocation and set-up of AWS accounts, services and resources would also have to be automated. There are existing tools and scripts that enable this to occur. Furthermore, we would want the Web Application to be hosted on AWS as well. Currently the React Web App will run on an individual's own computer, however in order for it to be scalable, avoid security leaks, and reduce system dependencies, the Web App can also be hosted internally on the cloud in AWS.

7.1 AWS and Web App Resource Management, and User Authentication

Overall, we were very pleased with how our React web application interfaces with AWS. Originally we wanted a web application that would be accessible for any user to open up on the web, and be able to access the EVE Dashboard purely in a *Read-Only* state. In order for EVE to be delivered as a viable product, the next steps in development would be to solve both the secure user authentication problem (for *Read-Only* access), as well as efficiently cache data in the browser to avoid data from constantly being re-pulled. Another feature would be to offer the user to refresh certain pages (e.g. currently saved pictures in the S3 Buckets), instead of it automatically re-pulling those resources every time one navigated to the page. This would require a user authentication process, as well as require us to build a secure web interface from the web app to AWS. Due to limited time resources, and the ambitious nature of our project, we decided to avoid implementing a *Read-Only* User interface for our web app, and avoid hosting it online. Instead we resorted to only having a local React web app that would interface with AWS. Furthermore, the React Web App was purely developed for demonstration purposes. We found that many aspects of the app would actively pull resources from AWS services (such as S3 Buckets), every time one navigated to a different part of the web app. This would result in the web app sometimes re-pulling hundreds of images every time you refreshed the page, or moved between views. In a production environment, this would be completely unacceptable, and a catastrophic waste of resources. During extensive testing, we noticed that many aspects of our AWS services were near the limits of our Free-Tier allotments. Furthermore, we gone over our limits for S3 Bucket *GET* requests, as well as been charged for other AWS services that we were experimenting with a possible solutions. Oftentimes we did not know if we needed to use a certain service to solve a given problem, nor did we know much of a given service did we need to use before performing basic testing and prototyping.

7.2 Deep Learning Enhancements

An immediate improvement to EVE is to enable AWS Rekognition to be actively performed in real-time on an AWS Kinesis stream, instead of merely static pictures that are acquired from the Surveillance (VWW) model that runs natively on the edge device. We would be able to extrapolate all of the current functionality to a real-time stream, where a dynamic bounding box would be present on human faces in the streamed video frames, as well as classification of human identities using the labelled ground truth indexed images in AWS Rekognition collections. This may require some changes in our current infrastructure and data pathway in the AWS Cloud in order to process this data effectively.

Another possible avenue for improvement is to have multiple bounding boxes produced for any image that is acquired. This would require more sophisticated code, management of data, and calls to the AWS Rekognition API in order to obtain multiple feature vectors of every human of interest in a given frame, and iteratively draw these bounding boxes. This process may be done on either still images, or on a video stream. In addition to multiple bounding boxes, if these are performed frame by frame in a video stream, it would have the effect of performing motion or object tracking. Not only could this be done for human faces, but also for any object of interest that we wish to specify for the AWS Rekognition service. This service is very feature-rich, and provides us with many possible ways to enhance our analysis of data acquired about a given environment.

7.3 Enhanced Notifications

In addition to alerts that may take place on the React Web App, there were many other means for us to notify a user that a change in the device state has occured, or a certain event has been detected. Due to time and resources constraints, and avoiding unnecessary complexity, we did not implement any of these additional features. It is a straight forward process to perform Email or SMS notifications for different detected events, or operational states of the edge devices.

7.4 Richer Analysis of Environment and IoT Considerations

The current iteration of EVE only demonstrates the capabilities of deep learning applied to a camera that is dealing with either single frames, or streamed video. There are a plethora of different sensors can be explored that would allow for a richer measurement of information about our environment that could be useful for state estimation, human detection, as well as allowing the device to operate in not only a lower power state, but be able to sense a wider variety of environments. For instance, if the environment has consistent low lighting, then a typical video camera would be relatively ineffective at obtaining high enough quality images for accurate detection of humans in the field of view (FOV). However, the device may be more robust in a larger variety of environments if augmented with different sensors that can detect a larger variety of electromagnetic signals, such as: sound or proximity sensors, IR or Infrared sensors, temperature sensors, etc. In line with the IoT paradigm, and to potentially meet the needs of clients and users that may not have more constrained power requirements, it would be helpful to perform in-depth testing for how long the device can run on battery power, have useful internal measurements about power use, as well as optimize the hardware selected, and software routines to reduce power consumption.

Overall EVE exceeded our expectations for a complete end to end product, and we look forward to the future developments in the field of inference at the edge.

8 References

[1] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 2019.

[2] Song Han Hai Cai, Ligeng Zhu. Proxylessnas: Direct neural architecture search on target task and hardware. *ICLR 2019*, 2019.

[3] Yiming Yang Hanxiao Liu, Karen Simonyan. Darts: Differentiable architecture search. *ICLR 2019*, 2019.

[4] Yutao Huang, Xiaoqiang Ma, Jiangchuan Liu, and Wei Gong. When deep learning meets edge computing. *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017.

[5] Yujun Lin Ji Lin Song Han Kuan Wang, Zhijian Liu. Haq: Hardware-aware automated quantization with mixed precision. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[6] William J Dally Song Han, Huizi Mao. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations*, 2015.

[7] Zhijian Liu Hanrui Wang Li-Jia Li Song Han Yihui He, Ji Lin. Amc: Automl for model compression and acceleration on mobile devices. *The European Conference on Computer Vision (ECCV)*, 2018.

[8] Han Yiwen, Xiaofei Wang, Victor C.M. Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys and Tutorials*, 2019.