RAID: Real-time Animal Identification and Deterrence

Jake Black

N00193326

Supervisor: Mohammed Cherbatji

Second Reader: Cyril Connolly

Year 4 2022-23

DL836 BSc (Hons) in Creative Computing

Abstract

The aim of this project was to create a novel system capable of leveraging computer vision technology not only to identify animals by species, but of distinguishing one individual from another of the same species based on pet images uploaded by the user, and to be capable of doing this in real-time via a live camera feed. With this achieved, the project was also intended to act as a deterrent for wild animals or stray pets by running a two-stage object detection system in which the first stage detects an animal's species, and the second determines whether the animal is the user's pet, or not.

The rationale for creating this system was the realisation that a system using computer vision technology in this way does not exist in the form of a user-friendly and full-featured application which brings the power of computer vision to everyone. Rather, only personal projects recognising a single animal, or standalone computer vision research projects seem to exist.

The steps involved in the development of the system were the creation of three separate but interdependent parts, a ReactJS client, a NodeJS Express and SocketIO web server, and a Raspberry Pi single-board computer connected to a web camera and a Piezo buzzer.

This report documents the research, program design, and user interface design carried out in preparation for creating this system, provides a detailed iterative log of the implementation process, and concludes by discussing the project management methodologies and technologies used over the course of the application's development, the potential business opportunities, and discusses my own personal views on the project. The purpose of the abstract is to give the reader of the report a concise overview of the project.

Both the initial research survey and user testing results suggest that a niche exists for this system. With further work in creating a smooth and polished user-interface and developing additional features, the system could prove popular.

Acknowledgements

I would firstly like to thank my supervisor, Mohammed Cherbatji for his guidance and encouragement throughout this project. Mohammed's expertise has been invaluable to me in solving the problems which have arisen during this project's development.

A special thank you is due to the Roboflow team, particularly to Mohamed Traore, Joseph Nelson, Jacob Solawetz, and Sachin Agarwal. Roboflow provided me with several training credits free of charge, which facilitated the implementation of the unique pet identification aspect of the project, and without which the core concept of this application would not have been possible.

The team engaged with me in solving a bug in the Roboflow API, and quickly merged a fix when the issue was discovered. I also appreciate Roboflow providing me with the opportunity to contribute to their codebase and merging my pull request.

I would like to thank my friends and classmates, Séan Óg Durack Monks and Paul Doyle. Their camaraderie has been immensely helpful throughout my time at IADT, and the experience would not have been half as enjoyable without them.

Finally, I would like to thank my family for their support, not only throughout the development of this thesis, but throughout my entire four years at IADT. In particular I would like to thank my fiancée Lea, my mother Sandra, and most importantly my father, Frank Black. Without his support my time at IADT would not have been possible.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

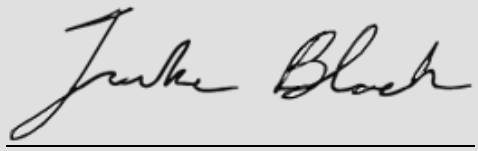
Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student: Jake Black

Signed:



Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

1 Introduction

The overall aim of this project was to create an animal identification and deterrence system capable not only of detecting and recognising various animal species, but of distinguishing between individual animals of the same species. Examples of leveraging computer vision technology to identify animals exist in the public domain and in literature, as discussed in the research chapter of this report. However, a complete and user-friendly implementation of such a system seems to be a novel concept. Existing examples of computer vision models capable of recognising individual animals are generally standalone image classification or object detection models, and are not wrapped in a user interface, nor are they designed to be repurposed based on an individual user's use-case. Advancements in computer vision technology and the increasing number of services offering cloud infrastructure on which to run computer vision models has allowed for this project to be realized, by combining a local custom-trained YOLOv4-Tiny model designed for identifying various animal species with cloud models trained on a per-user basis using Roboflow, which performs a secondary object detection step to uniquely identify an animal.

The system was designed as a web application consisting of three core components, a ReactJS client, a NodeJS, Express, and SocketIO web server, and a Raspberry Pi single-board computer connected to a web camera and a Piezo buzzer. The client combines a custom fork of React-BBox-Annotator and React-Dropzone-Uploader to allow users to upload and label images of their pet in order to train their custom object detection model. The server then leverages a comprehensive set of utility functions for interacting with the Roboflow API, which have been compiled by translating methods from the existing Roboflow Python SDK, merging them with repurposed methods from the NodeJS command line interface in order to convert raw coordinates to annotation files, combine them with the user's images, create a training, testing, and validation split, and finally upload to Roboflow to begin custom training. All the while, the server is receiving images, class predictions, and confidence values from the Raspberry Pi, which is constantly running inference on a web camera stream. In this way, the user can monitor a live broadcast received over the Internet from the Raspberry Pi, and the server and the Raspberry Pi can work together to first identify animals, and then use the custom model to determine whether an animal is the user's pet, or not. If not, the Raspberry Pi will trigger its Piezo buzzer to deter the animal and capture an automatic screenshot of the detection event.

The entire research and development process carried out over the course of this project will be detailed in this report. Beginning with the research chapter, the report first discusses the steps involved in facial recognition and some of the most common algorithms used in this area. We then examine some of the studies carried out into applying computer vision technology to animal identification, which explores the varying techniques used by researchers to identify a wide variety of animals, from pets and domestic farm animals to endangered species, and discusses how findings from these studies might be used to guide the creation of this project. The requirements chapter will then detail research into similar applications in the public domain, evaluating their advantages and shortcomings and how the technologies used by these projects fit into the scope of our own.

Within the requirements section, we also explore the results of the survey carried out into what features users would expect from an animal recognition and deterrence system, using this information to model the application's functional and non-functional requirements.

The program design and user interface design are then discussed in depth, and an explanation is provided covering all the technologies involved in making this project a reality, from the React client

through to the server, Raspberry Pi, and third-party services, explanations on technologies and how they fit into this project are given, including React itself, Vite, TailwindCSS, Express, NodeJS, and computer vision related technologies such as convolutional neural networks and the YOLOv4-Tiny algorithm.

The project was developed using the SCRUM methodology, and by far the largest section of this report is the implementation section, which is an iterative log of the seven sprints, detailing how every feature was implemented, including the various version iterations created as the core technologies changed. The implementation chapter in itself provides a detailed look into every technology used over the course of the project's development, which have been many and varied.

