.

Electrical & Computer Engineering
Boston University
EC464 Senior Design Project

Final Report

Community Managed Security System

Submitted to

Dan Ryan
VergeSense
724 Brannan Street
San Francisco, CA 94103 United States
1-617-618-5006
dan@vergesense.com

by

Team 24
Community Security

Team Members

Samuel Evans evanss@bu.edu
Esteban Hernandez hesteban@bu.edu
Daniel-John Morel djmorel@bu.edu
Steve Numata numatas8@bu.edu
Cameron MacDonald Surman csms@bu.edu

Submitted: April 10th, 2020

Table of Contents

Executive Summary

Community Managed Security System
Team 24 – Community Security

With recent advances in machine learning technologies, there is a growing concern over video data tampering. Currently, most privatized security systems are owned and operated by third-party companies with little accountability. Our project aims to get rid of third party involvement and provide accountable security through a community audible security. The final deliverable will be a video surveillance system that can be accessed through a web-based interface. The website will allow the user to view the live stream of the cameras and the recorded video data uploaded to the cloud storage. All actions taken by the users account will be recorded in a log that can be viewed by all users. This will ensure the privacy and security of the community.

1.0 Introduction

Dan's main concern when creating this project was the over privatization of data, specifically security feeds. As information about citizens expands with our ever increasing reliance on technology, the control over this data becomes more and more valuable. Dan is looking to create a security solution that doesn't forfeit the user's privacy and keeps it out of the hands of private companies and government bodies.

In order to fully understand the value of a community-driven security system, it's best to look at current statistics regarding concerns for online privacy. In the United States, over 75% of Americans are at least somewhat concerned about data vulnerability[1]. An estimated "$19 billion in 2018" was spent in the acquisition of consumer information[2]. As this market for data increases with time, so too will the desire for individuals to protect and keep that information out of private companies' hands.

Our group's approach to this project revolved around two concepts. We were looking to create a competitive security camera that is as close to industry standards as possible, all the while being completely unreliant upon the grid. We also needed to formulate an approach to the democratic holding of power and data. The idea was that not one person within the community can hold any real power over any of the other members. We also wanted to ensure that all data access is transparent, and all usage/downloads is visible to the entire community. This led to the idea of a logbook. It would be an immutable file, available to all residents.

Our aim to provide an industry standard camera will allow us to meet various requirements including strength and durability, visual capabilities, and ease of data transfer. Beyond that, the solar cell will allow us to have an autonomous and self-reliant product. The final design decision, the logbook, is likely the most important feature of our project. Ensuring its integrity is what will breathe life into the marketability of this camera system. We believe a logbook will provide much needed accountability among the user base and expose any possible abuse of the system.

While it is a required feature, the developed logbook should be an interesting mechanic, especially seeing how vital it is to this idea of a shared community managed security system. The system's sole charging mechanism, the solar panel, will be a great testament to the overall system's ability to operate with little-to-no advanced setup and easy integration into a community.

