

Smart Environmental Monitor and Cloud Streaming

A dissertation submitted in partial fulfilment of
the requirements for the degree of
BACHELOR OF ENGINEERING in Computer Science
in
The Queen's University of Belfast
by
James Hanna
21/05/2023

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER
SCIENCE**

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name:	James Hanna	Student Number:	40262681
Project Title:	Smart Environmental Monitor and Cloud Streaming		
Supervisor:	John Bustard		

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at <http://www.qub.ac.uk/cite2write/introduction5.html> and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

Student's signature

James Hanna

Date of submission

21/05/2023

Acknowledgements

I would like to thank John Bustard for all the help he has given me over the course of this project and always being welcome to provide guidance and support whenever I need it most. I'd also like to thank my family and friends for everything they've done for me over the years.

Abstract

This project demonstrates the capabilities of internet of things (IoT) devices to monitor human environments and how data captured can be cloud streamed and displayed in real time. The project's intended use is educational, demonstrating what's possible when IoT devices are used in tandem with each other for developers interested in their potential functionalities. The project showcases facial detection and recognition running concurrently across multiple cameras, cloud streaming and data clustering.

The project investigates the use of different facial recognition models, backend detectors and target sizes best for IP Cameras, as well as eps values best for clustering algorithms.

Table of Contents

1.0 Introduction and Problem Area.....	7
1.1 Problem Description	8
2.0 System Requirements and Specification	10
2.1 System Description	10
2.2 User Characteristics	10
2.3 Functional and Non-Functional Requirements	12
2.4 Solution Overview	14
3.0 Design	16
3.1 Architectural Design.....	16
3.2 User Interface Design	18
3.3 Concurrency and Data Persistence	23
3.4 Event handling and Error handling	24
4.0 Implementation.....	25
4.1 Choice of Implementation Languages and Development Environments	25
4.2 Software Libraries	26
4.2.1 Libraries used within the Camera Service	26
4.2.2 Libraries used within the Backend Service.....	28
4.2.3 Libraries used within the Frontend Service	30
4.3 Implementation of Components and Important Functions	31
4.3.1 Implementation of the Camera Service	31
4.3.2 Implementation of the Backend Service	33
4.3.3 Implementation of the Frontend Service	38
5.0 Testing	41
5.1 Testing of the Camera Service.....	41
5.1.1 Testing of the Facial Detection Algorithm	41
5.1.2 Testing of the Facial Recognition Algorithm	42

5.2 Testing of the Backend Service	43
5.2.1 Testing of the Clustering Algorithm.....	43
5.2.2 Unit Testing of the Backend Service	44
6.0 System Evaluation and Experimental Results.....	45
6.1 Ability to Meet Requirements	45
6.2 Experimental Results	46
6.3 Weaknesses	47
6.4 Steps taken in the Future	47
6.5 Conclusion	48
7.0 Appendices	49

1.0 Introduction and Problem Area

Internet of Things (IoT) is an umbrella term that covers all devices or “things” that are connected to the internet, that collect and exchange data with other devices over the internet.

As of 2021, there were recorded to be 21.7 billion connected devices with 11.7 billion of those devices record to be IoT devices, making up 54% of all devices connected to the internet, making up the majority of all internet-connected devices. [1][2]

IoT devices come in all different shapes and sizes and can range from being small things such as smartphones or smartwatches to being more advanced features used in industrial projects. IoT devices have the capability to transmit and process data without human intervention making them extremely malleable in their use cases. IoT devices can be broken down into many different types the most common being consumer IoT devices, commercial IoT, Military, Industrial IoT, and Infrastructure IoT. [2]

IoT devices have a history dating back to their invention in the 1990s, but their explosive rise in popularity comes down to advancements in other fields of technology such as sensors becoming more reliable and affordable than before, cloud platforms increasing in capabilities and commercial availability as well as advances made in the field of machine learning. [1]

The reason cloud computing was so important for IoT devices is because it was the first time there was a scalable and cost-effective place that could store and process the vast amounts of data, where previously IoT devices would have limited storage and processing capabilities but now information can be stored, processed, and analysed remotely which made IoT devices more efficient and cost-effective. This gave opportunity to companies to improve and create products as they could now afford to do so, but cloud computing was not always cost-effective for a consumer. [3][10]

This project is focused on solving the problem many smart security camera providers have, which is not providing adequate functionality when it comes to labelling in facial detection with the knock-on consequence of not being able to give movement histories for labelled individuals or tailored alert messages for their detection.

The goal of this project is to monitor a human environment using smart remote cameras capable of detecting individuals using facial detection and facial recognition, to track the movement of individuals throughout a building, and provide live tracking and movement histories of people both known and unknown. The system should be capable of identifying and labelling new individuals and

not just those trained prior to runtime. The system should provide functionality missing from many smart companies' products, demonstrating the potential possibilities in the future of smart home security. The project aims to utilize cloud streaming, storing all processed information in the cloud. The project also aims to investigate how different facial recognition models perform when tasked with real-time processing of faces for identification and detection purposes, and also to investigate how clustering algorithms can be used to separate out data on unknown individuals who are not known prior to runtime.

1.1 Problem Description

With such widespread use of IoT devices, they are capable of providing more control to the environments we surround ourselves in from our place of work to the homes we live in. IoT devices such as smart thermostats, smoke detectors, security cameras, water monitors and lights are examples of commercially available products which can be used to monitor our homes and may be accessed and controlled directly through your phone. [4]

While these capabilities are impressive and do provide excellent control over our homes, they can often times come with additional costs such as subscriptions or one-off payments for better features when the base product does not provide wide enough coverage for what users want. One prevalent example is with Ring products, at home cameras which can be used for security, monitoring package deliveries or monitoring children. [4]

What Ring cameras lack within their features is the ability to identify the individual without needing to open the camera. This feature is only available after purchasing their Ring Protect Plan subscription costing \$3 per month which will identify individuals as being "Unknown Person" or "Familiar Person" which in itself is not a completely satisfactory feature for many people. In addition to this the subscription cost increases with multiple devices making it unaffordable for many people wishing for that style of product. Furthermore, this feature in particular is only available with certain versions of their cameras meaning people will need to spend more on the base product before they can even acquire the control they want for their home. What is worse, this feature's approach to labelling is extremely vague and does not provide a great insight into who an "Unknown Person" might be or who a "Familiar Person" is, making it difficult to know who is being identified without having to manually check each time which can be extremely time consuming. [5]

There are several other companies who service the same smart home security cameras as Ring, the biggest of which being Nest Cam, Arlo, Wyze Cam and SimpliSafe. Out of these four, Nest Cam and Arlo are the only ones to also provide any facial recognition features but, in both cases, it does not provide the feature to all cameras, and similar to Ring, Arlo does not provide it as a base feature and must also be paid for through a subscription.[7] The next two biggest competitors, Wyze Cam and SimpliSafe do not use any facial recognition at all. [6] [8] [9]

The problem this creates is there does not appear to be a satisfactory product which will detect and identify individuals while also providing a higher level of labelling for the detection. The next problem it creates is a divide between smart camera products which will allow for a level of facial recognition and detection and those that will not. It is also worth noting that you cannot mix and match smart cameras from the various providers, as these providers force a user to interface with their products through their interface alone. This forces consumers to use one camera provider or face not being able to have complete control of their home environment from interface.

As cloud computing advances, it is now cost-effective enough for a consumer to use and has the potential to enable smaller developers to build better features using IoT devices that larger corporate companies have overlooked. [11]

2.0 System Requirements and Specification

This section will contain the specification and system requirements for the complete project, split into the following sub-sections:

- System description will as a brief overview of the system and how it will operate.
- User characteristics will consider the people expected to use the system and what their potential use cases would be for it.
- Functional and non-functional requirements will take into consideration the use cases of the potential users and break down the functionality required to achieve those use cases for the system to operate as a legitimate product.
- Solution overview will consider and discuss if the system with its functional and non-functional requirements would be capable of solving the problems addressed within section 1.1 and to what extent it solves them.

2.1 System Description

This system is a series of smart remote cameras, each running on its own thread with their video feeds being processed by a facial detection model. The facial detection model forwards detected faces to a facial recognition model to label individuals detected within each frame of video. The data processed by these models is then sent to a backend flask service which will further process the data before storing it in a cloud database through the use of MongoDB, a multi-cloud developer data platform. The end user will be able to see all information stored through a frontend website which will query the flask backend, which in turn will query the MongoDB database. This frontend will allow an end user to utilize all features developed from one location and will give them a concise and easy to follow breakdown of all information captured.

2.2 User Characteristics

This product has two users. The first of which being homeowners and small business who are looking for an additional level of security, and the second of which another being parents looking to monitor their children. These individuals will only need to interact with the frontend of the system and will not need to worry about any of the setting up involved with the facial recognition or clustering algorithms. All of these target users will have similar functional and non-functional requirements as each of which may be interested in the current location of individuals or creating custom alerts. These target users are considered to be the end users.

An additional target user would be developers, who would be responsible for maintaining and developing the system throughout its lifetime. A developer would be capable of interacting and working on all areas of the system but should not be considered as one of the end users. A developer's functional and non-function requirements will be quite different from other target users as they will be less interested in using the features developed and more interested in developing new features throughout the duration of the service and ensuring that the system is developed with a well-designed code-structure.

Throughout the development cycle of the project, the intentions are to meet both target users' interests by creating user stories tailored to them as a way of making sure that the system will meet their needs. To simplify the table, we shall classify end users as consumers.