Following the implementation chapter, the testing chapter will discuss functional testing of every aspect of the application's features to ensure they align correctly with the functional requirements set out in the requirements chapter and will discuss the user testing process and its results. The final chapters will describe the project management strategy and tools used over the course of the project's development, provide a personal reflection and my own views on the project, and discuss potential business opportunities and the skills gained over the course of the project's development. Finally, the concluding chapter will reiterate the technologies used, the research carried out, the program and user interface design, the entire implementation process, the testing results, and overall result of the project.

2 Research

2.1 Introduction

According to Kumar & Singh (2016), over 7 million dogs enter animal shelters in the United States each year, but only 1 in 10 of these animals are either adopted or returned to their original owners. At the time of writing, the author states that no animal biometric recognition system exists in literature or in the public domain suitable for solving problems such as missing or stolen animals, false insurance claims, or to act as an alternative to the currently available methods of animal identification, such as RFID tags, microchips embedded beneath the skin, or branding, all of which are invasive and distressing for the animal. With over 4 billion pet animals living with people throughout the world, the creation of a novel, non-invasive solution for biometric identification of animals would undoubtedly be of practical use.

Aside from domestic pets, a similar system may be even more applicable to the farming industry. Hansen, et al. states that the world's food demands are increasing alongside the rising population, but the need for environmentally friendly food production practices means the desire for sustainable farming intensification is becoming more commonplace. The ability to monitor the needs and outputs of each animal is becoming increasingly desirable among farmers to aid in this process. At present, the most common technology underpinning farm animal identification is the RFID chip, which is invasively fitted to the animal's ear, causing them distress. Furthermore, the RFID chip has a limited maximum detection range of only 120cm and was found to have an accuracy of only 88.6%, even when a pig was fitted with *two* RFID chips. In contrast, the proposed biometric recognition system for dogs cited by Kumar & Kumar Singh (2016) had an accurate recognition rate of 96.87%.

In terms of biodiversity conservation and studies in ecology, animal identification is also of paramount importance. Similar issues exist concerning wild animals as in the previously discussed areas of household pets and domestic farm animals. Researchers in this area need to keep track of individual members of threatened species, but existing methods can be either invasive, or ineffective. Researchers will either capture and tag animals, or rely on their expertise to observe animals from afar and distinguish between individuals based on unique features. The former is incasive and causes disturbance to the animal, and the latter inconsistent, due to variations in the identification techniques used by different researchers, especially when many researchers are involved in studies taking place over a large geographic area. The machine learning model trained in the identification of giant pandas by (Hou, et al., 2020) achieved an accuracy of 95% in identifying pandas based solely on facial images.

Owing to the advent of faster and more powerful computer hardware, improvements in algorithms, and the availability of abundant amounts of data, the field of computer vision within machine learning has seen huge improvement over earlier techniques, allowing for the development of facial recognition technology achieving near human-level accuracy.

An abundance of work has been published on the subject of human facial recognition, but a comparably under-researched area is that of animal facial recognition. In order to assess the feasibility of applying such techniques to an animal recognition system, it is necessary to examine these developments further. This literature review seeks to assess the gains made and the current state of the relevant technologies with a view to developing such a system. We'll discuss the problems such a system could help to solve, the successes others have had and the techniques they've applied.

2.2 Facial Recognition Steps and Common Algorithms

To provide some background on the technologies underpinning any facial recognition system, we refer to Facial Recognition Systems: A Survey, by Kortli, Jridi, Al Falou, & Atri, 2020. The authors explain that facial recognition is composed of three stages: face detection, feature extraction, and face recognition. The first step is used to detect the presence and location of a human face in an image or video. The feature extraction step is used to extract the relevant feature vectors from the human face located in the previous step. The final step, facial recognition, considers the extracted features and compares them with existing facial images stored in a database.

Regarding the algorithms used, the face detection step may use the Viola-Jones detector, histogram of oriented gradient (HOG), or principal component analysis (PCA). The study on using the Raspberry Pi single-board computer as a compact facial recognition system mentions the Viola Jones algorithm is used for face detection, while Locality Constrained Sparse Representation for Cat Recognition by Chen, Hidayati, Cheng, Hu & Hua, 2016 makes mention of principal component analysis, but ultimately used representation-based classification (SRC) for face detection.

Among the algorithms used in feature extraction, those discussed later in this literature review include Eigenface, independent component analysis (ICA), linear discriminant analysis (LCA), local binary pattern (LBP and scale-invariant feature transform (SIFT).

Finally, when comparing the extracted features with those in each database, convolutional neural networks (CNN) and k-nearest neighbour (K-NN) are used commonly. The CNN approach is mentioned by several studies examined herein.