2.0 System Description

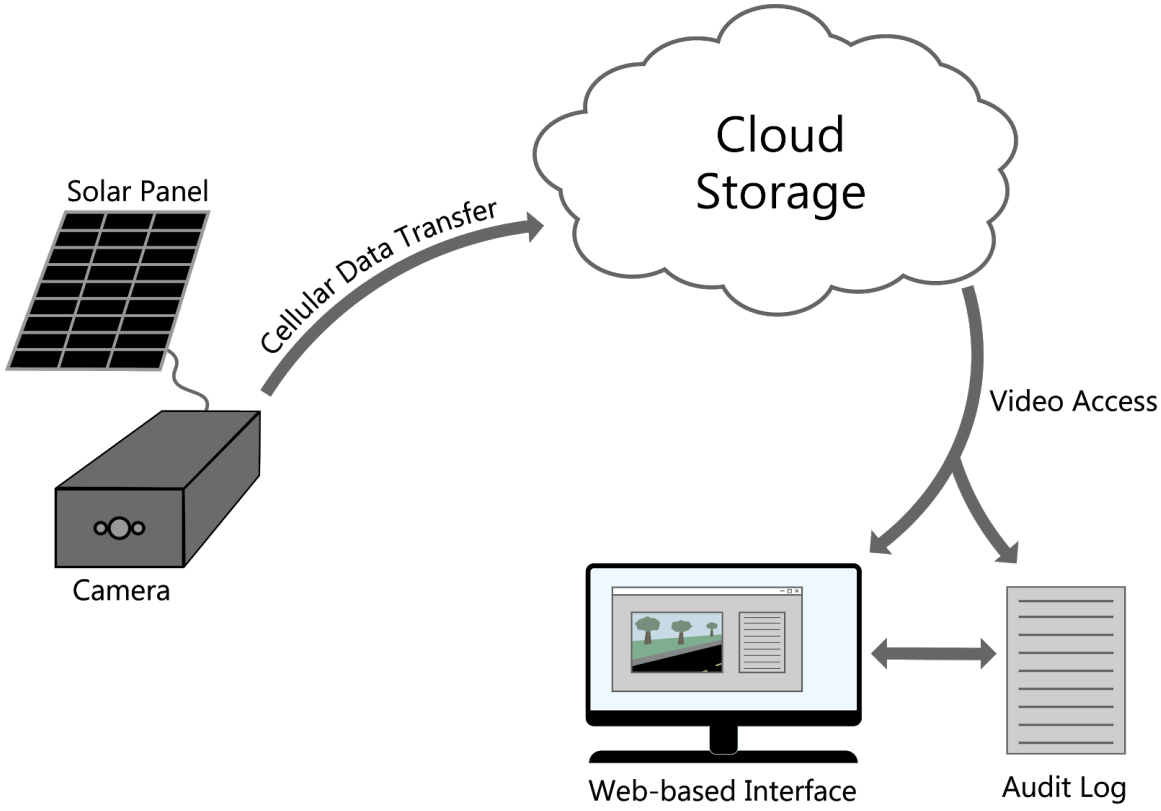

Figure 1: System Design Overview

2.1 Camera System

The current camera system is composed of two main hardware modules: the camera unit and the solar charging unit. A Raspberry Pi 3B+ microcontroller serves as the base for the camera unit. On top of the pi, there is an IR-CUT camera module to record video, and a cellular transmitter module to transfer video footage to cloud storage. The pi runs Raspbian as the operating system to regulate motionEye for video capture and crontab to regularly sync saved videos to AWS S3. The Pi also uses a piece of software from remote.it to establish a livestream connection between the pi and our website. The entire Raspberry Pi configuration sits inside an IP65, 3D printed box to protect the electronics.

The other aspect of the camera system, the solar charging unit, can be broken down into three pieces. The solar panel, charge controller, and battery connect in a chain off of the microcontroller to power the system. The overall connections are quite straightforward, and the charging mechanism insurance (current regulator) protects from overcharging the battery or battery discharge that makes it to the cell. This breakdown can be visualized by the figure below. The solar cell we are using is a 35 Watt monocrystalline PV that will suffice as an energy source in both sunny environments like California (our customer's location) as well as in Boston.

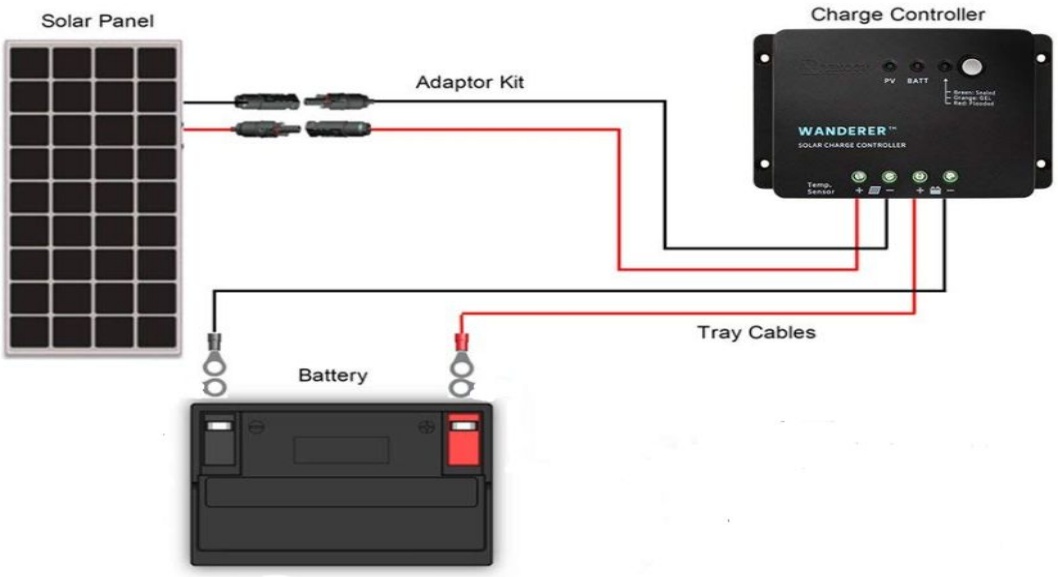

Figure 2: Charging System Overview [3]

