MaViS

Machine **Vi**sion **S**ecurity System

Short Report

Jason Deglint, PhD
&
Juan Park, MASc

May 14, 2021

Video demo and walkthrough: https://youtu.be/UVe5LXdPUYs

GitHub Repository for Nvidia Jetson code: https://github.com/jasondeglint/MaViS

Summary: The MaViS (**Ma**chine **Vi**sion **S**ecurity) system is a machine learning based security platform that automatically monitors and detects people in a scene, and then alerts the user in real time by sending an image and video to their email. The system is enabled through a combination of edge computing and cloud infrastructure. The edge platform used was the Nvidia Jetson Nano 4GB Developer Kit, and the cloud infrastructure was built using Amazon Web Services (AWS).

Table of Contents

Disclaimers

Sponsorship

This work was sponsored by Blue Lion Labs[1]. Both Jason Deglint[2] and Juan Park[3] are employees of Blue Lion Labs. Blue Lion Labs paid for the enrollment of Full Stack Deep Learning[4] course for which this project is a course requirement.

The tracking code for the Nvidia Jetson has been publicly released on Github[5]. However, the AWS cloud infrastructure code could not be made available as it is part of Blue Lion Lab's market offering.

Contributions

Jason Deglint designed and implemented the edge devices (Raspberry Pi 4 and Nvidia Jetson Nano), and Juan Park designed and implemented the cloud infrastructure (S3 bucket, Lambda functions, and RDS). Both Jason Deglint and Juan Park wrote this report.

[1] https://bluelionlabs.com/
[2] https://www.linkedin.com/in/jasondeglint/
[3] https://www.linkedin.com/in/juan-park-92a42b94/
[4] https://fullstackdeeplearning.com/
[5] https://github.com/jasondeglint/MaViS

Overview

As seen in Figure 1, the MaViS (**Ma**chine **Vi**sion **S**ecurity) system is an intelligent vision-based security service designed to automatically detect humans in a visual scene and notify the property owner in real-time. This is to ensure any and all unauthorized personnel on the private property are identified as soon as possible to provide a record of intrusions. The functionality is provided using a combination of deep learning, edge computing, and cloud technology, where the camera on-board the edge device continuously monitors an area. When the real-time analytics on the edge device detects one or more people in the scene, an image is immediately captured and the user is notified via email. Soon afterwards, a video clip of the scene with the person(s) is recorded and also sent to the user via email. The image and video are stored in the cloud for on-demand user access.