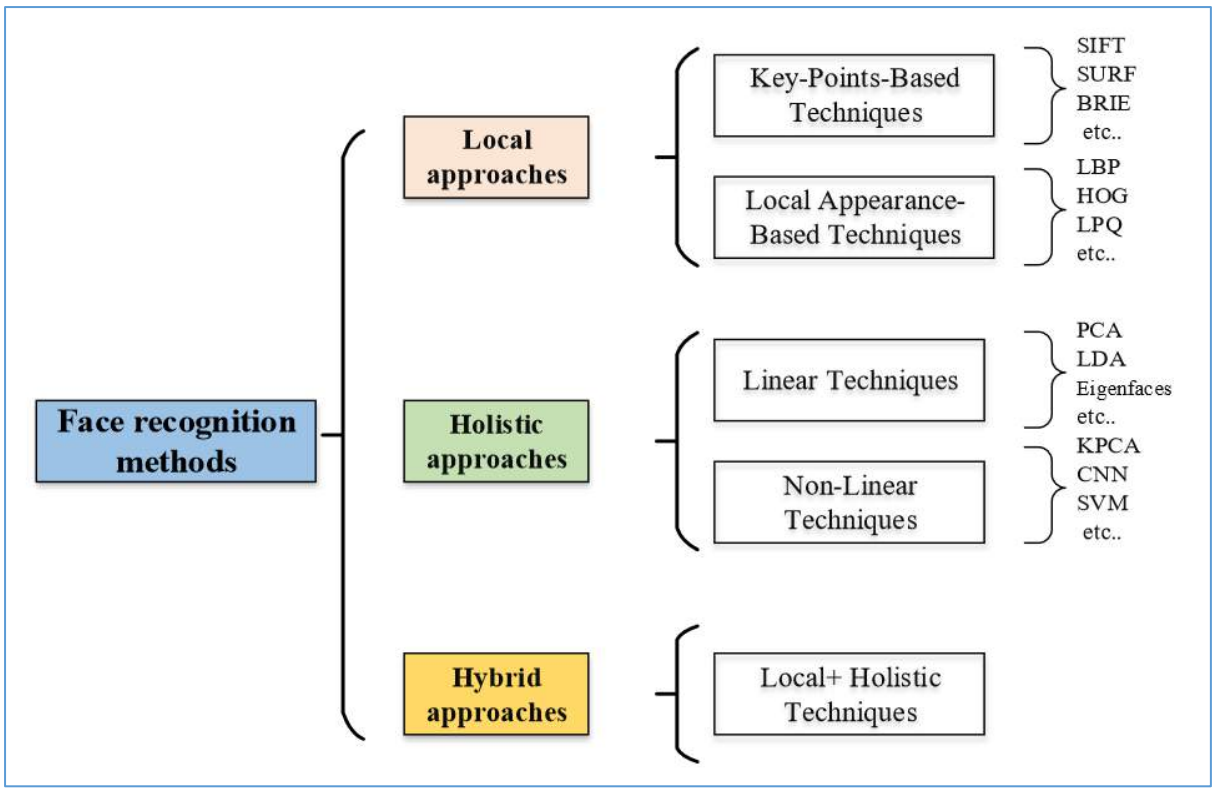


Figure 1 - Common algorithmic approaches to facial recognition systems

Throughout this literature review, studies will mention these algorithms and their merits and shortcomings for their respective systems. The diagram above from Kortli, et al. illustrates facial recognition methods which are mentioned throughout this review.

2.3 Leveraging Facial Recognition Technology to Identify Animals

2.3.1 Real-Time Goat Face Recognition Using Convolutional Neural Network

Billah, Wang, Yu, & Jiang (2022) noted that existing methods for distinguishing between individual livestock animals include the implanting of microchips, ear tagging, paint branding, and other invasive methods. As we'll also see discussed by other studies, these methods can prove unreliable, time-consuming, expensive, and even physically harmful to the animals.

Due to the high similarity and lack of obvious distinguishing features, the problem of animal facial recognition is more complex. The authors note that "Human face recognition is one of the most active research areas in the computer vision community. Whereas, only a few studies investigated animal face recognition problems".

Attempting to apply facial recognition technology to identify goats, the authors composed two publicly available datasets for detection and recognition, training state-of-the-art convolutional neural networks (CNN) models using this dataset. For the task of face and facial landmark *detection*, the authors tested the YOLOv4 and YOLOv4-tiny real time object detection libraries, finding that YOLOv4-tiny was 8x faster than YOLOv4. This model was trained on 1,680 images labelled with 13,671 total bounding boxes, identifying the position of the face, eyes, nose, and ears.

Table 1
Macro average for performance evaluation.

Model	Image Preprocessing	Accuracy (%)	Precision (%)	Recall (%)
Our CNN	Yes	96.4	96	95
	No	93	91	92
VGGFace (Fine tuning)	Yes	95	93	94
LBPFace	Yes	92	92	92
Fisherface	Yes	83	84	82

Figure 2 - The comparative performance of LemurFaceID using various methods

Since the difficulty in recognising animals is the similarity of their facial features compared with humans, a custom CNN model was trained. Every image is aligned based on the position of the eyes, while the horns, ears, and background are removed, contrast is increased, and noise is reduced. Finally, a greyscale version of the image is fed to the CNN model.

The table in figure 1 above demonstrates that the custom CNN achieved greater accuracy than other common pre-trained facial recognition models, achieving 96.4% accuracy overall.

Interestingly, the other recognition models, which were trained on human faces, still performed near the same level of accuracy as the custom CNN, which was trained specifically on goats. The authors suggest that "this indicates that many of the same features which the network has learned to be useful in discriminating against human faces often help to discriminate against goat face".

Having proven that goat facial recognition is possible using deep learning, the authors believe the findings of this research may help to improve "animal health and welfare, individual monitoring, activity monitoring and phenotypic data collection."

With regards to shortcomings, the authors note that this work used only a single breed of goat, and wish expand the model to recognise more breeds in the future.

2.3.2 Locality Constrained Sparse Representation for Cat Recognition

Chen, Hidayati, Cheng, Hu, & Hua, 2016 highlighted that at the time of writing, there were 164 million pets in the US, 60% of which were cats. Like other authors, Chen, et al. noted that existing methods for marking cats can affect their behaviour due to stress, e.g., when having a microchip inserted. Therefore, "there is a clear need for an ideal method to identify individual cats reliably and permanently without adverse effects" which would "contribute not only for cat identification, but also tracing, ownership assignment, and cat population management".

Table 1. Comparison of traditional methods for cat identification [2].

	Traditional cat identification methods							
	Permanent				Semi-permanent		Temporary	
	Tattoo	Microchip	Ear tip/notch	Freeze brand	ID collar	Ear tag	Paint/dye	Radio transmitter
Reliability	* * *	* * * * *	* * * *	* * * * *	* *	* *	*	* *
Cost	* * *	* * * * *	* *	* * *	* *	* *	*	* * * * *
Visibility	*	-	* * *	* * * *	* * * * *	* * * * *	* * * *	-
Longevity	* * * *	* * * * *	* * * * *	* * * *	* *	* *	*	* *
Requires anesthesia	* * * *	* *	* * * * *	* * *	-	* * *	-	*
Invasiveness	* * *	* *	* * * * *	* * *	-	* * *	-	*
Risk of harm	* *	*	* * *	* *	* * * *	* * * * *	*	*
Accuracy	* * * *	* * * * *	-	* *	* * * *	* *	*	* * * *
Space-restricted	* * * * *	-	-	* * * * *	* * * *	* *	* * *	-
Uniqueness	* * * *	* * * * *	-	* *	* * * *	* * *	*	* * * * *
Instant identification	* *	-	* * * * *	* * * *	* * * *	* * * *	* * * *	*
Training required	* * * *	* * * * *	* * * * *	* * * * *	*	* * *	*	* * * * *
Database required	* * * *	* * * * *	-	* *	* *	* * *	-	* * * * *

* * * * * (Very high) * * * * (High) * * * (Medium) * * (Low) * (Very low) – (None / Not Applicable)

Figure 3 - Table highlighting the strengths and weaknesses of traditional cat identification methods

Knowing that the nose print pattern of a cat can be considered a unique identifier of the animal, like human fingerprints, the authors collected 700 cat nose images from 70 different cats and used a combination of sparse representation for image classification and an SVM classifier to propose a new way of identifying individual cats based on their nose print.

Madarkar, Sharma, & Singh, 2021 explain that facial recognition is a two-step process consisting of feature extraction and classification. The task of extracting features from a facial image is complicated by noise, occlusion, similarities in structure between faces, and variation in facial features. The methods proposed to overcome these problems include principal component analysis, local binary pattern, independent component analysis, fisher discriminant analysis, and sparse representation-based classification (SRC).