ID	User	User Story
1	Developer	As a developer I would like to be able to develop features quickly and without having to make major changes to other areas of the system to do so
2	Consumers	As a consumer I would like the frontend of the system to be easy to understand and be self-explaining
3	Developer	As a developer I would like the code to follow convention so that it remains easy to understand key details of the system
4	Consumers	As a consumer I would like the frontend to be bug free to provide a clean user experience
5	Developer	As a developer I would like the code to be well documented so that it will be effortless to onboard a new member of the development team and avoid situations where nobody can understand how the code operates after completion
6	Consumers	As a consumer I would like to be able to use the feature from any web browser
7	Developer	As a developer I would like compartmentalization throughout the system to clearly break up the system into different areas, responsible for key services of the system meaning if one area breaks it will not cause a knock-on consequence of breaking other areas of the system
8	Consumers	As a consumer I would like to be able to fully utilize features without any difficulty
9	Developer	As a developer I would like code broken up into functions to create a culture of code reuse and to ensure that if one thing breaks within a particular area of the system it will not have a consequence of breaking

		other things within that area
10	Consumers	As a consumer I would like to be given informative and explanatory error messages if I use a feature incorrectly so I may rectify it
11	Developer	As a developer I would like the ability to test key areas of the system in isolation from other areas of the system
12	Consumer	As a consumer I would like clear compartmentalization on the frontend for where the features are and where the informative sections are
13	Developer	As a developer when adding a new feature to the frontend I wish it to be clear where the feature belongs, having a clear and logical position illustrated by the location of the other features already developed

2.3 Functional and Non-Functional Requirements

The system will have several functional requirements and several non-functional requirements. From an end user perspective an important functional requirement is that the frontend should be capable of querying the flask backend for information regarding an individual. This is important because it is necessary to provide the end user the ability to query the movement histories of individuals detected by the system. Another functional requirement would be to ensure that all features developed can be accessed by the end user through the frontend. With regards to the frontend, a non-functional requirement would be that the frontend should be user-friendly and responsive to ensure that end users feel satisfied with the quality of the product and do not feel hindered in anyway while using it.

With respect to the flask backend, the first functional requirement is that the flask backend should be capable of querying the MongoDB database. This will be imperative to developing and maintaining all the different features. Another important functional requirement is that the flask database must be capable of receiving information from the facial identification service as without this none of the features remain possible. Additionally, another functional requirement is that each endpoint should have appropriate error handling and should transmit appropriate error messages to the frontend whenever appropriate to be displayed to the user. In terms of non-functional requirements, the backend should maintain high availability and should have low latency to ensure that the system feels responsive to the end users when they use it on the frontend and does not feel like there is any delay between the detection of an individual and the time it takes to display the detection on the frontend service. Another non-functional requirement would be that the cloud database is scalable and secure to protect, not only to make sure that the project avoids growing pain processing vast amounts of data but also to protect the data of the end users in partially the images stored in the MongoDB. Furthermore,

another non-functional requirement is that the backend service should be easy to maintain, modify and extend in the future when developers create additional features for the system.

With respect to the facial identification service, the first functional requirement should be that the remote cameras used should be able to transmit their video feed over the internet as the feed is needed for processing. The next functional requirement should be each camera is capable of running concurrently with one another on its own thread as to ensure that the system can process frames on each thread, ensuring that the system can continue to run in real time. The next functional requirement is that the facial detection and facial recognition is able to label individuals within each frame of video without delay. A non-functional requirement of the facial identification service is that the facial detections and identification should be accurate and efficient. Another non-functional requirement is that the service should be scalable, meaning it must be able to cope if more cameras are connected. An additional non-functional requirement is that the service should be reliable and have minimum downtime. To summarize this section, the following table has been provided:

Service	Functional Requirement	Non-Functional Requirement
Camera service	Remote cameras can transmit video feed over the internet	Facial identifications and facial detections should be made accurately and efficiently
	Each camera video feed is processed on their own thread	The service should be scalable, allowing for more cameras to be connected without difficulty or loss in performance
	Facial detection and facial recognition are able to label individuals in each frame without delay	The service should be reliable with minimum downtime
Backend	The service should be able to query the MongoDB	The service should have high availability with minimum downtime
	The service should be able to receive and process information received from the camera service	The cloud service is scalable and secure
	The service should have sufficient error handling	High maintainability with ease

Frontend	Service capable of querying the flask backend service for required features	The service should be user-friendly and provide a seamless experience
	All features developed should be accessible to the end user through this service	The service should be responsive and fast, loading quickly and without hesitation

2.4 Solution Overview

The project's aim is to investigate and solve the problems outlined within section 1.1 by building a superior service. The main problem outlined in section 1.1 is the inability of smart cameras to provide complex labelling beyond the generics of "Familiar Person" or "Unfamiliar Person". To solve this issue, we can pretrain our facial identification model within the camera service to be capable of identifying and labelling known individuals and forward this information to the backend service. In addition to this, each detection will have the image of their face converted into a facial embedding, a multidimensional vector representing their face which will be sent to the backend and stored within the MongoDB. The purpose of this is in case an unknown individual is detected by the facial identification model; in which case the embedding of the unknown individual will be compared against the known individuals' embeddings using a clustering algorithm to determine the identity of the unknown individual. If the clustering algorithm also cannot determine an identity, then the end user will be able to provide one through the frontend service, where they will be presented with an image of the individual. After labelling the cluster algorithm will identify all similar unidentified facial embeddings and label them with the newly labelled facial embedding. With this approach, we are able to have a much more comprehensive labelling technique which will be capable of providing far greater detail than other smart camera providers. As noted in section 1.1, facial recognition and identification is only possible on certain models of cameras whereas with this project it will be designed to be possible for all remote cameras that can connect through an IP address.

The next problem outlined in section 1.1 is not being able to mix and match remote cameras and control all of them from one location. To fix this issue, we will develop a frontend website which will allow for all remote cameras regardless of type to be accessible through along with all developed features being accessible from within the same frontend interface. As a consequence of using more comprehensive labelling methods, additional functionality can be developed such as movement histories which many smart cameras service do not have. This feature will be available through the frontend interface.

Furthermore, with many providers not having comprehensive labelling, alerts are typically generic. Our solution to this using superior labelling, we will provide alert messages tailored specifically to individuals detected and to the rooms they enter. These alert messages will be sent through Discord, a popular communications platform used for voice and text communication. This will be achieved through the use of developing a Discord bot which will be able to interact with the Discord API to forward tailored messages to a text channel.

3.0 Design

3.1 Architectural Design

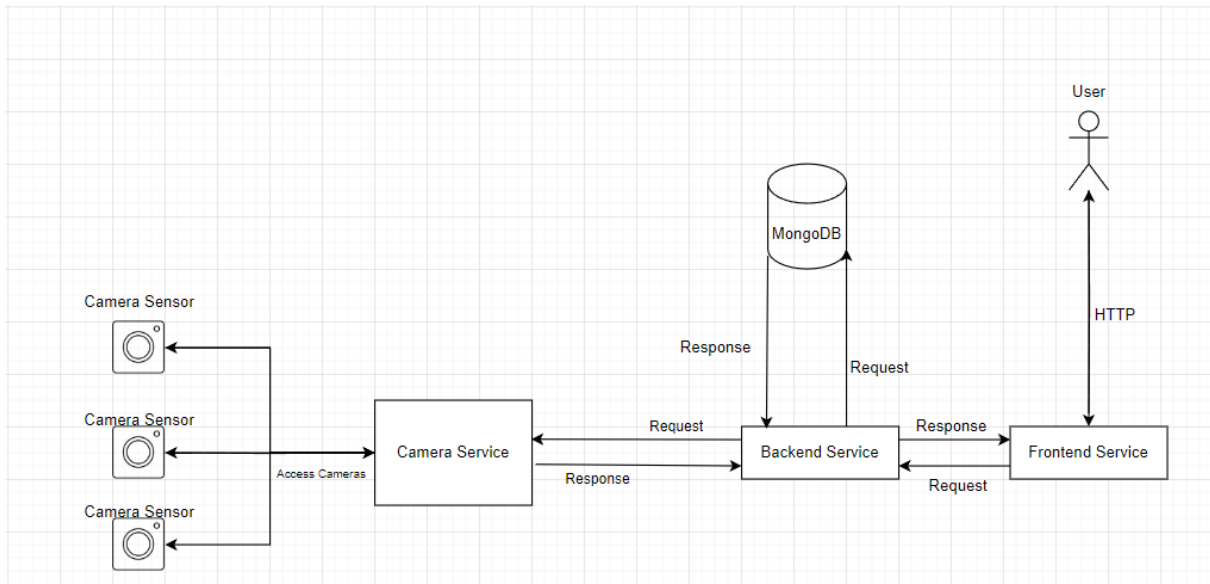


Figure 1

Figure 1 as seen above represents the architectural design of the system. The architectural design above showcases how each of the different services will be connected to one another. Each service will communicate with each other by sending requests, forwarding data in the process, and then awaiting a response. The design above demonstrates how multiple cameras can connect to the same camera service through each camera's unique IP address. Once the camera service has finished processing a frame using the facial recognition and detection models, to obtain a label and a facial embedding, the collected data is then forwarded to the backend service by creating a POST request to it.

After this has happened, the endpoint will process the data received from the camera service further, before forwarding it to the MongoDB database to be stored within its collections. This will be achieved through the MongoDB library which underneath sends requests to transfer the processed data and returns responses.

Figure 1 also shows how a user interacts with the frontend service primarily through the use of a web browser. Similar to the camera service, the frontend will interact with the backend service through the use of sending requests and awaiting responses in return. All endpoints will provide JSON responses, as the frontend service will utilize a templating framework which can be used to display dynamic information, in our case the JSON response data.

The end user will be able to add new rooms, cameras or labels using the frontend service. The frontend will forward the details of this to the backend service through a request to a corresponding endpoint dedicated to each feature which will verify the information before returning a success or error message which will subsequently be presented to the end user within the frontend.

The frontend service and backend service will both be designed so they may be deployed in the cloud. This will be for several reasons, the first of which being it will ensure that the services are reliable as cloud services are typically designed to be highly available. In addition to this, the services will become more scalable as additional containers can be ran if there is high demand for either service. Storing the services in the cloud will also have the added benefit of being provided automatic backups in-case of an unforeseen error and recovery states. What is more important, is that cloud services also provide built-in security features such as encryption which will be important in keeping data protected.