2.2 Cloud Storage

Our initial concept for storing and serving video data to our management platform is outlined in **Figure 3** below. Utilizing AWS Kinesis Video Streams Producer SDK, we will package video data and send it to Kinesis via Twilio 4G LTE cellular data. From there, we can either serve the data directly to our management platform via HTTP Live Streaming (HLS), or use Kinesis Data Analysis and Lambda to draw insights from video data before being stored in S3 Storage. We can also choose to alert individuals or organizations via Simple Email Service (SES) and Simple Notification Service (SNS) based on thresholds we will define later. S3 will store data for up to 30 days before our data lifecycle moves it to Glacier for long term storage. We can then serve on-demand video using CloudFront to our platform as well. Elastic Compute 2 (EC2) will host our management platform, and enable us to distribute data to our customers. Another feature to be implemented later is the use of CloudWatch and CloudTrail for our audit logs, allowing users to view aggregated or specific data related to stream and VOD access.

As of project suspension in late March, we have implemented some of the original plan. Development EC2 instances for our management platform were created, and instance images taken to reproduce the hosting environment. S3 storage was utilized through the associated API, and video footage clips could be automatically synced with our cloud storage for near real-time remote access of data. These clips could then be accessed on the platform via iframes, updated via Lambda. We had a working Kinesis pipeline through which livestream video could be accessed, but ran into compatibility issues that were unresolved upon project suspension. The primary problem involved the Twilio network, as outside communication could not be received by the Raspberry Pi unless the Pi initiated the connection. Streaming video is quite expensive, so in order to

conserve resources, we would only stream once the server requested a livestream. Our team was working on ideas to initiate a handshake between the Pi and AWS so streaming could be accessed, but this was not completed. One resource that we added to our infrastructure was the use of the AWS Relational Database Service, which allowed our databases to stay synced during development. Additionally, this allowed us to use Lambda to populate the database as data was received from the Pi, allowing for a more streamlined data pipeline. Our frontend remains mostly unchanged, with the addition of React and Bootstrap for UI.

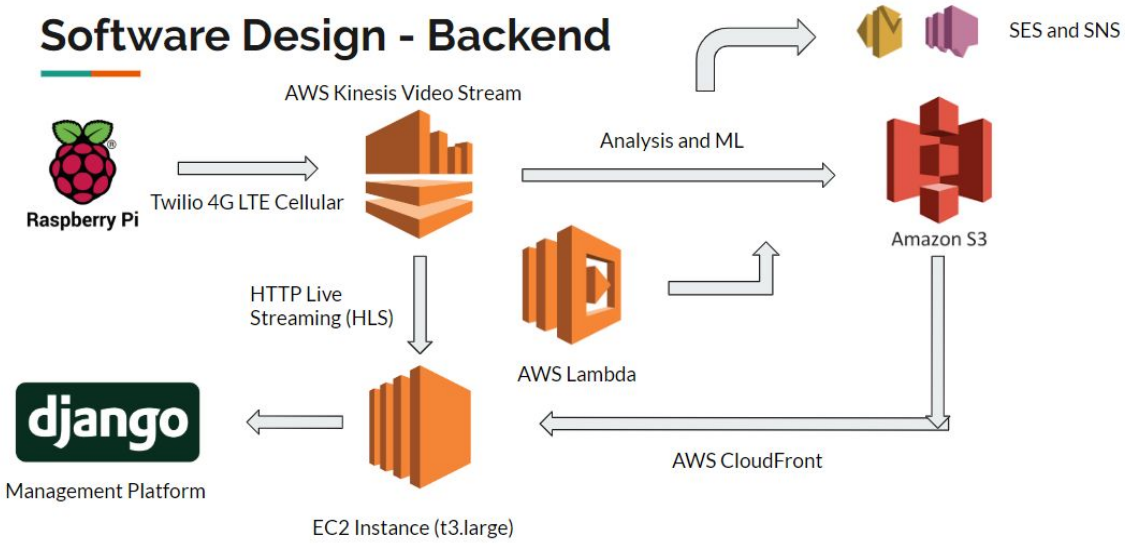

Figure 3: Backend Design Overview

2.3 Web Development

The design of the web application is a client server architecture, where the front end created using react would communicate to the backend rest api implemented in django. This allows the frontend to have dynamic changes to rendering using javascript, while the backend is used for communicating to the database as well as other platforms and processing information. The backend communicates to a Postgresql database hosted using RDS which allows for dynamic scalability based on demand, and AWS Lambda processes footage inserted into AWS S3 and inserts them into the database to use. The overall design of the system is that a user would select a camera as well as a date, and select from a list of available video timestamps to view the footage in an iframe.