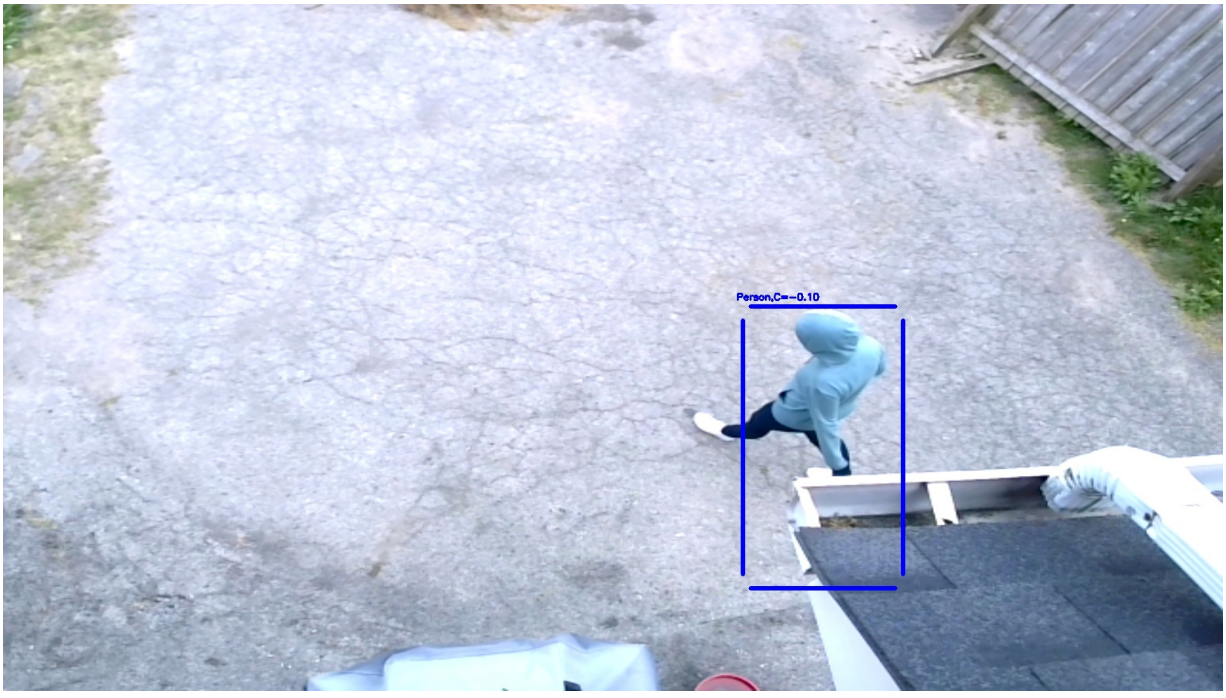

Figure 1: A sample output image of the MaViS system. This image was captured by the Nivida Jetson and uploaded to Amazon AWS. Once uploaded an email is automatically sent to the user in real time.

Motivation

Home break-ins, porch pirates stealing deliveries, and trespassers cause headaches for homeowners everywhere. Installation of surveillance cameras equipped with motion sensors are a common solution to this problem. However, surveillance cameras only detect motion and collect data - meaning that the user must sift through frames in the video footage to discern any meaningful information. As a result, this visual information is typically inspected after an unfortunate incident, such as a package being stolen.

To improve this situation, we propose MaViS, a machine vision security system to automatically collect and analyse relevant images to identify critical and actionable information and send it to the user, in real-time.

System Overview

As seen in Figure 2, MaViS operates through a combination of an edge device and cloud infrastructure services to provide the user with real-time visual (image or video) updates via email of any personnel detected on the camera onboard the edge device. When the edge device detects a person in the scene, an image is captured and uploaded to the cloud and a link to access the image sent to the MaViS user. A video recording of the intrusion event is also captured and uploaded to the cloud, where its access link is also sent to the user. The user will then be able to access the image and video directly from any electronic device.

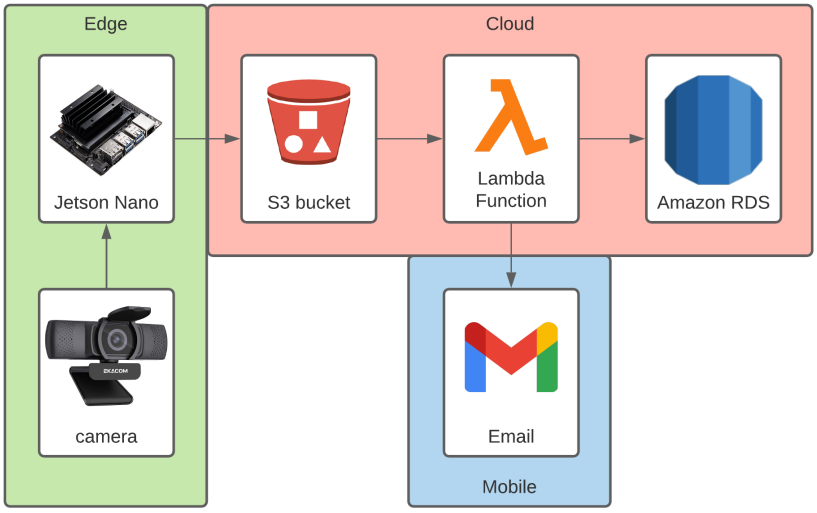

Figure 2: The MaViS system is broken into three main sections: (1) the data is collected and processed on the edge, (2) this data (image and video) is uploaded to the cloud for storage as well as (3) sent to the users email for real time notification of an intruder.