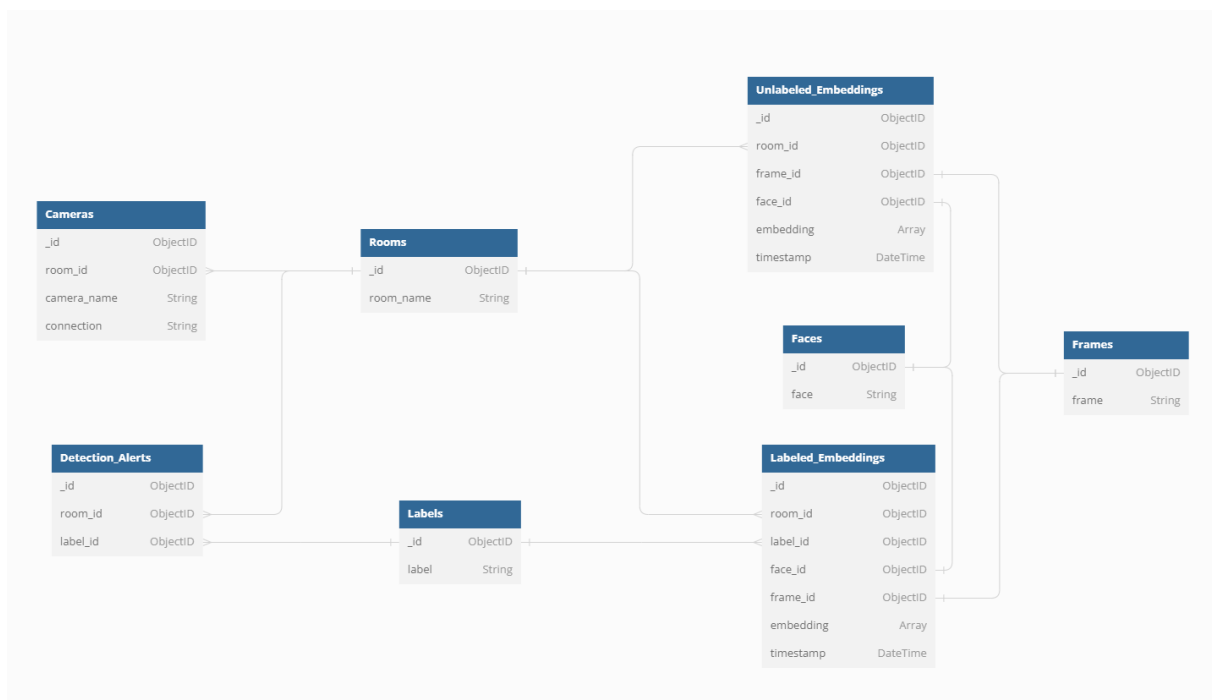


Figure 2

Figure 2 as seen above represents the entity relationship diagram of the MongoDB database stored in the cloud. The tables represent the collections and the lines represent the relationship each collection has with one another. MongoDB uses ObjectIDs as primary keys so we will use them as foreign keys in other tables to forge relationships between the collections.

3.2 User Interface Design



Figure 3

Figure 3 showcases the chosen design for the user interface. The user interface utilizes a blue-grey and amber colour scheme featuring white sub-text as to stand out on the page, being easy to read to any end user wishing to access the system. The user interface lends itself in its simplicity, as it compartmentalizes the different features into sections, the first being features that allows the end user to input information to the system being stored at the top, room specific features being displayed within the rooms own section, while more general features are displayed beneath this. This was an intentional design decision following user research.

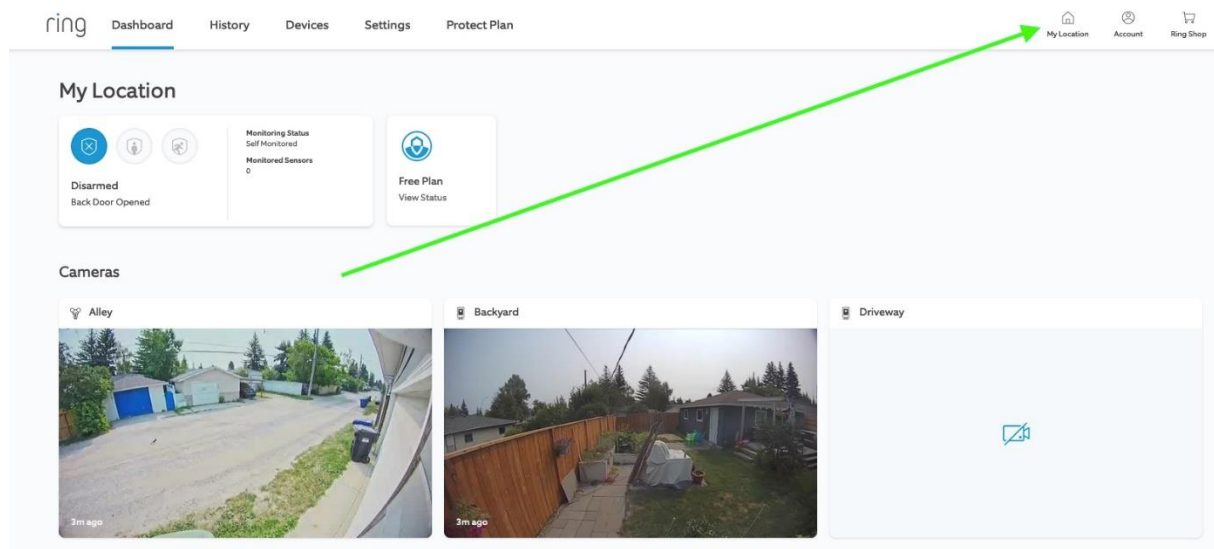


Figure 4 [12]

Figure 4 showcases the frontend of Ring. Ring took the approach of separating out the features into their own tabs rather than featuring them all on a single page. When conducting user research on this by asking people what they thought of the above interface, it was discovered that many people believe that this approach is tiresome and adds additional complexity to something that could be simplified. After this, people were presented the UI design showcased within Figure 3 and asked for feedback. The feedback received focused around its simplicity and ease of use but conceded that my interface does not show the cameras video feed within the same page.

Upon researching UI design principles, the colour amber was chosen as a highlight colour because it would stand out against many darker backgrounds, and using amber as my titles or sub-titles would be understandable to users wishing to use the system. Additionally, blue grey was chosen as amber is said to go well and stand out best with shades of blue. After deciding that the sub-text colour would be white, it was decided a dark shade of blue grey would be used as cold colours such as white, blue, and grey all go together. [13] [14] [15]

Figure 5

Figure 6

Figure 5 showcases the interface presented to the user when clicking the add a room feature and Figure 6 showcases the interface present to the user when clicking the add label feature. Due to their similarity, they shall be evaluated together. As can be seen the forms simply appear in a modal over the top of the website without the need for redirecting the user to another page. With this approach the user can easily return to the main page. These forms consist of a single text field, with the prefilled instruction for the user on what to provide. The forms each have a highlighted button in the bottom corner to submit it upon completion.

Figure 7

Figure 7 showcases the interface presented to the user when clicking the add camera feature. It uses a dropdown input to allow the user to select from all available rooms when adding a new camera to the system. It utilizes the same input text and submit button style as seen in Figures 5 and 6 as to provide consistency to the design, remaining familiar to what the user will be used to seeing elsewhere in the system.

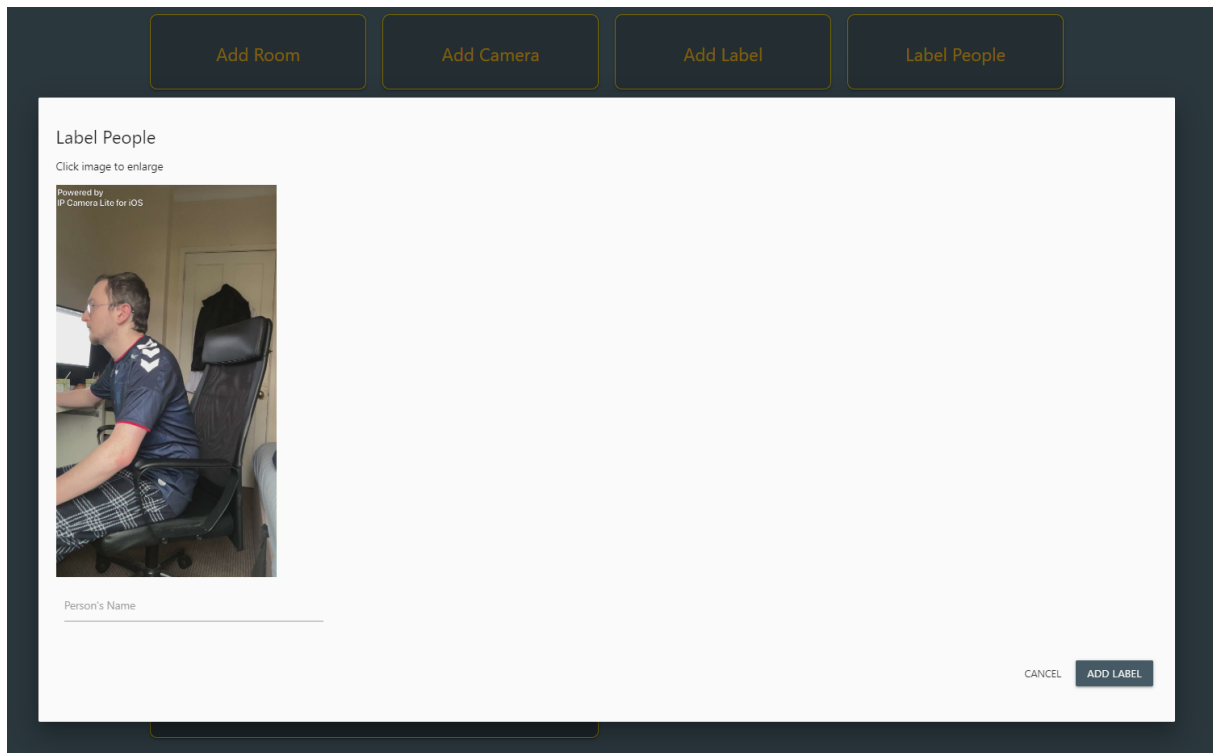


Figure 8

Figure 8 showcases the interface presents to the user when clicking the label people feature. The interface presents the user with a photograph of a frame or picture, alongside a text input field similar to Figures 5,6 and 7 for them to input the label of the person within the photo. This design choice is simple and intuitive for the user to understand and operate while also not taking them far away from the main page where other areas of functionality is, keeping them close. Additionally, the design choice to allow the image to enlarge upon being clicked was made based on it being important for the user to be able to get a detailed view of the person within the photograph.