Of these methods, SRC has shown greater performance over the others in recognising an occluded face, and this method has been used in many computer vision applications beyond facial recognition including image denoising and deblurring. To obtain features from cat nose images Chen, Hidayati, Cheng, Hu, & Hua, 2016, converted cat nose images into grayscale, partitioned the images into patches, and employed a sparse representation of each image patch, since using the entire cat image would require training on large amounts of data. A support vector machine (SVM) classifier is then trained based on the features obtained from the original cat nose image.

The authors found that their model achieved an accuracy of 91.2%, but they observed that since many cats share nose print colours, if the size of the collected data set were increased, they would probably notice a dramatic decrease in the performance of the model**.** In the future, the authors plan to 'precisely localise the nose print region, which is considered as a unique identifier of each cat and extract more distinct visual information from this region'.

2.3.2.1 Cat's Nose Recognition Using You Only Look Once (Yolo) and Scale-Invariant Feature Transform (SIFT)

Of particular interest in this case is that two years following this work, a similar paper was published also from the National Taiwan University of Science which used the same dataset along with the YOLOv2 object detection system for identifying the cat nose in real-time, and then applying Scale Invariant Feature Transform (SIFT) to compare the similarity between the given cat nose and existing cat nose images in the database to find a match. The recognition accuracy in this case was 95.87%, exceeding that of the previous study. This work also highlights the possible use of a real-time animal identification system, given that at least once every 5 years, 15% of cat owners lose their pets in the US, and only 38% of people who lose their pets are reunited with them, of which only 3% are cats.

2.3.3 LemurFaceID: A Face Recognition System to Facilitate Individual Identification of Lemurs

This study by Crouse, et al., 2017 differs from those discussed previously in this literature review in that the others have dealt with domesticated animals. The authors of LemurFaceID faced different challenges given that they sought to use facial recognition technology to identify individual lemurs in the wild.

According to the authors, when performing ecological studies, it is necessary for researchers to collect information on individual animals for studies over large areas or long periods of time. However, this is particularly challenging to do for long-lived species. Capturing and tagging animals can be difficult due to the difficulty of capture and can be expensive when working with a large number of individuals. Furthermore, the authors note that capturing is stressful for the animals and can impose health risks such as broken bones and even paralysis. An alternative is identifying individuals from a distance by relying on individual researcher knowledge of variation between individuals, but this is difficult to integrate when multiple researchers are working on a study over a large area and requires extensive expertise.

The authors collected 462 images of 80 individual lemurs. These lemurs had already been identified by researchers and assigned a name or ID number. Images are then cropped and rotated based on eye location, converted to grayscale, a gaussian filter is applied, and the image is gamma corrected to eliminate "small-scale texture variations".

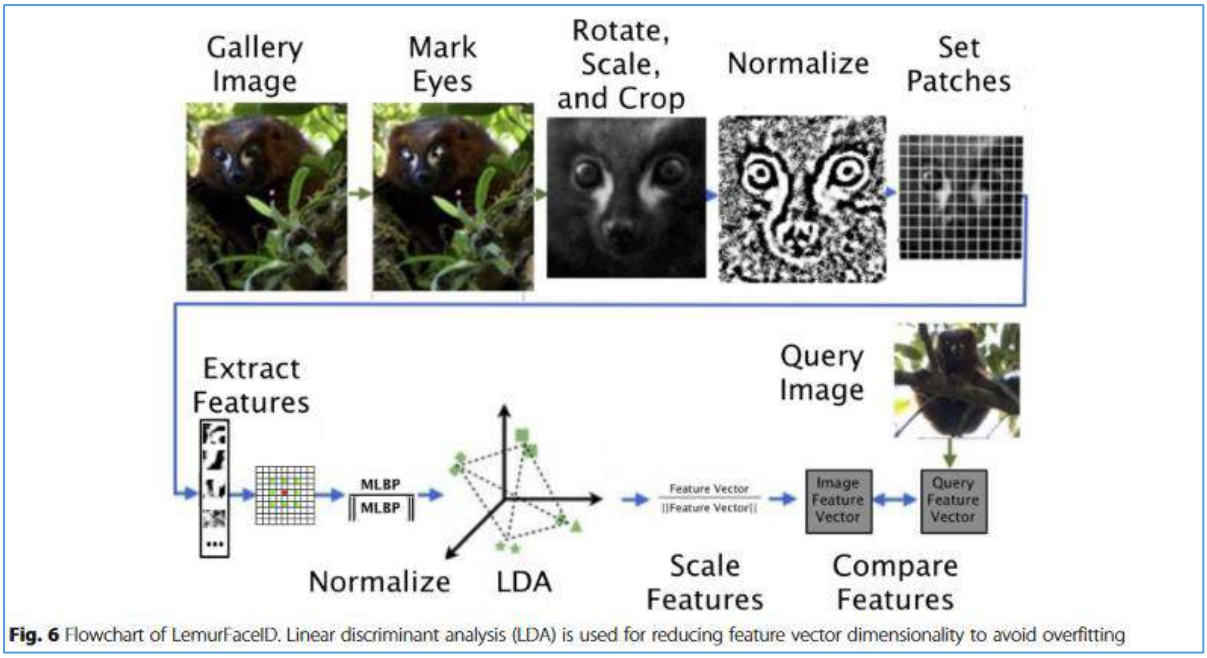


Fig. 6 Flowchart of LemurFaceID. Linear discriminant analysis (LDA) is used for reducing feature vector dimensionality to avoid overfitting

Figure 4 - The pre-processing, feature-extraction, and identification process

The authors used the OpenBR biometrics framework to facilitate the image normalisation process, illustrated in figure 3 above.

Local Binary Pattern representation was then used to split each image into areas where each pixel in the image is assigned a value based on its relationship with the surrounding pixels to find edges. This method allowed the authors to reduce the mean number of features per image from 396,850 to 7,305, while retaining 95% of the variation between individuals.

To identify a match, the database of lemur facial images is then searched, with each image having been normalised following the above process. Individuals are then identified by comparing the Euclidian distance between each feature vector in the given image and every corresponding lemur gallery image, where the highest similarity score represents a match, presenting the final 5 highest scores in descending order.

The authors found that individual lemurs can be correctly identified more than 95% of the time but note that their system uses a small dataset where there is much variation between individuals, which may yield different results in the wild, where two lemurs may appear very similar to one another. The dataset also consisted of known individuals to a study and may falter when dealing with lemurs which had never been seen before.