Some intended features that were not implemented due to the progress halt during march was a finished version of user login and authentication, camera registration, live

stream integration into the web app (functionality of livestreaming was implemented), and a finished cryptographic audit trail.

3.0 Second Semester Progress

This semester, the team finalized the hardware design and focused on the software aspect of the project. The team continued to divide the security camera design into modular components so that progress could be made in parallel. For the most part, each component was functional, and the team was looking to connect the different parts in the month of March.

Due to the COVID-19 pandemic, progress was halted as the team could no longer meet and work on the project in person. While some progress was made with the website, further progress involved with connecting the pi to the rest of the system was not possible. Our next sections will describe the work we achieved for the semester as well as a discussion of what was in progress of completion.

3.1 Raspberry Pi Setup

The current security camera design uses a Raspberry Pi 3B+ as the microcontroller and an IR-CUT camera module for recording video. Last semester, we had the pi run on the open source operating system, motionEyeOS, for its effectiveness and simplicity in setting camera configurations such as resolution, frame rate, motion detection, and video file format. However, the addition of the cellular transmitter posed some challenges as motionEyeOS limited the software we could run on top of it. As such, we changed the operating system to Raspbian. Although we could no longer use motionEyeOS, we could use a simpler version known as motionEye. The motionEye application runs on top of Raspbian and still has the same camera configuration interface as motionEyeOS, but the application just isn't a full-fledged operating system.

As mentioned earlier, the cellular transmitter was installed on the pi at the beginning of the semester. The SIM7100A cellular transmitter had a lack of documentation, so we looked for a third party to get the drivers. Since the SIM7100A used a Qualcomm chip, we were able to use the IOTBit library from Altitude-Tech's Github repository to set up a 4G LTE connection. While the cellular transmitter's configuration could access the Internet, establishing the connection wasn't reliable. The connection used wvdial, a point-to-point protocol dialer, but the modem would often fail to establish a connection. It was suspected that the underlying cause was a power issue, but our power tests showed that the module received a sufficient amount of power according to the specifications listed in the limited documentation for the transmitter.

Because of the SIM7100A's complications, the team decided to replace the module with a cellular transmitter from Sixfab. While the new module was more than twice the price of the SIM7100, the ease of setup and quality of the data connection made the Sixfab transmitter worth it. Sixfab had their own software library for the device, which simplified the system design and enabled the pi to establish a cellular connection on boot. As such, we no longer needed the IOTBit library.

With the cellular transmitter operational, we then moved onto transferring videos saved on the pi's 256GB micro SD card to our cloud storage in AWS S3. To do so, we installed the AWS Command Line Interface (CLI) on the pi, which provided us with a sync function to sync the pi's local storage with S3. A crontab was configured on the pi to regularly call the sync command every hour.

Besides transferring saved videos, we also looked into using AWS Kinesis for livestream. The pi was unable to run a Kinesis livestream and motionEye at the same time, however, due to both processes competing for the same camera module. Since Kinesis would only livestream if motionEye was off, we removed Kinesis from the system design. Instead, we used the livestream features available from motionEye. With the motionEye livestream available over HTTP, we could forward the video feed to our website. While a separate device could access the motionEye livestream over HTTP when the pi had either an Ethernet or wifi connection, the livestream couldn't be accessed with the pi on a cellular connection.

The reason behind this cellular discrepancy is that cellular providers only enable their devices to initialize a connection. This design choice is done for security so that an external device can't ping a device within the cellular network, like a cell phone for instance. To resolve this issue, we looked into using a third party, remote.it. With this solution, the pi establishes a connection to remote.it, and an external source can then use zoom.it's connection to pull the livestream from the pi. The plan was to have our website use zoom.it's API library to retrieve the pi's HTTP livestream as a URL, and then present the video feed as a frame for our users. Unfortunately, we were unable to finalize this part of the camera design due to the COVID-19 situation. As a result, this step is included in our Technical Plan section.

One of the other steps needed to be taken for the camera design is to confirm the video resolution and frame rate configuration. Tests from last semester are included in the tables below, but the results illustrate that there is a correlation between the amount of motion in a video and the storage size for that video. In other words, videos that record a static environment require less storage space due to video compression techniques.