Figure 9

Figure 9 showcases the Rooms section within the frontend. This section has been designed to contain all information regarding each room in one location so that a user can find information more easily. The first thing to notice is that each camera connected to the room is given a button to view the live feed of the camera. This is provided at the top of the room section for ease of access. The next thing to note is the use of a donut chart, representing the total number of detections an individual has had in the room, also covering unknown individuals. The donut chart was chosen to represent this information because it has good use of space, it is easy to read and provides a good comparison for the data it is representing.

The next design choice within figure 9 was the use of pagination tables. This decision was made for both the detection alert table and the in-room table because in cases where lots of labels exist in the database and lots of detections have been made in the room, it causes the tables to become disproportionately long compared to other components within the section, so using pagination was a stylish and professional technique used to get around this issue.

The final design decision within figure 9 is the use of switches for turning on or off detection alerts for individuals within the room. The decision to use switches came down to the fact that they make it easy for an end user to understand if a switch has been toggled on or off rather than other alternatives such as a check box. Switches provide a label to inform the user of the action of the switch, leaving no room for ambiguity.

Labels	Room Name	Time of Detection
James	Room 1	Mon, 08 May 2023 07:02:42 GMT
	Room 1	Mon, 08 May 2023 07:02:39 GMT
	Room 1	Mon, 08 May 2023 07:02:34 GMT
	Room 1	Mon, 08 May 2023 07:02:33 GMT
	Room 1	Mon, 08 May 2023 07:02:33 GMT
	Room 1	Mon, 08 May 2023 07:02:31 GMT
	Room 1	Mon, 08 May 2023 07:02:29 GMT
	Room 1	Mon, 08 May 2023 07:02:28 GMT
	Room 1	Mon, 08 May 2023 07:02:28 GMT
	Room 1	Mon, 08 May 2023 07:02:27 GMT

First Previous 1 2 3 4 5 ... 41 Next Last

Figure 10

Figure 10 showcases the movement histories feature. The design decision to use a select dropdown which on change populates a table with pagination was made based on its ability to provide instant feedback to the user in a concise way. Similar to the tables in figure 9, pagination was used to avoid tables becoming disproportionately long. As additional space was available the design decision to include additional buttons such as “last” and “first” was made in the case of large sets of information which was less likely to occur in the other tables such as in last detection and detection alert.

3.3 Concurrency and data persistence

The decision to use concurrency was made based on the need to process different camera feeds simultaneously and to increase performance. As the number of cameras increases, using a linear approach would create a situation where the frames could not be processed quickly enough and might lead to the potential scenario of missing a face. Using concurrency avoids this issue and allows the system to send multiple requests at the same time to the backend, which in turn protects the system for a scenario that two individuals are detected at the same time across different cameras. Having a dedicated thread per camera running the facial detection and facial recognition algorithm helps to ensure that no information is lost and the system remains up to speed.

Throughout the system data persistence is present and exists, with all data processed by the backend service being stored in the MongoDB within the cloud. This ensures that when the system is closed all data is kept and will remain accessible the next time the system is accessed by an end user. It also ensures that if something goes wrong with the backend service, the data is kept secure and will remain unaffected, giving an additional level of data security.

3.4 Event handling and error handling

Event handling is used throughout the system but begins in the frontend service when the user utilizes one of the features. The first example of event handling is when the user utilizes one of the features that uses a submit button such as adding a new room, camera, label or providing a label to an unidentified individual. In these scenarios event handling is used to submit the form from the frontend service to the backend service by calling a request to the corresponding endpoint responsible for that feature. Within this endpoint there is error handling in place to ensure that data has been provided, and that the data is correct and satisfactory. If these conditions are not met then the endpoint will return an error message within a JSON response, along with a status code informing the frontend service that something was not correct. This error message is displayed to the user so they may rectify the issue if possible.

The next type of event handling is that which utilizes AJAX functions. When the user clicks to enable detection alerts against a label on the frontend, or when the user selects a label to see their movement histories this creates an event which starts calling an AJAX function to forward that request to the backend service rather than having to refresh the page. The request is received by a corresponding endpoint responsible for that feature which analyses the request data with error handling before processing further and then forwards a JSON response back to the frontend, informing the user if the feature was successful, or provides an error message if an error was captured by the error handling.

4.0 Implementation

4.1 Choice of implementation languages and development environments

For the development of the camera service, the decision to use Python was made because Python is a well-established language featuring lots of clear documentation that is also widely used and widely accepted as being one of the best languages for machine learning projects, particularly projects involving fields of computer vision. In addition to this, Python has lots of different libraries available which could be used to simplify complex aspects of the service such as connecting and retrieving the video feeds from the different cameras and achieving concurrency. Python was also chosen due to the researcher having previous experience with the language, making it a comfortable decision for them to work with something they were already familiar with the syntax of, so not needing to learn something entirely new which might have hindered the development of the service.

As for the development of the backend service, Python was chosen again because the researcher already had some experience setting up a Python service when completing the Cloud Computing module. This module provided context of how to use the Flask library which made setting up and creating a service in Python simple so it was a natural decision to make when picking which language to use for developing the service. In addition to this, the researcher also knew already how to dockerize a Flask service developed in Python from this module so it made complete sense to them that Python should be used for this area of the project. Another thing that impacted the decision to use Python for the backend service was the need to establish cloud streaming within the project. They knew that MongoDB had well documented libraries within the Python language so using Python would be a good language to help establish the cloud streaming necessary to achieve this requirement.

With the development of the frontend service, the researcher made the decision to use NodeJS, a server-sided runtime environment which runs on the JavaScript engine. They had several reasons for this decision, the first reason being that NodeJS has the largest collection of open-source package libraries which are all easily accessible through the use of the node package manager, and with access to these libraries, they knew it would make the development of my frontend service much easier and faster. The next reason they chose to use NodeJS is because it uses JavaScript to write the server-side code and the client-side code meaning they would not need to switch between languages any more than is necessary when creating frontend templates. NodeJS is also well suited for creating microservices which benefited the researcher considering the need to dockerize the service after completion. The researcher also had experience working with NodeJS as well as the templating library Nunjucks used to create the frontend webpages, so they chose to work to their experiences and work with the languages and libraries they were familiar with already prior to starting the project.

4.2 Software Libraries

Throughout the project and the development of the different services, many software libraries have been used to achieve different objectives and requirements. For clarity this section will be separated out for each of the different services and explain what each library is and the used for within the service.

4.2.1 Libraries used within the Camera Service

Software Library	Description and Implementation
cv2	An image processing library that was used to access the video feeds being transmitted by the smart cameras
threading	A library allowing for the facilitation of creating and managing threads within python programs, enabling for concurrent execution. It was used in the project to create threads for each of the camera feeds and facilitated running the facial detection and recognition systems on each of them
requests	A library used for creating HTTP requests and was used to request camera information from the backend service to establish their connections and find out how many threads should be created
json	A library providing methods to operate over JSON data. Was used to deserialize the JSON response sent from the backend service regarding all information on cameras, prior to establishing their connections
deepface	An image processing library specialising in facial recognition tasks. The library provides access to many different models useful for facial detection, facial recognition, and lots more
deepface.DeepFace	A class within the deepface library acting as the primary interface to various facial analytic functions. Throughout the camera service it was

	used for facial detection, facial recognition and creating facial embeddings
deepface.comms.functions	Access to common utility functions within the deepface library. Was used within the service to find the target sizes of images when using different facial detection and recognition models
re	re is a regex library that was used to cut out the label name from a string with additional information attached
base64	Base64 is a library used to encode images into binary data or to decode them from such state. It was used to encode frames or facial areas into binary data so it could be forwarded to the backend service for storing
aiohttp	A library used for creating asynchronous HTTP requests. This was used to forward detection information such as labels, embeddings and encoded images to the backend service and was used so threads could continue processing images without waiting on a response from the backend service
asyncio	A library which provides a framework for creating asynchronous code. Was needed for asynchronous components of the service to function such as with sending asynchronous HTTP requests

4.2.2 Libraries used within the Backend Service

Software Library	Description and Implementation
Flask	A web framework that facilitates the development of web applications in Python. This library was used to create web endpoints which could be accessed through HTTP requests
request	An object belonging to the Flask library. Used to represent incoming requests and was used in the project to access the query parameters of GET requests, and the body of POST requests.
jsonify	A function part of the Flask library used to simplify the process of creating JSON responses from Python objects. Was used within the project to handle the creation of JSON responses which are returned from the flask endpoints at the end of their processing.
Sklearn.cluster	A module from the scikit-learn library, a popular machine learning library which provides access to various clustering algorithms. The project used this to access the DBSCAN clustering algorithm which was used to identify known individuals if the facial detection was not sure enough and to separate data on all unknown individuals.
numpy	A library used for scientific computing. Was used to convert lists coming from the MongoDB database to lists compatible with the clustering algorithm.
Pymongo.MongoClient	A class from the pymongo library used to establish a connection to a Mongo database. The project used it to connect to the cloud database prior to querying it.
Bson.ObjectId	A library providing support for binary JSON encoding and decoding. MongoDB uses ObjectIds as primary keys, so the project utilized Bson.ObjectId to convert strings to ObjectIds for

	querying and updating the cloud database throughout the different endpoint functions.
ast	A library to parse, analyse and manipulate python code. It was used to convert the facial embeddings array, which was translated into a string form in the camera service to be sent through a request, back into an array.
datetime.datetime	A library providing classes and functions for working with dates and times. Used in the project to create timestamps to provide live tracking and movement histories of individuals detected by the system.
Flask_cors	An extension to the flask library that provides Cross-Origin Resource Sharing. This was used to make it possible for AJAX functions on the frontend service to call flask endpoints within the backend service.
Discord	A library for creating discord bots and allow them to message into connected discord servers and channels. The project uses this to connect to the Discord API.
Discord.ext	An extension to the discord library providing additional modules for commands and event handling. Used within the project to develop the code which causes the bot to message a discord channel when a user with a detection alert has been identified.
Thread	A library allowing for the facilitation of creating and managing threads within python programs, enabling for concurrent execution. Within the backend service, it was used to create a thread for the flask application and another thread for the discord bot, as both cannot run on the same thread.