This study did not use CNN techniques, but the authors refer to a later study on chimpanzees which did use CNN techniques, noting that this study was able to improve upon recognition accuracy. However, the CNN technique required a much larger dataset, which was not available in this case.

The LemurFaceID system can recognise a lemur (using a quad core i7 processor) in less than one second, which shows the system's potential for time saving in ecological studies. The authors state that although there is currently no "one size fits all" solution to animal recognition, once such a system is developed it will enable researchers to answer questions which cannot currently be answered.

2.3.4 Towards On-Farm Pig Face Recognition Using Convolutional Neural Networks

This study by Hansen, et al., 2018 deals with domestic farm animals. The methods used for tracking individual livestock animals are similar to those used for household pets, but the authors claim that the need for an on-farm animal identification solution in particular has increased due to the intensification of sustainable farming practices. The most common method for identifying animals on the farm is the RFID tag, which is pierced into the pig's ear. The authors write that these tags have a maximum reading distance of only 120cm and are not entirely accurate. Even fitting two RFID chips to a single pig yields an accuracy of just 88.6% at close-range.

The authors created a custom CNN which was written the with keras, sci-kit learn, and tensorflow libraries which performed feature extraction as well as classification, but also tested the performance of the popular Fisherfaces, and a pre-trained VGG-Face facial recognition model, which were used in conjunction with a Support Vector Machine (SVM) for classification.

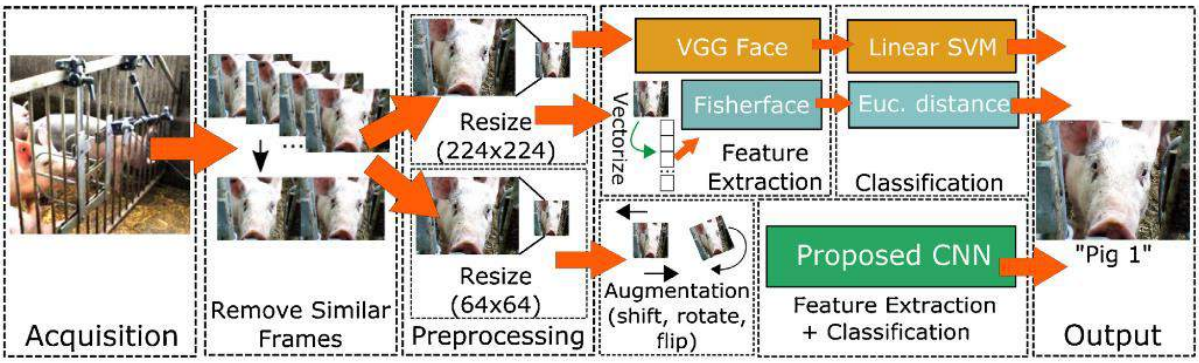


Figure 5 - The acquisition, pre-processing, feature-extraction and identification methods applied to pigs

The image pre-processing, feature-extraction, and classification process is outlined above. A web camera was mounted on the pigs' drinking nipple and connected to a laptop running the "iSpy Connect" camera software, which captured images when motion was detected. After discarding similar images, the dataset consisted of 1553 images of 10 pigs. As expected, the deep learning CNN approach performed best at 96.7%, compared with 78.4% for Fisherface, and 91% for VGG-Face in combination with the SVM.

As was noted in other studies, it is very interesting to see that the VGG-Face pre-trained model performed as high as 78.4% accuracy given that this model was trained only on human faces. The authors note that this "hints toward how a trained network for faces in one species may be transferable to other species, at least in so far as using it to bootstrap a new model for a new species using a reduced training set". The system performed well regardless of aspects such as pose, dirt, and lighting, which were not controlled.

The authors suggest using an overhead camera in combination with a 3D camera for modelling an animal's size and weight, to avoid the need to mount a camera on the pig's drinking straw. It is also unclear how well the system would perform when dealing with variations such as very dirty faces or tear stains below the animal's eyes. It remains to be seen how well the system would perform with the absence of any markings to distinguish the individual pigs, given that the pigs in this experiment were unique in that sense.

2.3.5 Identification of Animal Individuals Using Deep Learning: A Case Study of Giant Pandas

This study by Hou, et al., 2020 specifically refers to the aforementioned LemurFaceID system. The authors note that "Most studies on behavior and ecology of wild animals requires that research subjects are individually recognizable", which at first glance is obviously the case in the case of lemurs, but when attempting to use deep learning to recognise individual giant pandas, the difficulty is immediately clear in that giant pandas have no discernible difference in individual appearance. Pandas lack both expressive faces and unique individual coat patterns.

The same reasoning exists behind developing such a system as was noted by the authors of LemurFaceID, in that studies across large areas or timeframes require many experts capable of identifying the animals individually, or the use of physical tracking methods like RFID tags, which can be expensive and unreliable at times. Hou, et al. note that a common past method of identification specific to the giant panda has been the GPS collar, the service life of which is about 2 years, and can be prohibitively expensive for studies involving many individuals. Furthermore, the giant panda is by nature a secretive animal, difficult to locate in the wild.

In attempting to develop a novel system to identify individual giant pandas, the authors collected images from 25 captive pandas, featuring pandas engaged in a variety of activities, e.g. walking, eating, sitting to make the system more robust. The total data set included 65,000 images of giant pandas from Chinese panda breeding research bases, with about 4300 images per individual. The dataset was divided into a 6:3:1 ratio for training, validation, and calibration of the model. Some images were given a treatment to simulate dirty faces and poor lighting, by increasing saturation and brightness.

The VGGNet identification network, a type of CNN, was used to identify the pandas. The model consisted of five convolutional modules, 3 fully connected layers, and a soft-max layer. The system first extracts the image features, then compresses the image while retaining its main features, and the extracted features are then classified. The soft-max layer finally conducts probability mapping to determine which individual from the dataset is in the given image.

Their model achieved 95% accuracy based solely on facial images, and over 90% accuracy when low photo quality was simulated. The model can also identify individuals not from the dataset with 78.7% accuracy.

The authors note that **"**although giant pandas lack obvious characteristics for individual identification because their whole body is composed of only black and white fur (Li et al., 2018; Norouzzadeh et al., 2018), image recognition based on deep learning has shown good identification performance in similar circumstances". They found that the need for large rotation angles in an image greatly reduced identification accuracy, so suggest that in future studies, cameras should be positioned near an animal's head. They also found that the more pixels present in an image, the more accurate identification can be achieved, so recommend choosing cameras with high resolution.