	Resolution		
Frame Rate	320 x 240p	640 x 480p	1280 x 720p
10 fps	60.37 GB	130.92 GB	329.76 GB
15 fps	63.60 GB	132.97 GB	262.63 GB
20 fps	62.69 GB	120.95 GB	298.00 GB
25 fps	58.22 GB	114.28 GB	425.48 GB
30 fps	54.63 GB	105.21 GB	414.95 GB

Table 1: Video Storage for 30 Days of Continuous Recording

	Resolution (640x480p)	
Frame Rate	No Motion	Motion
10 fps	144.76 GB	274.47 GB
15 fps	147.28 GB	322.13 GB
20 fps	121.80 GB	370.60 GB
25 fps	113.14 GB	416.53 GB
30 fps	143.18 GB	425.98 GB

Table 2: Video Storage for 30 Days of Continuous Recording (Motion Tests)

Based on the collected results from **Table 1**, 480p at 30fps is seen as the optimal combination because it provides the best video quality settings while not exceeding the 256GB micro SD card limit. While the results in **Table 2** suggest that 480p at 30fps will go over the 256GB local storage, it should be noted that the results show the storage size for continuous recording. The final camera system will only save motion-detected clips for local storage, so the required storage per month will be significantly less. It should also be noted that the actual recording environment will more likely resemble the *No Motion* trials. Since the cameras will be mounted on elevated objects such as buildings, trees, and lampposts, motion in the environment (be that people or cars) will encompass a relatively small percent of the overall frame. This means that most of the recorded frame will remain the same, so the recorded videos will resemble the *No Motion* data rather than the *Motion* data.

Since the camera design was not finalized prior to the COVID-19 dismissal, we were unable to perform field testing, so we couldn't get a better idea of the settings to use for the cameras. As such, this step is listed in our Future Plans section as an action we would have wanted to complete. However, assuming the field testing resembled the *No Motion* data, we would configure the camera to record 480p video at 30fps, and the video file format would be set to H.264 (.mp4) at 50% movie quality within motionEye.

Finally, a final design for the raspberry pi camera case was created and printed in the Epic lab.

3.2 Web Interface

During the first half of the semester, web application kept its monolithic structure intact, using Django's MVT structure, the frontend and backend were consolidated into one framework, where the frontend was rendered in html. An immediate change at the start of the semester was to migrate the database from a locally stored database to a cloud stored database using Postgresql. The reason for choosing this relational database was its compatibility with Django using object relational mapping, where database entries can be accessed using classes. In addition to that AWS RDS which hosts it provides easy scalability, so the cost of storing information scales with demand in storage. AWS

Lambda was used to automatically populate the database directly upon a creation of footage in the AWS S3 bucket using a direct database connection. Finally, web development had progressed to the point where it could extract data from the database by selecting a camera and its footage. This all together allowed a user to successfully select data sent by a camera module in real time, as the database was updated.

The most recent development was changing the structure of the web app from a monolithic structure to a decoupled React and Django using client server architecture incorporating bootstrap css into the frontend as well. In this design, React communicates with the Django backend rest api to retrieve and store the information it needs. As Django is written in python and React is written in javascript, Django has a serializer class implemented in order to allow the to communicate using a json format. This allows the frontend to render items dynamically without sending requests to the server to refresh the page. It also makes the user interface look significantly better.