Hardware & Software Stack

The software stack for MaViS can be seen in Figure 3. The MaViS first iteration of hardware was a Raspberry Pi 4 with the Detectron2 framework, which was replaced by the NVIDIA Jetson Nano for the second iteration and final iterations. For a database management system, AWS Relational Database Service (RDS) are used, images and videos are stored in an AWS S3 Bucket, and data is processed in the cloud using AWS Lambda functions. AWS SES is used to send emails to users. The deep-learning person detection functionality is enabled on the Jetson Nano using NVIDIA's DeepStream toolkit. Code for the edge device application and AWS lambda function is written using Python.

Figure 3: MaViS software and hardware stack showing all services and tools used during this project.

Edge Device

The overall goal of the edge system is to detect a human in a scene and upload this to the cloud within 5 minutes so that an alert can be sent to the user.

The edge development went through 3 iterations, as seen in Figure 4. The initial approach to solve this problem was to use a Raspberry Pi 4 computer and run inference on the CPU. However, after experiencing challenges with pre-trained models and libraries on the Raspberry Pi, we decided to switch to the Nvidia Jetson Nano. Once using the Nano we decided to use the DeepStream SDK which used the built-in GPU of the Jetson Nano and allowed real time processing of data and real time uploading of data to the cloud.

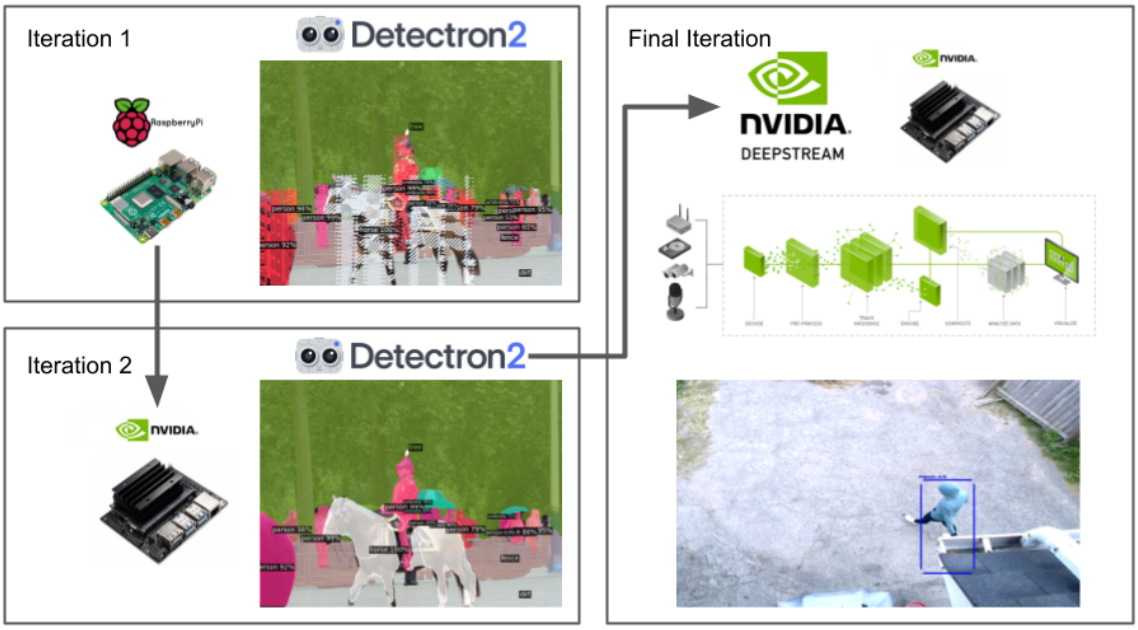

Figure 4: The MaViS edge component went through three iterations. The final system used the Nvidia DeepStream SDK on the Nvidia Jetson Nano.