A system such as this could prove invaluable to ecological research around the world, given the widespread use of infrared cameras to track animals. This system could make it much easier to track rare and endangered animals across wide areas and timeframes.

2.3.6 Deep Learning Framework for Recognition of Cattle Using Muzzle Point Image Pattern

Kumar, et al., 2017 proposed a system using “muzzle point image pattern (nose image pattern)” for recognition of individual cattle using a deep learning approach. At the time of writing, no such animal biometrics-based recognition system existed in literature or in the public domain. Like the aforementioned study on recognising cat noses, the cow’s muzzle is considered very similar to the human fingerprint. The authors collected a dataset of over 5000 images of cattle muzzles, consisting of over 500 cattle.

Kumar, et al. believe that deep learning-based approach are ideal for addressing the significant feature variations in the muzzle points of cattle, especially in uncontrolled conditions involving occlusions such as poor lighting and blurred images to the animal’s movement. Their proposed system uses a CNN and a Deep Belief Network (DBN) to *learn* the muzzle point texture of cattle for the purpose of individual identification.

Similar to other studies mentioned in this literature review, images were pre-processed to negate problems relating to poor illumination and image quality. Images are transformed to greyscale “to mitigate artefacts and noise from muzzle point images”. Images were also improved using “contrast limited adaptive histogram equalization (CLAHE)” to improve the contrast between muzzle ridges in the images.

A common current technique for identifying cattle is ear-tagging. Ear tags can vanish easily and can cause damage to the ear over long periods of time. It’s also common to create sketches of a cow’s fur pattern for later reference, but accuracy depends on the skill of the person creating the sketch. Inaccurate manual cattle recognition causes problems for “breeding association, registration and health monitoring of livestock”.

In a comparison of approaches, the author found that CNN, SDA, and DBN based learning approaches performed similarly, with their accuracies increasing as the number of features included in muzzle point images was increased. CNN, SDAE, and DBN yielded 95.98%, 96.92%, and 98.99% accuracy when given 400 feature sets, respectively.

Table 1
Illustrates the identification accuracy (%) of CNN, SDAE, and DBN deep learning approaches.

S. No.	Proposed approach	Identification accuracy (%)
1	CNN	75.98
2	SDAE	88.76
3	DBN	95.99

Table 2
Identification accuracy (%) of CNN, SDAE and DBN deep learning approaches.

Number of feature sets	Identification accuracy (%)		
	CNN	SDAE	DBN
50	63.75	67.75	65.95
100	67.98	68.65	69.85
150	73.85	71.96	75.85
200	76.75	76.92	77.94
250	79.98	78.67	82.99
300	82.99	85.98	86.92
350	86.75	89.79	94.75
400	95.98	96.92	98.99

Figure 6 - Comparative performance of CNN, SDAE, and DBN methods in identifying cattle using muzzle point images

The authors also tested 'handcrafted texture features-based representation algorithms' including Local Binary Pattern and Circular-LBP and found that they performed to an accuracy of 16.80% and 26.97%, respectively. Overall, the author found that their proposed deep learning framework-based approach outperforms the older handcrafted feature descriptor methods, such as LBP, which are used by studies mentioned previously in this literature review. This study seems especially relevant given that the author achieved an individual recognition time of 10.25 seconds.

The authors would like to take a multi-modal approach to cattle recognition in the future, using both muzzle images and face images for identification and verification of cattle in real-time.

Finally, Kumar, et al. would also like to increase the size of their cattle database in the future, for validation of their results benchmarked with existing handcrafted texture descriptor techniques compared with deep learning-based approaches.

2.3.7 Cat Face Recognition Using Deep Learning

Given that in Taiwan, more than 3 thousand cats and dogs go missing every year, Lin & Kuo, 2018 created a cat database consisting of 1500 images of 150 different cats. A CNN was used to identify the eyes, nose and mouth of the cats. The Eigenfaces algorithm was used to quality the facial features which were then used to identify the cats with support vector machines, achieving an accuracy of 94.1%. Unlike the study by Chen et al., which identified cats using their nose prints, this study used entire cat faces.

The authors used two faster R-CNN models. The first was used to detect the presence of cat faces in the images and define a bounding box for its location, cropping the image accordingly, and the second was used to locate the facial features within that bounding box.

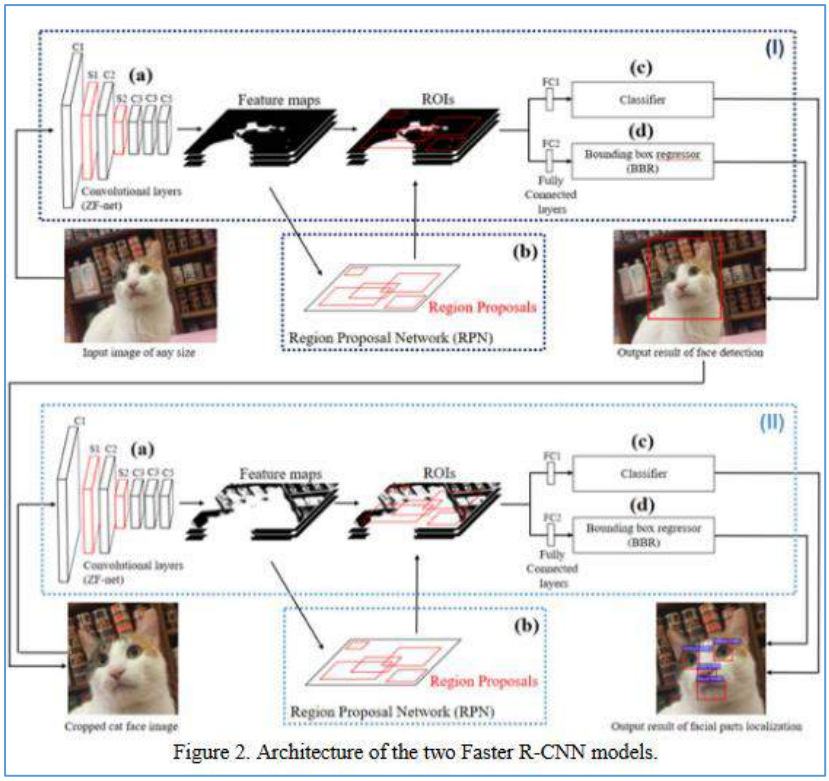


Figure 7 - Illustrating the architecture behind using two Faster R-CNN models for cat identification.

The model even succeeded in identifying three ginger cats which were very similar in appearance at first glance but had slight variations in colouring when viewed more closely.