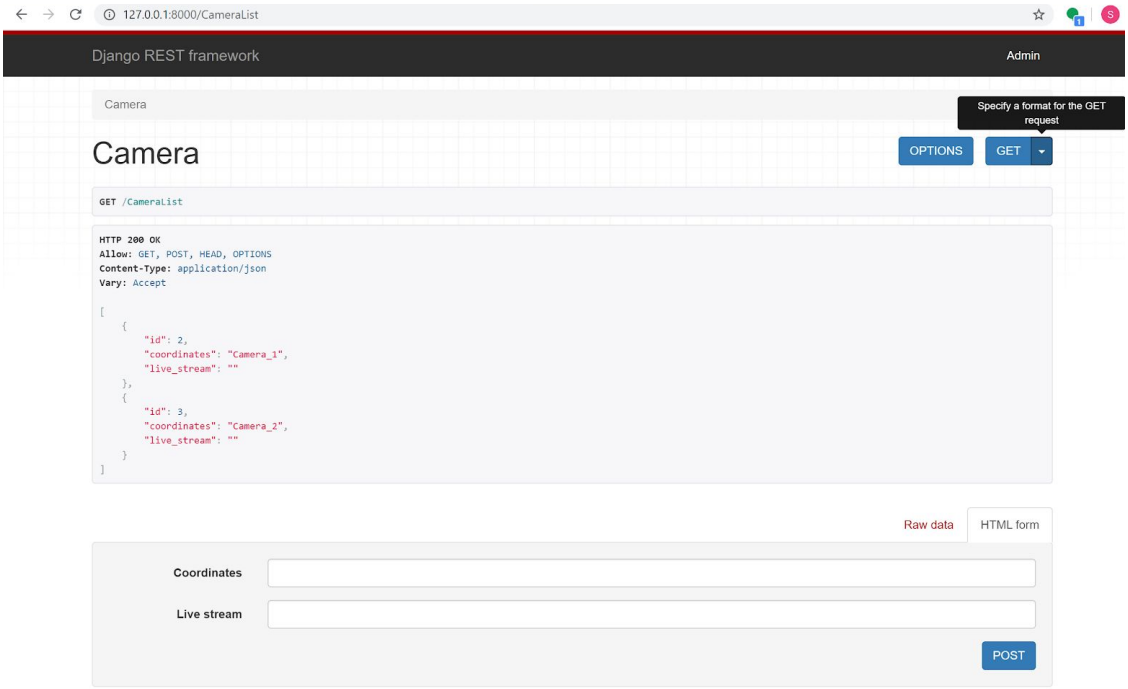

Figure 1. Screenshot of django rest framework in the browser. Django has the ability to test within a browser by sending various requests.

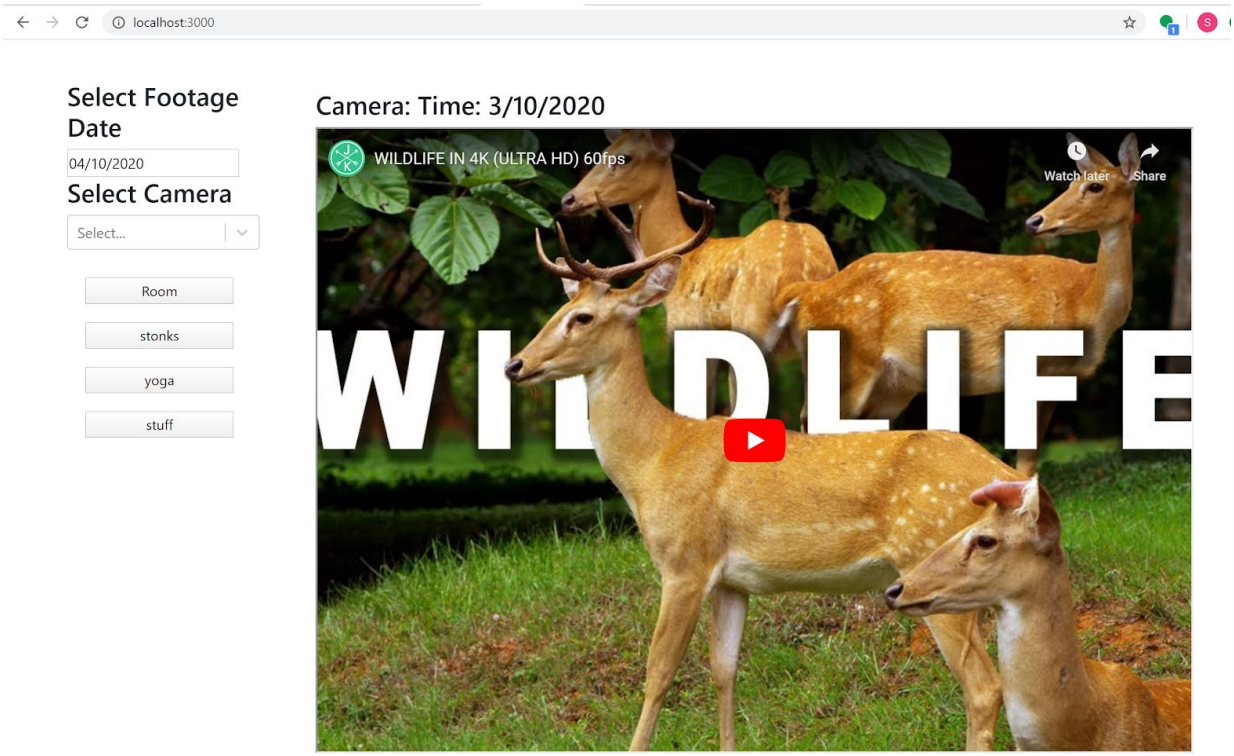

Figure 2. React frontend within a browser with a date, camera selection, and footage thumbtag examples. Due to not having access to the camera modules, links to youtube videos were used as proof of concept.

Finally, the team was in the process of integrating the cryptographic audit trail, however progress on that was halted to due the Covid-19 outbreak, as access to the camera module was lost.

There were additional plans to include machine learning analytics into the web application using AWS Rekognition and Open Alpr. This would have been done by having a lambda function process and mp4, send data back to S3 in a file, which would trigger another lambda function to store the information in the database, which could be extracted in the web app.