4.2.3 Libraries used within the Frontend Service

Software Library	Description and Implementation
express	A web application framework providing features and utilities needed to build web applications. The frontend service uses this to define routes and handle incoming HTTP requests, responding by rendering appropriate HTML pages.
node-fetch	A library used to simplify sending HTTP requests from a node application. The frontend service uses this to forward requests created by users using different features to the backend service and returns the response sent from the backend service to the user on the frontend.
nunjucks	A templating engine for JavaScript that is used to generate dynamic HTML pages. Within the project it is used to generate the HTML templates shown to the end user by arranging elements based of data sent from the backend service, and additionally it is used to break up different features of the main page into individual template file, to compartmentalize different aspects of the frontend page.
express-flash	A library used to store temporary messages. The project uses this to store success or error messages sent from the backend service after a user uses a submit style feature. The message is sent to the rendered html page using express-flash, where it is displayed
express-session	A middleware module part of express.js framework used to enable session management abilities. Within the project, it is needed for express-flash to function.
Materialize	A frontend framework for web development used to create UI components for frontend webpages. The project uses this to help structure and stylise

	the frontend pages, utilizing the many different components the framework has such as columns and rows, card-panels, and colour options. The framework was also used for their style of input fields, alert messages and modals for which forms were placed in for the different features to avoid having to change page.
Chart	A JavaScript library allowing for the creation of interactive and visually appealing charts. Within the project it is used to create the donut chart features, breaking down the detections of individuals within a specific room.
Datatables	A library providing additional functionality for displaying information within tables. It is used within the project to provide pagination for tables that otherwise could be excessively long in size such as the detection alert tables, currently in room table and movement histories table.

4.3 Implementation of components and important functions

The project revolves around three main components, namely the camera service, backend service and the frontend service. Below we discuss how each of these components was implemented, highlighting the implementation of the key functions required within the services.

4.3.1 Implementation of the Camera Service

The implementation of the camera service starts by sending a GET request to the backend service to retrieve the connection, and room_id of all available cameras stored in the cloud.

After this has happened, we use the videoCapture function from the cv2 library on each of the connection addresses, creating each camera a VideoCapture object responsible for accessing their video feeds. Later, each VideoCapture and room_id corresponding to that camera are used to initialize new camThread objects, the instantiations of a class which extends the threading.Thread class, used to represent the threads responsible for each camera preview and facial recognition. Within the

camThread class, the run method is the entry point for the thread to begin processing. The camThread class inherits the run function through the threading.Thread class but we override this function so that we can call the camPreview function responsible for performing facial recognition and detection analytics. This is done by utilizing the asyncio.run() method within the asyncio library and passing the camPreview function as a parameter. What this achieves is when all threads are ready to begin processing, all of their run methods are called causing the camPreview to run across the threads, processing all smart camera feeds.

The camPreview function is the most important function within the camera service. It begins by creating a loop asserting that the function will not stop until the video feed terminates. Within this loop it captures the live frame by using the read method on the VideoCapture object responsible for the camera feed. Once the frame has been obtained, it is then passed to the extract_faces function belonging to the DeepFace class which in itself is part of the deepface library. This method is used to detect the location of faces within images and can be used to co-ordinate the location of faces if any within the passed frame. Along with the frame, the extract_faces take several other parameters such as target_size, a variable representing the desired size of faces from the input image after they've been resized and detector_backend, a parameter used to determine the underlying library used to perform facial detection. Both of these parameters were chosen through after performing testing experiments which are later discussed in section 5.1.

Once the extract_faces method has completed, if any faces are detected it will return the co-ordinates of their locations within the frame. From here the camPreview function loops over the detected faces within the frame and passes a crop of each face to DeepFace's find function. This function uses machine learning to compare the face to photographs of known individuals stored within a local database and provides a prediction of who it believes the individual to be with a confidence score in its prediction. In addition to the face crop, the find method has other parameters. It expects the local database of labelled images to be passed so it can compare the cropped face to them, and the names of the facial analytics model and backend detector wishing to be used to do so. Similar to the extract_faces model the model and backend detector chosen were done so after testing to determine which model and detector are best for this project, which is discussed in section 5.1.

The find method returns a data frame containing the labels for the images it believes have the most similarity to the cropped faces, accompanied by confidences scores for such predictions. If a prediction exists within the returned data frame, then the camPreview sets up to create a POST request to the backend service, informing it that an individual has been detected. Before it can do this, the represent function from the DeepFace class is used to convert the facial image to a multi-dimension vector which can later be used within the clustering algorithm in the backend service. The embedding

and `room_id` is added to a dictionary which will be forwarded in the POST request. Next, the function determines if it should provide the frame the face has been detected in and if it should provide the cropped image of the face to the backend service. This is determined by checking how many frames have passed since a face or frame image were sent to the backend service for storing. If it is appropriate to forward a frame or face to the backend service, then the frame is encoded to base64 using the base64 library and the resulting string is added to the dictionary.

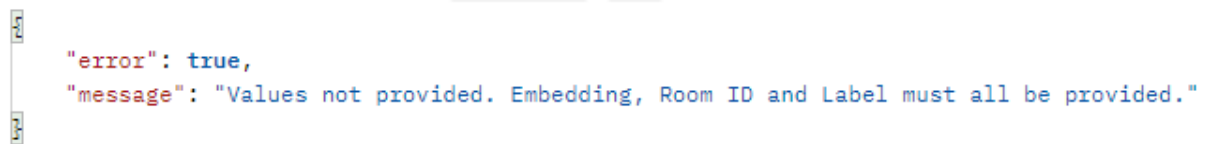
The next stage of the function is determining if the confidence score in the prediction is high enough to be considered a labelled detection or an unlabelled detection. This is calculated by checking the prediction's confidence is above a certain threshold calculated through testing and experiments discussed in section 5.1. If it is confident enough to be considered labelled, then the label of the individual is taken from the image file location passed in the data frame using regex functions belonging to the `re` library. The label is then added to the dictionary ready to be passed to the backend service with the other parameters. In both cases, a POST request is sent using the `ClientSession` function belonging to the `aiohttp` library either to the `labelled_embeddings` endpoint or the `unlabelled_embeddings` endpoint respectively. `Aiohttp` was chosen instead of `requests` for this because it is asynchronous meaning that the function would not need to await a response which is particularly needed when forwarding the base64 strings, representing the frame of face images.

4.3.2 Implementation of the Backend Service

The implementation of the backend service begins with the setup phase. In this phase we must connect to the MongoDB database using the `MongoClient` object which is retrieved from the `pymongo` library. The next step is to setup a discord bot object which will be later used within the detection alert feature. This is achieved by utilizing several functions from the `discord.py` library, establishing a default bot. The final stage of setup is to establish a Flask object which will be used to create the different endpoints acting as entry to our features. This is done by using the `Flask` class which is part of the `Flask` library and lastly using the `CORS` function from the `flask_cors` library, we pass the created flask object to it so that cross-origin resource sharing is enabled allowing AJAX features on the frontend to call the backend services.

Now that setup is complete, we can begin defining the endpoints. Each endpoint is defined using a decorator which looks like the following `@app.route('/example_route')`. What this does is informs flask that when a client sends a request to the `/example_route` route, the function declared beneath the decorator will be called to handle that request. This format is used consistently throughout the service.

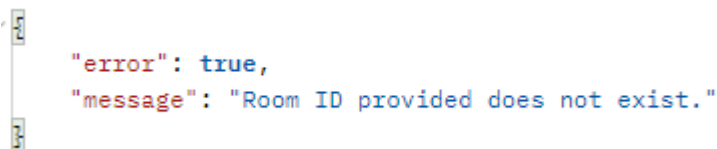
In total there are 15 endpoints but generally most are used for adding, updating, or removing information from a certain collection. With that being said, the key endpoint functions for this service are “/labelled_embeddings/add”, “/unlabelled_embeddings/add”, “/labels/track”, “/labels/update”, and the root endpoint defined simply as “/”. “/labelled_embeddings/add” and “/unlabelled_embeddings/add” are used by the camera service to forward data gathered during the detection of an individual to the MongoDB. Each endpoint begins by checking that the expected data has been passed. This is achieved through the use of the request class belonging to the Flask library. If all required variables are passed then the functions continue but, in the case, they are not, a JSON response is returned informing that an error has occurred with an explanation that all variables must be passed, along side an error code of 400. This is achieved by using the jsonify class belonging to Flask.



```
{
  "error": true,
  "message": "Values not provided. Embedding, Room ID and Label must all be provided."
}
```

Figure 11

Assuming all variables are passed the function continues by checking for the existence of key variables, checking that a room exists within the MongoDB database by searching for a room within the rooms collection, using the room_id sent as one of the arguments to the endpoint. If a room does not exist matching the provided ID, then the function does not continue and returns a JSON response with jsonify explaining the issue.



```
{
  "error": true,
  "message": "Room ID provided does not exist."
}
```

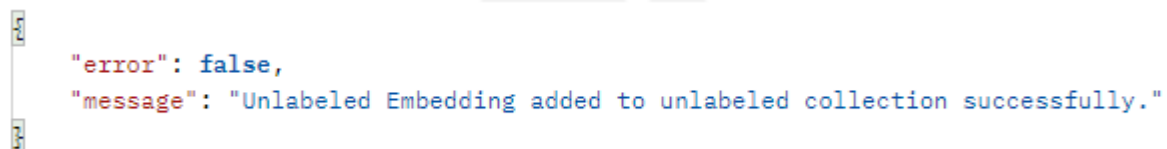
Figure 12

If a room exists the next thing is to check if a frame or face base64 string was provided, and if so add it to the frames or faces collection. This is done by using the defined cluster in the setup to reference the corresponding collection and later utilizing the insert_one function to pass the frame or face to the collection for storing.