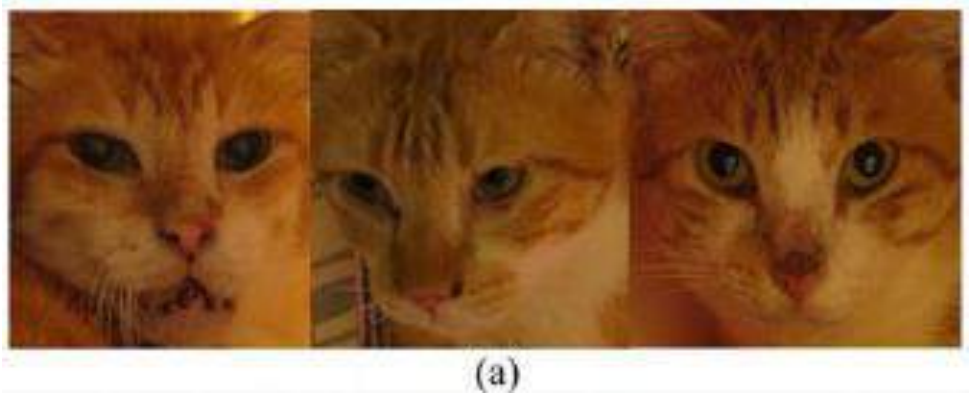


Figure 8 - Three ginger cats with few distinguishing features which the system had difficulty differentiating.

The system had the most difficulty in distinguishing black cats, which had very little variation in their features.

2.3.8 Automatically Identifying, Counting, and Describing Wild Animals in Camera-Trap Images with Deep Learning

This work by Norouzzadeh, et al., 2018 differs from the others in that the proposed system not only identifies species, but also attempts to count the number of animals in an image, 'describe their behaviour, and identify the presence of young'. However, some methods used in this system may not be applicable to my own, given that the dataset used was extremely large, with the training set alone containing 1.4 million images.

Similar to the DNN cat facial recognition system, this work utilised a two-stage pipeline, whereby the first stage decides whether an image is empty of animals. The second stage consists of three related additional tasks trained within a single model, which is identifying the type of animal present, describing their behaviour, and identifying whether young are present. The author found that learning multiple related tasks in parallel can improve performance on each individual task, and so solving a task is faster and more energy efficient.

Table 1. Performance of different deep learning architectures

Architecture	No. of layers	Short description
AlexNet	8	A landmark architecture for deep learning winning ILSVRC 2012 challenge (31).
NiN	16	Network in Network (NiN) is one of the first architectures harnessing innovative 1 × 1 convolutions (49) to provide more combinational power to the features of a convolutional layers (49).
VGG	22	An architecture that is deeper (i.e., has more layers of neurons) and obtains better performance than AlexNet by using effective 3 × 3 convolutional filters (26).
GoogLeNet	32	This architecture is designed to be computationally efficient (using 12 times fewer parameters than AlexNet) while offering high accuracy (50).
ResNet	18, 34, 50, 101, 152	The winning architecture of the 2016 ImageNet competition (25). The number of layers for the ResNet architecture can be different. In this work, we try 18, 34, 50, 101, and 152 layers.

Figure 9 - The performance of various deep learning architectures in identifying wild animals using camera-trap images.

Several deep learning architectures were tested for the first stage of identifying whether animals are present in an image, of which the VGG model achieved the highest accuracy of 96.8%.

For the task of identifying the species present to match one of 48 possible species, the authors identified the top-1 accuracy and top-5 accuracy scores, with the former being a guess at a solution, and the latter the top 5 guesses made by the model. The highest performing single model in this case was ResNet-152, which achieved 93.8% top-1 accuracy, and 98.8% top-5 accuracy.

The third step of counting the animals in the image performed more poorly by comparison, as literally counting the objects in an image requires labelled bounding boxes, which this dataset did (Sajjad, et al., 2017)classed as having 1-10, 11-50, or 51+ images, for which the best model was also ResNet-152, achieving 62.8% top-1 accuracy, and 84.7% top-5 accuracy.

Determining the behaviour of animals in an image is considered a multilabel classification problem, as animal behaviours are not mutually exclusive, i.e., an animal could be moving and eating at the same time. ResNet-152 again performed best, at 75.6% overall accuracy.

The authors used data from the SS Project, the world's largest camera-trap project, consisting of '225 camera traps running continuously in Serengeti National Park, Tanzania, since 2011', which contains 1.2 million images of 48 different species.

Having tested the accuracy of several state-of-the-art deep neural network methods, the authors found that their system could save 99.3% of human labour (over 17,000 hours) identifying animals by hand, to the same level of accuracy at 96.6%. The VGG and ResNet deep neural network models were found to be most accurate in this work.

The authors note that their system is incapable of identifying multiple species in the same image and will instead only label a single species. A possible future improvement is the development of the system to identify multiple species in a single image.

The authors also question the value that this system will have in alleviating the workload for human volunteers identifying the animals in the captured images, given that labelling accuracy could be negatively impacted if humans take the AI's suggestions at face value, despite them not being 100% accurate.

2.4 Applying the Aforementioned Methods to the Raspberry Pi

Sajjad, et al., 2017 note that the human face is a very good metric for the identification of individuals along with other biometric solutions such as fingerprint and iris scanning. We have seen from the literature discussed previously that these metrics are also applicable to animals. We must keep in mind that the goal of this project is to apply such a system to the Raspberry Pi, a small single-board computer with limited hardware resources. The authors note that the Raspberry Pi's limited hardware resources including its memory, storage space, and processing power pose a limitation on its ability to perform a task as complex as facial recognition. Even in the aforementioned studies on animal facial recognition, the authors have performed the training aspect of their facial recognition models on more powerful hardware, usually desktop computers with high-end graphics processing units. Therefore, the authors of this study propose using the Raspberry Pi in combination with a cloud-assisted facial recognition framework. This study is of particular interest to this project, as the proposed animal facial recognition system running on the Raspberry Pi is expected to require significant hardware resources, meaning a cloud service will likely be applied for more resource-intensive aspects of the system.

Sajjad, et al.'s system applies specifically to suspect recognition in aid of law enforcement, particularly in third-world countries where expensive hardware is not available. The authors note that methods such as CNNs for face detection and SVMs for classifying faces are computationally expensive methods which would be less suitable for real-time detection and recognition on a smaller device like the Raspberry Pi. Therefore, their proposed method uses Bag of Words in combination with the ORB (Oriented FAST and Rotated BRIEF) feature extraction method, which they suggest is more suitable for such a system. The system furthermore uses a 'cloud-based ensemble support vector machine (ESVM) for suspect recognition. In their proposed system, a three-step process is used. A live video feed is streamed from a camera to the Raspberry Pi, which uses the Viola Jones algorithm to detect faces. Feature extraction is performed with the aforementioned BoF and ORB methods, and the cloud service running a support vector machine is used to identify faces to correspond with those existing in the database.