Finally, while live streaming capabilities were demonstrated on the camera, integration into the web app was not achieved again due to being unable to access the camera.

3.3 AWS Infrastructure

Our backend infrastructure for this project is hosted on Amazon Web Services, and is required due to the remote and decentralized nature of the project. A major portion of the project requires a web server, and Ubuntu 18.04 LTS running on Elastic Compute Cloud (EC2) was used for this. Video could be synced to AWS S3 storage with the associated API, allowing for near real-time access of data. A trigger in Lambda was used

to update our Relational Database Service, so video access URLs could be retrieved by all servers as they were uploaded to S3. Earlier in the semester, we planned on using AWS Kinesis for live streaming, but ran into compatibility issues with the motionEye program used to capture video. motionEye and Kinesis live stream could not run at the same time, so a livestream hosted on the Raspberry Pi was used instead, so motion-activated video and livestream could be accessed concurrently.

We planned to include more services, such as Simple Notification Service and Cloudwatch for our audit log, but these components were incomplete at the time of project suspension.

As of April 4, 2020, our total cloud services costs are as follows:

Amazon Web Services:

January	TOTAL: $0.00
February	Elastic Compute Cloud: $1.17 Kinesis: $4.45 TOTAL: $5.62
March	Elastic Compute Cloud: $8.80 Kinesis: $11.16 Relational Database Service: $0.93 TOTAL: $20.89
April	Elastic Compute Cloud: $0.03 Relational Database Service: $0.18 TOTAL: $0.21

OVERALL TOTAL: $26.72

Many services fell underneath the AWS Free Tier plan, and had no cost associated. During testing, we were still using the Twilio free trial, but would have switched to pay as you go at $0.10/MB for 4G LTE service.

3.4 Machine Vision

The Machine Vision allows our project to process the video data received from the camera and retrieve important data from it. In order to process our video files we use OpenALPR to retrieve License Plate numbers from passing cars. The machine vision is set up using the Raspberry Pi to upload video files to the ALPR cloud server which then processes the mp4 files. On the Raspberry Pi we run a python script that uploads video files based on whether there is a car in the video frames, by doing this we limit the number of uploads needed. Once uploaded, the ALPR server retrieves the car's plate

number, model, type, and how accurate it is. After it's processed the data is then stored in a data file and sent back to the Raspberry Pi.

After we installed the machine vision we ran several test clips in 240p, 480p, 720p, and 1080p to find the ideal video quality for this project.(Note: test clips were taken from city traffic cameras)

Video Quality	240p	480p	720p	1080p
Avg. Accuracy	68%	76%	82%	87%

From the test clips, the most optimal video quality is 480p based on cost and effectiveness. This will give our machine vision a higher chance of accurately detecting plate number while not going over budget.

4.0 Technical Plan

Looking at the various tasks we completed in the Spring semester and what needed to be completed in the later part of March we deduced this set of tasks that need to be finished in order to create a final prototype.

Task 1. Complete User Interface
Complete the web application user interface which allows a user to view specific types of video footage from selected cameras by clicking on links as well as look at analytics.

Task 2. Log User Interaction with Data
Enable integration of AWS CloudWatch logs with the management platform. This will include the partitioning of data using AWS security groups, as well as filtering and formatting of relevant information.

Task 3. Handle Livestream requests from User Site
Give users the ability to start a live stream on a specific camera, and use the zoom.it API to connect the target camera to our website over HTTP.

Task 4. Integrate AWS Rekognition
Use AWS Rekognition ino order to identify people, cars, and street activity to develop aggregated or clip-specific analysis that will add value to the platform.

Task 5. Log User Machine Analysis Requests
Integrate CloudWatch and CloudTrail to determine usage and access to analytics data.

Task 6. Adding and Removing Users From Community
In order to fully realize the community side of the project it's important to flesh out the process for adding and removing members from community access groups within the webpage and in the AWS security group.

Task 7. Adding and Removing Cameras from Community
Besides members, community cameras must be easily and securely added/removed and is also very important to overall functionality of the member webpage.

Task 8. Test Product on Cummington Mall
Combine all electronics and casing into one unit, and mount the camera system to a location on Cummington Mall. Have the unit record video for a week, and make necessary adjustments to the system as needed.

Task 9. Replicate Camera to Form Network
In order to imitate a small community it's important for user’s to be able to access multiple cameras and it creates a more fully realized end product.