Raspberry Pi

Given the goal of detecting a human in a scene, the initial plan was to use a hardware sensor on a Raspberry Pi 4 Model B[6] and use a hardware sensor to detect motion in that scene. Once motion was detected an image would be captured, and then processed on the CPU on the Raspberry Pi. Before testing the hardware sensor we wanted to ensure the Raspberry Pi 4 could run inference on a deep learning model on it's CPU. To accomplish this the following steps had to be complete: (1) install a deep learning framework on the Raspberry Pi and (2) run inference on the Pi's CPU within 5 minutes.

Installing PyTorch and Detectron2

The framework we chose to install was PyTorch as it is the preferred framework for us. Installing PyTorch on the Raspberry Pi proved very difficult until we found a dedicated OS image from Qengineering that had Pytorch 1.8.0 and TorchVision 0.9.0 natively installed[7]. This image from Qengineering also comes with OpenCV 4.5.1, TensorFLow 2.4.1 and TensorFlow-Lite 2.4.1.

Given the installation of PyTorch we decided that to determine if a human was present in an image we would run panoptic segmentation on that image. The library we choose to use was Detectron2, which is Facebook AI Research's next generation library that provides state-of-the-art detection and segmentation algorithms[8]. Installing Detectron2 on the Raspberry Pi was as simple as following the installation instructions[9].

Running Inference

With PyTorch and Detectron2 installed we can now take an image and determine the inference time on the CPU. The inference time for a sample image of 1600 x 900 pixels took 76.19s seconds, and therefore was suitable for our task of reporting the presence of a human within 5 minutes.

However, when running the sample code provided by Detectron2 with the R50-FPN model, the output mask was incorrect, as seen in Figure 5. To verify the install and code were correct, we ran the exact setup and code on a different personal laptop. When doing so no incorrect output was found. To help with debugging the issue we created a github issue on the Detectron2 github page[10].

---

[6] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[7] https://github.com/Qengineering/RPi-image
[8] https://github.com/facebookresearch/detectron2
[9] https://detectron2.readthedocs.io/en/latest/tutorials/install.html#
[10] https://github.com/facebookresearch/detectron2/issues/2952

Figure 5: The incorrect output when running Detectron2 on the Raspberry Pi 4 model B CPU.

Nvidia Jetson Nano

Given this error on the Raspberry Pi we decided to explore if the Nvidia Jetson Nano[11] was a more suitable platform for our use case. While the Nvidia Jetson is slightly more expensive, it comes with an onboard 128-core Maxwell GPU, which would enable real time processing, a significant advantage for home security. Given the available ecosystem of Nvidia tools, we wanted to control video on the Jetson Nano, and so we installed DeepStream SDK 5.1[12]. DeepStream SDK 5.1 requires the installation of JetPack 4.5.1[13]. Both the installation of JetPack and DeepStream was straightforward using the online documentation.

After installing the PyTorch and Detectron2 frameworks on the Jetson Nano, we decided to explore the native pre-trained models on the Nvidia platform. These models come pre-compiled and ready to run on the Jetson Nano device. This saves time since the alternative is to take a

[11] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/
[12] https://developer.nvidia.com/deepstream-sdk
[13] https://developer.nvidia.com/embedded/jetpack

torch model and convert it to ONXX, and then use the Nvidia TensorRT framework to export a model that can run on the Jetson Nano[14].

The DeepStream SDK is built on the GStreamer framework[15], which has a very steep learning curve. After spending a significant number of hours going through the tutorials and example code, we were able to write a script that would automatically detect people in a video stream and write these images to a folder, as seen in Figure 6. The pretrained model that we chose was a ResNet10 model trained to recognize the following classes: `"Vehicle"`, `"TwoWheeler"`, `"Person"`, `"RoadSign"`. We customized our code to ignore all classes except for the "Person" class, which was done in the `mode_config.txt` file.