Both “/labelled_embeddings/add” and “/unlabelled_embeddings/add” are identical to this point, but here unlabelled_embeddings differentiate itself by utilizing a clustering algorithm in a final attempt to label the provided embedding passed to the endpoint. This is first done by first, adding all labelled embeddings to a list then appending the list with the unlabelled embedding. The labelled embeddings are stored within the labelled_embeddings collection within the MongoDB so to access these we use the .find() function after referencing the collection. Next, we define a cluster algorithm by creating a

DBSCAN object, a clustering algorithm part of the sklearn library. The DBSCAN object takes three arguments in its constructor being the metric used to calculate the Euclidean distance between embeddings, the eps value being the distance embeddings two embeddings need to be within to be considered a cluster and the min_samples, a value defining how many embeddings need to be within the eps value to one another to be considered a cluster. These arbitrary values were decided upon after performing testing discussed in the 5.2.

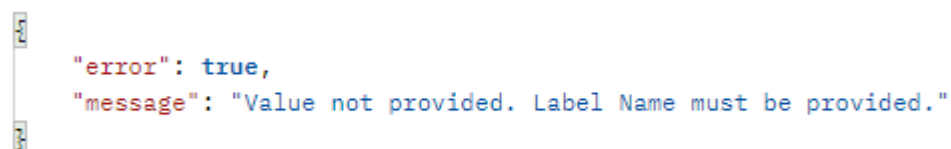
The next stage in the endpoint passes the list of embeddings to the cluster algorithm using the fit_predict method, and in return the DBSCAN algorithm returns a list of labels corresponding to the embeddings passed. The labels returned are numbers where if embeddings have the same number, they are part of the same cluster. As the unlabelled embeddings is the last element in the embeddings list, its prediction is the last element in the list of labels. If the label predicted for the unlabelled embedding does not match any other embeddings, then it means it is a new individual and is simply added to the unlabelled embeddings collection, using the insert_one function against the collection. If the label predicted is also predicted for other embeddings, then we find another embedding predicted with the same label and search the labelled embedding collection for the label belonging to it. From here “/labelled_embeddings/add” and “/unlabelled_embeddings/add” are similar again. Both call the check_alerts function used to send an alert using the discord bot if the individual detected has an alert against their label when detected in that particular room. Next, the embedding is passed to the labelled embedding collection alongside the label id, the room id and the timestamp for when the detection occurred. Finally, a JSON response is returned using jsonify to tell the camera service that the operation was successful, with a 200-status code indicating all is ok.



```
{  
  "error": false,  
  "message": "Unlabeled Embedding added to unlabeled collection successfully."  
}
```

Figure 13

“labels/track” is used when retrieving the movement histories of an individual. The function queries the labelled embeddings collection, retrieving the room id and datetimes for all entries where the label id matches the label id passed to the endpoint. Within this endpoint there is validation ensuring a label has been provided and if not, then an error message is returned inside a jsonify response.



```
{  
  "error": true,  
  "message": "Value not provided. Label Name must be provided."  
}
```

Figure 14

Using the room ids, the function queries the room collection to find the names of the rooms before creating a dictionary with each entry containing the room name and the timestamp the detection occurred.

```
{
  "room_name": "Room_1",
  "timestamp": "Mon, 08 May 2023 07:02:42 GMT"
},
{
  "room_name": "Room_1",
  "timestamp": "Mon, 08 May 2023 07:02:39 GMT"
},
{
  "room_name": "Room_1",
  "timestamp": "Mon, 08 May 2023 07:02:34 GMT"
},
{
  "room_name": "Room_1",
  "timestamp": "Mon, 08 May 2023 07:02:33 GMT"
},
{
  ...
}
```

Figure 15

Within the jsonify JSON response, the dictionary is returned to the frontend where it is then utilized for the feature. Above in figure 15 is an example response.

“/labels/update” is used when a user utilizes the label people feature on the frontend to label a previously unknown individual. This function starts by validating a label name was provided by the user and that the images ObjectId has been provided. If this is not the case then a jsonify response is returned with a corresponding error message.

```
{
  "error": true,
  "message": "Value not provided. Label name must be provided."
}
```

Figure 16

Assuming all parameters are provided, the function checks if the label already exists or if it needs to make a new label in the labels collection by utilising the find_one function as part of the MongoDB library, querying the labels collection. After this, the embedding corresponding to the image displayed to the user is located using the frame id or face id attached to the embedding within the unlabelled_embeddings collection. Once located, all unlabelled embeddings are passed to the DBSCAN clustering algorithm to identity embeddings that belong in the same cluster as the now labelled embedding. Next, all embeddings within the same cluster as the newly labelled embedding are deleted from the unlabelled_embeddings collection and moved to the labelled_embeddings collection

and provided with their new label. Upon completion, a jsonify response is returned with a success message and a status code of 200 indicating all is ok.

```
{  
  "error": false,  
  "message": "Labeled successfully."  
}
```

Figure 17

“/” is used when a user connects to the frontend service and serves the purpose of providing all information needed by the frontend to correctly display the webpage, forwarding all labels, room names, active alerts, camera connections and the current locations of individuals both known and unknown. The function begins by using the clustering algorithm on all unlabelled_embeddings so that each cluster can be considered its own individual. The labels list provided by the clustering algorithm is stored and later used towards the end of the function when asserting what the most recent room each unknown individual was detected within and the timestamp of that detection.

This function revolves around appending slowly to a dictionary which is later returned within the jsonify response. All labels stored within the labels collection are added to the dictionary so they can be used for creating the detection alert table. After this, the function moves on to gathering information on each room by looping through the rooms collection. This includes all camera connections and active alerts both found through querying their corresponding collections using the room id field. After this, the most recent detection for each label is then attributed to the room it was detected in, determining the last location of all individuals. All this information is stored within a temporary dictionary with the key being the room name and the value being the information. At the end of the room loop, this dictionary is appended to the returning dictionary and later forwarded to the frontend using jsonify. Figure 18 which can be seen on the next page showcases the layout of the JSON response.

```

"rooms": {
  "Room_1": {
    "cameras": [...],
    "labels": {
      "James": [...],
      "Georgie": [...],
      "Sheldon": [...],
      "unknown person 1": [...],
      "unknown person 3": [
        "2023-05-08 06:32:04.434000",
        "2023-05-08 06:32:02.365000",
        "2023-05-08 06:32:01.221000",
        "2023-05-08 06:31:35.805000",
        "2023-05-08 06:31:34.195000"
      ],
      "unknown person 2": [...],
      "unknown person 4": [...]
    ],
    "in_room": {
      "James": "2023-05-08 07:02:42.652000",
      "Georgie": "2023-05-08 06:31:18.271000",
      "Sheldon": "2023-05-08 07:03:30.316000",
      "unknown person 1": "2023-05-08 06:43:31.533000",
      "unknown person 3": "2023-05-08 06:32:04.434000",
      "unknown person 4": "2023-05-08 06:28:39.936000"
    },
    "alerts": {
      "unknown_individuals": false,
      "James": false,
      "Georgie": false,
      "Jack": false,
      "Sheldon": false,
      "bad image": false,
      "George": false,
      "Daniel": false
    }
  },
  "Room_2": {...}
},
"label_image": {},
"labels": [...],
}

```

Figure 18

4.3.3 Implementation of the Frontend Service

The implementation of the frontend service begins by creating an express app responsible for dealing with various aspects of web development such as defining the routing users will utilize to handling incoming requests and returning responses. The express app starts by defining all libraries it requires which were discussed in detail within 4.2.3, and then moves on to configuring the middleware of the application. The most important of the middleware configurations is setting up Nunjucks templating engine to render views within the express application, providing lots of additional functionalities when serving dynamic templates. The most important function within the express application is the main “/”

route. This route is accessed when a user attempts to connect to the website and begins by forwarding a request to the backend services “/” route, which returns a JSON response, containing all information needed to display the frontend. The express application parses this JSON and passes it to the nunjucks templating engine while calling for index.html to be rendered.

To keep index.html simplified and to give some compartmentalization between the different features, each feature is provided their own template file which is then called within index.html using the “include” keyword from the Nunjucks. This is also useful because for features that appear multiple times, there is no code reuse required and the template file can simply be called again with new data. A simplified example of how this is used can be seen in figure 19.

```
{% for item in items %}  
{% include "item.html" %}  
{% endfor %}
```

Figure 19

The index.html file starts up by including a dependencies template which imports all JavaScript and stylesheets needed without cluttering creating unnecessary clutter in the file. The key dependencies are Materialize, chart, and datatables. Materialize is a framework providing pre-designed CSS and JavaScript components and with index.html, the frontend service utilizes the container class to provide a fixed-width area for content within the centre of the screen, giving a bit of padding from the sides of the screen. The service also uses the col and row classes which are used in unison to create a responsive grid layout allowing for other components to be aligned more easily. Each row has a grid length of 12 so for the buttons at the top of the page, a col length of 3 is used in each to provide even spacing for the four of them. In addition to this, Materializes was utilized for its card-panel component to make the buttons, to be the underlying space room details appear in, and to contain the movement histories feature at the bottom of the page. The buttons can be seen below in figure 20.



Figure 20

I also used Materializes form components, using their text input boxes when creating the add label, add room, and label people feature, while also using it with the select input component in the add camera feature. All of the mentioned features were created within the modals template, the modal being what appears when their corresponding button is clicked. A JavaScript function was used to

open the correct model form using the id of the button clicked. Each of these forms sent a POST request back to the express application which forwarded the form to the backend service, the express application waited for a response before forwarding the success or error message into a flash message then redirecting to the “/” route. The flash message is then displayed as the index.html page is loaded again. The modals can be seen in figures 5, 6, 7 and 8 respectively.

The rooms section as seen in figure 9 is dynamically generated using the JSON passed from the backend service. Nunjucks uses a for loop to generate a card-panel for each room within the JSON and then populates the rooms card-panel with all information stored against its name. This includes creating the donut chart. The donut chart is created using chart.js, a library specialising in creating graph components. Details of the number of detections each label has within a room are used to create the graph and then it is displayed on the page. The donut chart code is kept in its own template named donut_graph. The JSON file also contains information on individuals last detection, which is used to populate the table within the rooms section. The table used within the room section is from datatables, a library allowing for the enchantment of HTML tables. Nunjucks uses a for loop to populate the table and datatables library takes care of providing the pagination. This is handled in a template named currently_in_room.