5.0 Budget Estimate

Hardware Costs

Item	Description	Cost
1	Raspberry Pi 3B+	$35.00
2	Raspberry Pi 3B+ Case	$10.99
3	Raspberry Pi 4	$47.69
4	Raspberry Pi IR-CUT Camera Module	$25.99
5	256GB Micro SD Card	$38.65
6	SIM7100A Cellular Transmitter	$67.96
7	Raspberry Pi 3G-4G/LTE Base Shield V2 - with short header	$39.00
8	Quectel EC25 Mini PCIe 4G/LTE Module - EC25-A	$54.90
9	LTE Full Band PCB Antenna – u.FL Plug – 100mm	$6.50
10	TalentCell Lithium Ion Battery	$65.00
11	Power Meter Monitor	$19.99
12	25 W 12V Solar Cell	$59.96
13	35W Monocrystalline Solar Cell	$40.55
14	Charge Controller Module	$17.99
15	USB 2.0 A Male to Micro B Cables	$7.31
16	Micro HDMI to HDMI Cable Adapter	$7.99
17	USB Type-C to USB 2.0 A Male Cable	$6.99
	Total Cost	$552.46

AWS Budget and Resource Estimates

AWS Service	Specs	Quantity	Cost/Unit	Estimated Cost	Description
Elastic Compute 2 (EC2)	t2/t3.large - 2 CPU, 8 GB	732 hr (1 month)	$0.0928/hr	$67.93	For hosting web server
S3	Standard storage	1000 GB	$.023/GB	$23	Video storage, 30 days
Kinesis Stream	30 records/sec - 4 KB/record	1 shard	$12.09/shard	$12.09	From Amazon estimate of 7.5 MB/min for video streams
Glacier	Long Term storage	1000 GB	$0.004/GB	$4.00	Long term storage
SES	Negligible	0	$0	$0.00	Simple Email Service
SNS	Negligible	0	$0	$0.00	Simple Notification Service
IoT Core	Negligible	0	$0	$0.00	Connectivity, messaging, shadow operations
VPC	TBD	0	$0	$0.00	Virtual Private Cloud
Route 53	TBD	0	$0	$0.00	DNS
Cloudfront	TBD	0	$0	$0.00	Media delivery - similar to kinesis
			TOTAL:	$107.02	

Cellular Variable Costs

Twilio 4G LTE Cellular Data	$0.10/MB

6.0 Appendix 1 – Engineering Requirements

Team 24 Team Name: Community Security

Project Name: Community Managed Security System

Requirement	Value, range, tolerance, units
Case dimensions	6 in x 4 in x 3.5 in
Case protection	IP65 rating
Camera Weight	< 6 lbs
Power Source	Solar Cell Optional: Power over Ethernet
PV Cell Sizing	2 ft x 1 ft x 5 in
PV Cell Weight Limit	< 8 lbs
Data Transfer	Cellular Data 4G LTE
Local Storage	30 days of local storage
Cloud Storage	30 days on AWS S3, 1+ years in AWS Glacier
Sustainability	24/7 operation for a full year without maintenance
Livestream Latency	< 1 minute
Camera System Budget	< $1000 per Camera
Audit Log	Only Read Access No Ability to Write to file
Machine Vision	Detect: People, Cars, License Plate Numbers Trigger event when detection occurs
Camera Module	≥ 5 Megapixels Night Vision Capabilities

From these requirements we were able to create a fully functioning, solar-powered, cell-equipped remote camera module. The module includes a Raspberry Pi, camera module, and cellular shield, along with an IP 65 case (that was printed at EPIC but never picked up due to COVID-19). Additionally, we could equip the camera module with a solar cell and battery for remote deployment. Our backend infrastructure is primarily comprised of AWS EC2, S3, Relational Database Service and Lambda. Our server runs Django-Apache, with React and Bootstrap for UI.

Bibliography

[1] J. Clement, “U.S. online users who feel their data is vulnerable to hackers 2019 l Statistic,” *Statista*, 01-Aug-2019. [Online]. Available: https://www.statista.com/statistics/972911/adults-feel-data-personal-information-vulnerable-hackers-usa/. [Accessed: 10-Dec-2019].

[2] L. Matsakis, “The WIRED Guide to Your Personal Data (and Who Is Using It),” *Wired*, 19-Feb-2019. [Online]. Available: https://www.wired.com/story/wired-guide-personal-data-collection/. [Accessed: 10-Dec-2019].

[3] Randy, “Renogy 200W 12V Solar Panel Monocrystalline Off Grid Starter Kit with 30A Wanderer Charger Controller,” *Walmart.com*, 16-Jul-2018. [Online]. Available: https://www.walmart.com/ip/Renogy-200W-12V-Solar-Panel-Monocrystalline-Off-Grid-Starter-Kit-with-30A-Wanderer-Charger-Controller/118614164. [Accessed: 10-Dec-2019]. Figure (2)