The Python code for the Nvidia Jetson is broken into two files: `main.py` and `monitor_and_upload.py`. The `main.py` contains all the DeepStream code and is responsible for the real time analysis of the video stream from the camera. When a frame in the stream contains a detection, this frame is automatically written into a specific image folder.

The `monitor_and_upload.py` script automatically watches the specified image folder. As soon as a new image arrives, it immediately uploads this to AWS to alert the home user. Once no more frames have been added after 5 seconds, the `monitor_and_upload.py` script creates a video from all frames in the image folder and uploads that video to AWS, which will also trigger a second email notification to the user.

Once both the image and video have been uploaded to AWS, the `monitor_and_upload.py` script saves a local copy on the device in the specified archive folder. The script then deletes all frames being stored in the image folder.

[14] https://developer.nvidia.com/tensorrt
[15] https://gstreamer.freedesktop.org/

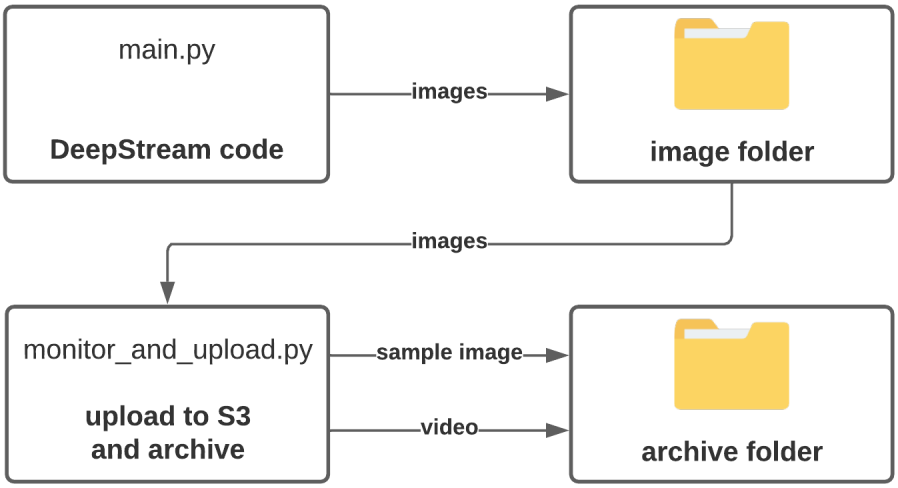

Figure 6: The Python code for the Nvidia Jetson contains two scripts: First the `main.py` script monitors the video stream and automatically saves frames that contain a positive classification. Second, the `monitor_and_upload.py` script uploads a sample image as soon as an intruder enters a scene, and then also uploads a video once the intruder leaves the scene.

Cloud Infrastructure

The overall goal of the cloud infrastructure is to process images and videos uploaded from the edge, store it in S3, add records to the database, and send links of the video and image to the user for easy access via email.

Implementation Details

The cloud infrastructure is composed of four AWS services: S3 object storage, a relational database service (RDS), and two Lambda functions. The S3 object storage is used to organize and store data from the edge, which is accessible to MaViS users. AWS's RDS is used to host a Postgres version 12.5 database that is used to store device and DL model metadata as well as records of image and video data that is input into the S3 bucket. This information can be used to keep track of the status of all deployed devices and models, and the metadata of each individual image and video for easy retrieval for any tasks (e.g. model diagnostics, additional data labelling, etc.).