The JSON file also keeps track of all labels stored within the database and all active alerts. This information is used to create the alerts section within each room section, using Materializes switch components to enable and disable them. Each switch triggers an AJAX function when changed, when forwards a message to the express application. The express application then passes the request on to the backend service where a response is awaited, once a response is returned it is displayed on the frontend service in the form of a toast, also a materialize component. This is all handled in a template named alert_room.

After the room sections have been created, Nunjucks creates the movement histories section as seen in figure 10. This section uses a select input, which is populated with all labels using Nunjucks. The select begins empty and upon change, the chosen label is then passed to the backend service using an AJAX function where the movement histories of that label are returned in response. After they’ve been received, they’re populated into the table using a Nunjucks for loop cycling through the JSON response. To create the pagination, datatables simply updates the frontend UI. This feature is kept within the label_track template.

5.0 Testing

5.1 Testing of the Camera Service

The deepface library that provides access to the facial detection and facial recognition algorithms also gives lots of choice in which model, backend detector or target sizes should be used when utilizing their functions. With that being said it was important to tune the service so performing tests to find out the most appropriate values for the service was a necessity. In addition to this, the facial recognition algorithm returns a confidence value and it was important to decide upon the threshold value for which a confidence value needs to be beneath to be considered the detected identity. (Confidence score closer to zero is good in deepface library).

Furthermore, testing the detection capabilities of the models was also an important factor seeing if detections from some models were possible even under more difficult conditions such as an individual wearing glasses, or possibly wearing a hat.

To test for all of this, several tests were created to try out all models, backend detectors, target sizes and confidence value combinations when processing a selection of videos chosen some for their similarity to the expected use of the system, and some to test the limitations on what the models are capable of detecting.

Due to the many combinations of values possible, the decision was made that if a model was not capable of processing the video within the timeframe of one minute, then the time limit would be deemed exceeded and the algorithm testing the combinations would continue. This decision seems reasonable given the need to process images in real-time so if the model can not process a video within the given time-limit it is unlikely to be suitable for the system.

5.1.1 Testing of the Facial Detection Algorithm

The first test made was to find the target size and backend detector for the facial detection algorithms. The videos were processed one after the other by the detection algorithm, using all combinations of target size and backend detector where the time taken and number of faces detected was stored within a text file. The results showcases that the models retinaface and mtcnn never finished within the time limit and thus were removed from consideration for being used within any of the algorithms. Due to technical issues, it was not possible to test the dlib backend detector on my system and due to time constraints, not possible to run on another computer therefor we can not rule dlib out as a possible

choice within a future implementation of the camera service. Two backend detectors that performed well were OpenCV and ssd, but it should be noted that in larger videos OpenCV did not always finish within the timeframe where as ssd did. To compare OpenCV and ssd in videos were both finished, OpenCV would detect almost three times as many faces but would take almost four times as long. It was discovered that the target size did not affect the number of detections but did affect how long it would take the algorithm to complete but this effect was minimal. From this information, the decision was made to use ssd as the backend detector with facenet for the target size as the algorithm finished quickest with these values while also detecting faces.

What was also discovered in testing is that even for some videos that did finish processing, no faces were detected. These videos highlighted the limitations of the algorithm, struggling to detect individuals when they're wearing a hat and a mask, when the camera uses a black and white filter or when the camera is using an ultra-low resolution.

5.1.2 Testing of the Facial Recognition Algorithm

A similar test approach to the one used in 5.1.1 was used to test the facial recognition values such as backend detector, model, target size and confidence values. The videos chosen for this algorithm featured myself as the algorithm could be trained to identity my face and the values used for the facial detection algorithm were the ones discovered through testing in 5.1.1. Through testing, several things were discovered. The first thing discovered was that SFace would not be a suitable model due to its high rate of incorrect identification, particularly as the confidence value got closer to zero, while OpenFace and DeepFace models would also not be used due to their failure to detect any individuals from the cropped face photographs provided from the facial detection algorithm. The results of these models can be seen in figure 21 when used with the OpenCV backend detector.

```
OpenFace, candidate score: 1.1, time taken: 15.072077989578247, correct detections: 1, incorrect detections: 0, no detections: 41
OpenFace, candidate score: 1.05, time taken: 13.193252086639404, correct detections: 1, incorrect detections: 0, no detections: 41
OpenFace, candidate score: 1.0, time taken: 13.173793315887451, correct detections: 1, incorrect detections: 0, no detections: 41
OpenFace, candidate score: 0.95, time taken: 14.20468783378601, correct detections: 1, incorrect detections: 0, no detections: 41
OpenFace, candidate score: 0.9, time taken: 13.051167726516724, correct detections: 1, incorrect detections: 0, no detections: 41
DeepFace, candidate score: 1.1, time limit exceeded, correct detections: 2, incorrect detections: 0, no detections: 16
DeepFace, candidate score: 1.05, time taken: 19.400050163269043, correct detections: 2, incorrect detections: 0, no detections: 16
DeepFace, candidate score: 1.0, time taken: 20.653000354766846, correct detections: 2, incorrect detections: 0, no detections: 16
DeepFace, candidate score: 0.95, time taken: 19.782551288604736, correct detections: 2, incorrect detections: 0, no detections: 16
DeepFace, candidate score: 0.9, time taken: 18.834648370742798, correct detections: 2, incorrect detections: 0, no detections: 16
```

Figure 21

Retinaface once again proved itself not to be suitable for use as a backend detector, as the time limit was exceeded once again when testing it. MTCNN managed to finish within the time constraints but was four times as slow as OpenCV and proved to provide similar results. When using OpenCV as the backend detector, all models finished in time. The best model when using OpenCV was determined to

be FaceNet512. FaceNet512 only had 1 incorrect detection out of 47 and finished within a reasonable amount of time, compared to other models such as ArcFace and Facenet where incorrect detections were higher. The confidence value also decided was 0.9, the lowest value and most confident value considered. The results of FaceNet512 when used with the OpenCV backend can be seen in figure 22.

```
Facenet512, candidate score: 1.1, time taken: 18.488009691238403, correct detections: 47, incorrect detections: 0, no detections: 0
Facenet512, candidate score: 1.05, time taken: 13.017855644226074, correct detections: 47, incorrect detections: 0, no detections: 0
Facenet512, candidate score: 1.0, time taken: 13.752110481262207, correct detections: 47, incorrect detections: 0, no detections: 0
Facenet512, candidate score: 0.95, time taken: 13.478657245635986, correct detections: 47, incorrect detections: 0, no detections: 0
Facenet512, candidate score: 0.9, time taken: 15.554991245269775, correct detections: 46, incorrect detections: 0, no detections: 1
```

Figure 22

5.2 Testing of the Backend Service

To test the backend service, unit tests were created alongside a test script for finding the best eps value to be used within the clustering algorithm. The unit tests will assert that the endpoints are working as intended, for expected cases and for cases requiring error handling. The test script for the clustering algorithm was created to make sure that the algorithm is tuned correctly and will provide the best results for the system. This section has been separated to discuss both independently.

5.2.1 Testing of the Clustering Algorithm

To test the clustering algorithm, a script similar to the one used in 5.1.1 was developed, which would detect faces using the backend detector and target size already decided upon, then process the detected face into embeddings just like they would be when used within the backend service. From here the algorithm was used with varying eps values, while the number of clusters detected was monitored. The clustering algorithm was processing the same videos used as samples in the other testing experiments, with the correct number of expected clusters being provided by the developer for each video prior to testing. The test outlined interesting results with a consistent pattern being that when the eps value was too high the number of clusters remained at 1 even if multiple people were within the video. This is because with an eps value too high, the clustering algorithm clusters too generally. Similarly, when the eps value was too low then the clustering algorithm clustered too strictly meaning too many clusters were created.

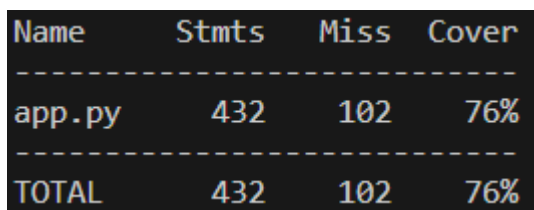
What was discovered from testing, is an eps value of 0.3 yielded the best results, correctly identifying the right number of unique individuals within the videos and thus was used within the backend service.

5.2.2 Unit Testing of the Backend Service

To thoroughly test the backend service, unit tests were written to provide adequate code coverage of the different flask endpoints. These unit tests were developed using the unittest library and cover the expected paths and the unexpected paths, checking that sufficient error handling is in place while also providing adequate error messages. Using the unit test library, a test client could be created which would be capable of acting as the flask application within the test environment. With this approach, unit tests could send requests to the test client and receive responses as if it had messaged the actual application, without having to setup the application within a flask container. Each test was given a descriptive name first prefixed with test, followed by the action, then the collection name and lastly the scenario being accounted for. For example, test_add_camera_no_room_id_provided tests adding a camera to the camera collection without a room id being passed to the endpoint.

To assert the status code and JSON response for each unit tests, `assertEquals` method was used, comparing the expected value to the actual value. If the two values do not equal each other then that test is failed. In total 40 unit tests were developed with each of these unit tests passing as can be seen in figure 24.

After developing the unit tests, the code coverage was calculated using the coverage library in python, showcasing a coverage of 76% as seen in figure 23. Some of the lines not covered include setup code at the start and end of the flask application and thus we conclude the system is likely to boast a coverage closer to 80%. Achieving a test coverage of 80% is considered at the top end of optimal so it is sufficient to say that there has been adequate test coverage for the backend service endpoints. [16]



Name	Stmts	Miss	Cover
app.py	432	102	76%
TOTAL	432	102	76%

Figure 23

These unit tests were used to fix bugs detected when updating different endpoints and should continue to be added if a developer continues to work on the application.

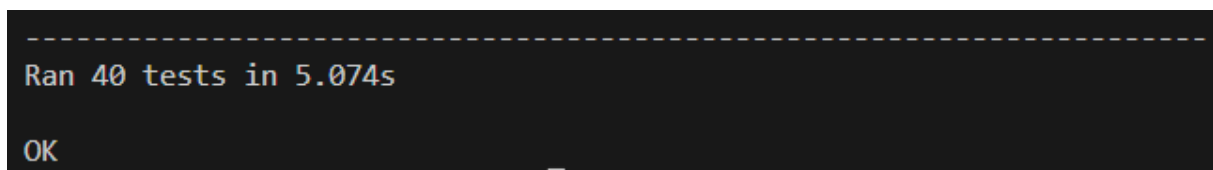


Figure 24

6.0 System Evaluation and Experimental Results

Within this section, the overall system will be evaluated based on its ability to meet the requirements previously outlined during section 2. Experimental results discovered during the development of the system will also be discussed, followed by steps needed to be taken in the future to ensure that the system could be considered a professional product.

6.1 Ability to Meet Requirements

Section 2.2 discussed the user characteristics of someone using the system and what that type of individual is likely to want from it and within section 2.3, functional and non-function requirements were created which have been used to evaluate the overall success of the system. To evaluate the success of the system requirements created during 2.2, user feedback was obtained through allowing people to interact with the system and to use the different features through the frontend application. After using the system, individuals were shown the requirements outlined and asked which of the requirements they believe the system achieved. Users were able to use the system on different browsers without any crashes and were capable of utilizing all of the different features presented through the frontend. Overall, the feedback from this was resoundingly positive, with users all agreeing that the website was intuitive and when issues did arise through misusing the features, the error messages were informative on how to rectify the issue.

Users also agreed that separating the features was an added benefit that made the whole experience seamless and simple to use, and thus can be assumed that the end users will also be able to use the system as effectively and efficiently.

With regards to the function and non-functional requirements outlined within section 2.3, the system has been able to achieve these requirements. For the camera service, the system can process the video feeds of multiple cameras, each given its own thread where processing of facial detection and recognition occurs. The system has testing in place to ensure that detections are made accurately and overall, the system has minimum downtime.

The backend service developed has been capable of interacting with the cloud database stored in mongo both forwarding information and retrieving it for different features. The backend service receives communications from the camera service and provides sufficient error handling proven through the unit tests developed as part of section 5.2.2. The service has high scalability with its dockerization, allowing cloud providers to spin up additional containers should the service be

overworked. With all of this in mind, it is safe to evaluate the backend service meeting its expected requirements.

The frontend service was expected to provide users access to all features through the frontend and be capable of accessing the MongoDB database through the backend service. It was also expected to be seamless to use and be responsive to users using the website. The frontend service has been able to meet these requirements, proven both in the user feedback but also in its ability to communicate information to endpoints part of the backend service, providing a connect to the database through the frontend.

6.2 Experimental Results

During the development, experimental results were discovered particularly within the camera service. One interesting experiment was testing if it was possible to monitor all cameras without needing to use threading and concurrency to run the different facial detection and recognition models. What was discovered was that it is possible but requires looping through the different VideoCaptures containing the incoming video feeds within a never-ending loop and then evaluating the current frame. With this approach, it is plausible to use the camera service on computers with a limited number of threads or limited computing power but would come with the risk of detections being missed if too many VideoCaptures need processing as the detectable frame would be missed. Nevertheless, this experiment proves the possibility of a different version of the system being releasable where the number of cameras is small mitigating that potential risk.

In addition to this, another interesting experiment was removing facial detection and simply relying on facial recognition on the entire frame for every frame. The reason this would work is because the facial recognition function “find” uses the facial detection function “extract_faces” to locate the co-ordinates of faces before processing those detected faces. The difference between using this approach and the final approach used within the system is that it is overall more expensive computationally to simply use “find”. This is because you add additional redundancy in terms of more face detection computations therefor increasing the overall computational cost. It was for this reason the decision was made to break up the detection and recognition phases into their own section and only call the recognition if a detection was made.

The next interesting experiment was relying on the clustering algorithm DBSCAN for facial recognition and remove the algorithm provided through deepface. While this approach did seem promising, relying solely on one algorithm seemed riskier and thus was not chosen for implementation

in the live system but did prove to be very fast. Despite this, the DBSCAN algorithm will becoming slower should the number of embeddings increase which would happen over long periods of time.

All of these experimental approaches can be found within the deepface branch within the experiments folder.

6.3 Weaknesses

While overall the system was successful it highlighted several weaknesses. The first weakness being difficulty in displaying live camera feeds from generic IP cameras within a live page and while the frontend service provided an adequate fix to this, it did redirect users to another tab which was something not done by competitors and thus is seen as a weakness of the system.

Additionally, the system utilizing threading to process multiple IP camera feeds simultaneously makes it difficult to run on lower-end systems. With this in mind, the current implementation of the camera service only lends itself to systems with more processing power and would be difficult to utilize in its full capacity by all users. While experimental developments of the camera service would be more suitable for these individuals and could be possible in future adaptations of the service, the current camera service is not inclusive of all machines and thus could consider that as a weakness.

6.4 Steps Taken in the Future

There are improvements to be made to the system that would need to be taken in the future for the system to be considered a final product. One major improvement would be to dockerize the camera service like the backend and the frontend service. With this approach users would not need to run the camera service locally and could allow computations to be done on the cloud where much more powerful CPUs and GPUs could potentially be used to run the facial detection and recognition algorithms.

Another improvement would be to develop a mobile application or at least update the website to be mobile compatible for users who do not have access to a computer to use the current implementation of the frontend service. This would make the system more accessible for users and potentially increase the marketability of the product. Similarly, making the frontend service accessible to those with disabilities by providing accessibility features would be an important update needed should the system wish to be more professional.

6.5 Conclusion

In conclusion, the system developed was a resounding success. It demonstrated that it is possible to use IoT devices to monitor an environment and do so in a way that can provide features even the largest companies do not offer. The system proved it is possible to run facial detection and facial recognition algorithms using the feeds of generic IP cameras and that it is not necessary to need specialist cameras to do so. The system managed to take full advantage of cloud streaming and proved how easily IoT devices can be used in unison with it to create new products which challenge existing thinking.

7.0 Appendices

[1] Duggal, N. (2021). *What Are IoT Devices : Definition, Types, and 5 Most Popular Ones for 2021*. [online] Simplilearn.com. Available at: <https://www.simplilearn.com/iot-devices-article>.

[2] www.ETTelecom.com (n.d.). *At 12 billion, IoT connections to surpass non-IoT devices in 2020 - ET Telecom*. [online] ETTelecom.com. Available at: <https://telecom.economictimes.indiatimes.com/news/at-12-billion-iot-connections-to-surpass-non-iot-devices-in-2020/79318722?redirect=1> [Accessed 21 May 2023].

[3] Limited, I. (n.d.). *Three ways cloud computing and IoT deliver business benefits / Infosys BPM*. [online] www.infosysbpm.com. Available at: <https://www.infosysbpm.com/blogs/business-transformation/three-ways-cloud-computing-and-iot-deliver-business-benefits.html> [Accessed 21 May 2023].

[4] updated, M.P. last (2021). *Ring Video Doorbell 3 Plus review*. [online] Tom's Guide. Available at: <https://www.tomsguide.com/reviews/ring-video-doorbell-3-plus> [Accessed 21 May 2023].

[5] Ring (2020). *Ring's Stance on Facial Recognition Technology*. [online] The Ring Blog. Available at: <https://blog.ring.com/about-ring/rings-stance-on-facial-recognition-technology/> [Accessed 21 May 2023].

[6] Oct 2020, W.G. 14 and P.m, 3:12 (2020). *The Best Indoor Home Security Cameras for 2020*. [online] PCMag UK. Available at: <https://uk.pcmag.com/home-security-cameras/39333/the-best-indoor-home-security-cameras>.

[7] ARLO. (2021). *Arlo Secure Subscription - Maximum Subscription Security | Arlo UK*. [online] Available at: https://www.arlo.com/en_gb/serviceplans [Accessed 21 May 2023].

[8] PCMag UK. (2022). *SimpliSafe Home Security System*. [online] Available at: <https://uk.pcmag.com/home-security/79585/simplisafe-home-security-system>.

[9] Wyze (2022). Does Cam Protect include Facial Recognition? [online] available at: <https://support.wyze.com/hc/en-us/articles/4409136316443-Does-Cam-Protect-include-Facial-Recognition->

[10] IoT.Business.News (2021). *Increasing Use of Cloud Computing Accelerating the IoT Connected Machines Market*. [online] IoT Business News. Available at: <https://iotbusinessnews.com/2021/04/15/39909-increasing-use-of-cloud-computing-accelerating-the-iot-connected-machines-market/#:~:text=The%20cloud%20also%20offers%20solutions> [Accessed 21 May 2023].

[11] Salesforce (2015). *12 Benefits of Cloud Computing and Its Advantages - Salesforce.com*. [online] Salesforce.com. Available at: <https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/>.

[12] Facebook.com/TechGadgetsCanada (2021). *How to Access Ring Video Doorbell Cameras with a Web Browser*. [online] Digital Trends. Available at: <https://www.digitaltrends.com/home/how-to-access-ring-video-doorbells-cameras-through-web-browser/> [Accessed 21 May 2023].

[13] artincontext (2022). *Amber Color - What Does the Color Amber Look Like?* [online] artincontext.org. Available at: <https://artincontext.org/amber-color/> [Accessed 21 May 2023].

[14] marian (2015). *Color Palette #1439*. [online] Color Palette Ideas. Available at: <https://colorpalettes.net/color-palette-1439/> [Accessed 21 May 2023].

[15] Babich, N. (2016). *Design Principles: Introduction To Hierarchy*. [online] Medium. Available at: <https://uxplanet.org/design-principles-introduction-to-hierarchy-ec56fcbd0306>.

[16] learn.microsoft.com. (n.d.). *Test Coverage Definition - Unit Testing - Microsoft Q&A*. [online] Available at: <https://learn.microsoft.com/en-us/answers/questions/778016/test-coverage-definition-unit-testing>.