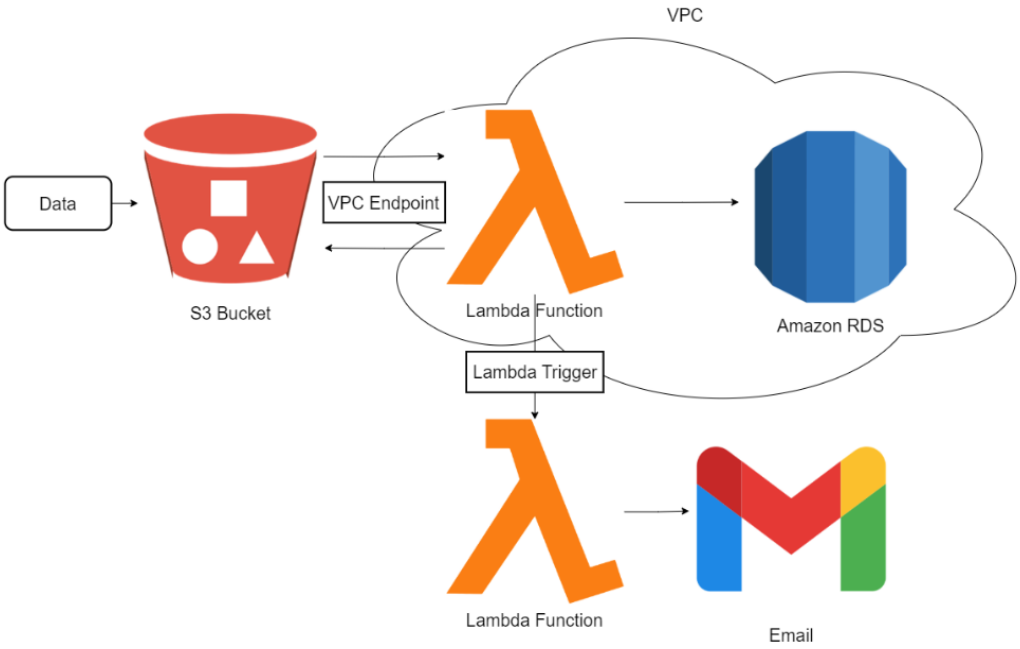

Figure 7: The process flowchart for the cloud infrastructure. Data from the edge is stored in the S3 bucket, which triggers a Lambda function that processes the data, uploads records to the RDS database, and triggers the second Lambda function that sends the notification email to the user.

An AWS Lambda function is used to perform a single function on demand. In our case, the first function is triggered when a video or image is uploaded to the cloud. The first Lambda function processes the data from the edge and creates records of the newly added data into the database. Afterwards, the second Lambda function is triggered to send an email to the user notifying them of the intrusion with photo and video evidence.

Connectivity issues with AWS Virtual Private Cloud

For security reasons, the database is housed in an AWS Virtual Private Cloud (VPC). A VPC is a logically isolated virtual network where AWS services can be stored to restrict network access into and out of the VPC. Since the database resided in a VPC to ensure secure access, there were some difficulties in connecting all the AWS services together due to the isolated nature of VPCs.

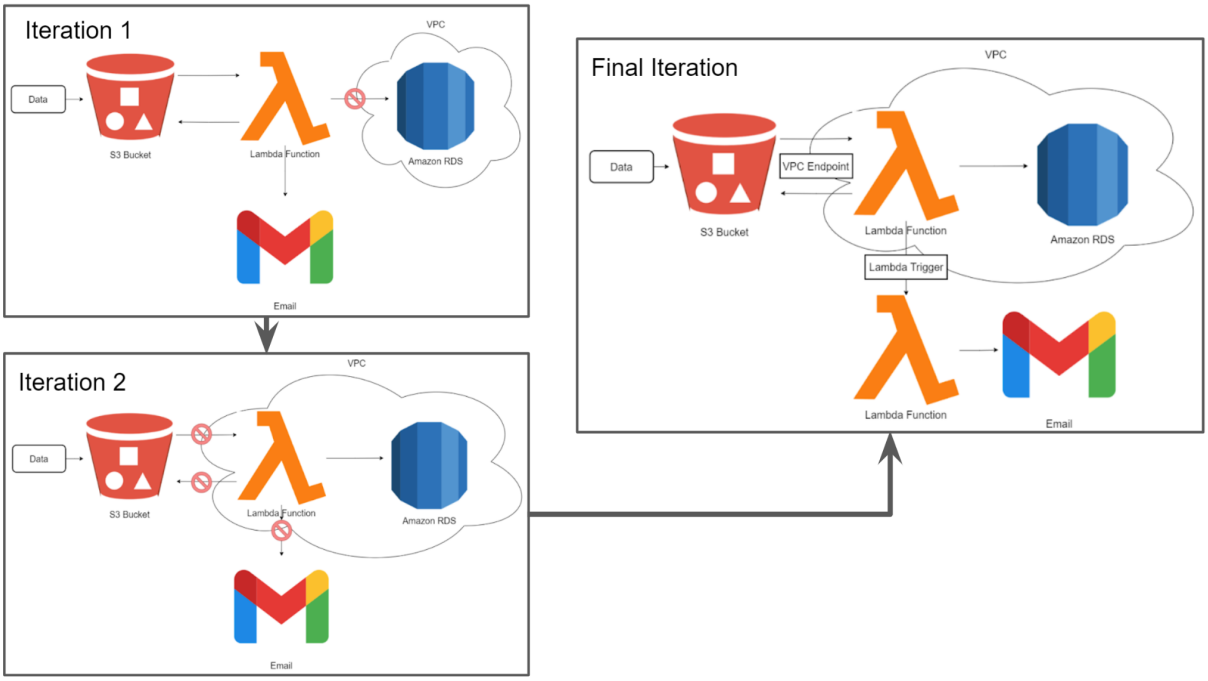

Figure 8: The iteration process for the cloud infrastructure. Interaction 1 has the Lambda outside the VPC, and has connection issues with the RDS database. In iteration 2 we move the Lambda inside the VPC at which point, the connection issue with the database is resolved, but new connection issues arise between the S3 and SES services. In the final iteration we add in a VPC endpoint to facilitate connection between the Lambda in the VPC and the S3, and take the email functionality outside the VPC and place it in a separate Lambda. This new Lambda function is connected to the previous Lambda function via a Lambda trigger.

Our initial configuration consisted of one RDS postgres database in a VPC, one Lambda function not in a VPC, and a S3 bucket. The lambda function was responsible for processing and storing data in the S3 bucket, updating the database, and sending a notification email to the user. But due to the VPC, the Lambda function was not able to communicate with the database.

To resolve this problem, we placed the Lambda function inside the VPC to allow for communications with the database, but then could not access the S3 bucket and SES (AWS simple email service). During a solution search, we found AWS API Gateway, which would allow for services in the VPC to communicate with the public web securely, but did not pursue it due to the associated costs of running the service. As a second option, we found and implemented VPC endpoints, allowing the VPC to connect to other AWS services. Using this method, we connected the Lambda function in the VPC to the S3 bucket. However, the same method did not work for connecting the Lambda function to SES.

Thus, we split up the Lambda function into two functions, one function in the VPC to interact with the database and S3 bucket, and the other function outside the VPC to send notifications to users using SES, which solved our connectivity problems with the VPC.

As a result, our final cloud infrastructure configuration ended up consisting of one RDS postgres database operating in a VPC, two Lambda functions, one operating in the VPC, responsible for processing data from the edge and updating the database, and the other operating outside the VPC, responsible for notifying the user via email with links to the image/video, and an S3 bucket responsible for storing the images/videos.

Future Work

There are three areas to explore to improve the current MaViS system. They are (1) better video recording, (2) improve the model and (3) setup a local server.

The first area to improve is to add event based recording, opposed to writing all the images with positive classification into a folder and then converting all those images into a video. Saving frames that have a positive classification and then writing these frames to a video often creates a choppy video.

The second improvement that could be made is to increase the model's performance on person identification. For example, the model could be updated with a whitelist, and then the MaViS system would only alert if someone was present who was not on the whitelist. Another area would be to update the model to recognize people delivering packages or mail.

The last improvement would be to remove any connection to the cloud to ensure maximum privacy and lower costs. This would involve running a server on site, for example, on a Raspberry Pi 4. This may be advantageous for a larger building or for a user who does want their data online.