Table 2
Results of the proposed method on face-95 with different number of images, cross validation size, and k-size.

No: classes	Training	k-size	Cross validation	No: of features	Average accuracy
70	30	500	0.7	364 320	100
70	30	500	0.3	850 080	99
70	05	500	0.7	60 720	92
70	05	500	0.3	121 440	89.3

Table 3
Results of the proposed method on face-95 with different cross validation and k-size.

No: classes	Testing	k-size	Cross validation	No: of features	Accuracy %
70	30	500	0.3	179 162	96.8
70	30	500	0.7	422 163	98.4
70	05	500	0.3	32 219	86.3
70	05	500	0.7	61 417	88.7

Table 4
Results of the proposed method on face-95 with different no. of images and cross validation and k-size.

No: classes	Training	k-size	Cross validation	No: of features	Accuracy %
33	20	500	0.7	858 000	98.7
33	20	500	0.3	330 000	95.0
33	05	500	0.7	132 000	89.1
33	05	500	0.3	66 000	79.1

Figure 10 - The resulting performance of the proposed Raspberry Pi-based facial recognition system

The Face-95 image database created by the California Institute of Technology was used in this study, which consists of images in different lighting variations, facial expressions, and backgrounds. Images were tested with Gaussian, motion, and median blur to simulate the effects of movement on recognition accuracy. The resulting performance of the system was highly accurate, despite the hardware limitations of the Raspberry Pi. The results are shown in the above table.

The authors note that they did not test the performance of CNN and deep belief networks in this study, and in the future, they plan to conduct a detailed analysis of the performance of real-time facial detection algorithms on the Raspberry Pi. Despite its limitations, the results of this study bode well for the application of the previously mentioned animal facial recognitions on the Raspberry Pi, albeit with the help of cloud services.

2.5 Conclusion

The main question this literature review sought to answer was whether it was possible to use facial recognition technology not only to identify a specific type of animal, but to distinguish one *specific* animal from another. The proposed project also seeks to apply this technology to the Raspberry Pi single-board computer. It is clear from the existing literature written over the past several years that many varying approaches have proved very effective at distinguishing individual animals in or near real-time, which bodes well for the feasibility of this project. Furthermore, studies have shown that though the Raspberry Pi may not be suitable for performing the complex calculations required for facial recognition, cloud services may still be leveraged to augment the abilities of the Pi, facilitating the development of our proposed system.

3 Requirements

3.1 Introduction

The proposed system is a web application intended primarily for mobile use which will be connected to a Raspberry Pi single-board computer, possibly running the MotionEyeOS operating system. Mounted on the Raspberry Pi will be a webcam and a piezo buzzer component.

The application should enable a user to upload photos of their pet to a database and train an image classification model to recognise the animal. Once the system can recognise the user's pet to a reasonable degree of confidence, it should be possible to 'whitelist' the users' pet, while deterring any unwelcome animals using the system's piezo buzzer. For example, the user may position the Raspberry Pi so a pet's food bowl is in view of the camera, in order to allow their pet to eat from the bowl, but deter other animals, e.g. stray cats.

3.2 Requirements gathering

3.2.1 Similar applications

As was noted previously in some of the literature quoted in the research chapter, the proposed system is a novel concept, and so very few applications exist in literature or the public domain with features similar to those of our proposed project. However, we will discuss some of the most similar existing applications, their advantages and disadvantages to gain insight into how this project might be developed.

3.2.1.1 – A Cat Identification System Using Raspberry Pi and Tensorflow

The author of this project used a Raspberry Pi running MotionEyeOS and an infrared camera to identify which of their two cats was in a litter box. MotionEyeOS was configured to automatically back up captured videos to Google Drive, with clips being overwritten every 30 days. For training the machine learning model, Google Colab was used due to the convenience of mounting an existing Google Drive disk to fetch videos, and the availability of a free GPU runtime. Data was labelled by first converting video to images, then labelled using a simple UI the author created using IPyWidget, a Jupyter Notebook library for creating interactive elements such as text inputs and sliders. 1500 images were labelled in total.

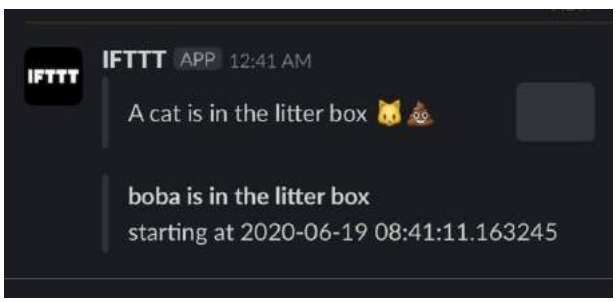


Figure 1 - Slack notification telling the user which cat was detected.

Due to the storage limitations of the Raspberry Pi, the Xception image recognition model was used, which requires less than 100MB of storage space. With 20 epochs of training, the author's model achieved 100% training accuracy, and 97.5% validation accuracy. The author found that when running on the Raspberry Pi, Tensorflow was too slow, so instead used Tensorflow Lite, which is optimised for use on mobile and edge devices like the Raspberry Pi.

The author was unable to perform predictions on the same Raspberry Pi she used to run the webcam, so instead used a second Raspberry Pi to fetch images from the camera-equipped Pi and perform predictions.

The entire architecture was setup as follows:

1. The Raspberry Pi running MotionEyeOS sends a cURL request to the Home Assistant API when motion is detected.
2. A command line switch (an integration specific to the Home Assistant API) is used to call a Python script, which in turn makes the prediction. The script fetches 10 seconds of video frames and performs a call to IFTTT with its result.
3. Finally, a webhook was configured within IFTTT to send a Slack message identifying the detected cat.

Advantages:

Using Tensorflow Lite with the Xception image recognition model, the author achieved very accurate detection results, and was able to load a model to the device in less than a second. The model seemed to be capable of accurately differentiating the author's two cats. This project is unique among the similar applications discussed here in that it can distinguish between two animals of the same species.

Disadvantages:

At 1500 images, a very large training set was required for training the model in this project. For my purposes, 1500 seems unrealistic, when we expect this application to be user-friendly in that a user should need to upload only a small number of images.

The author was unable to configure a single Raspberry Pi to simultaneously run MotionEyeOS for motion detection, and perform image recognition, and instead had to use two separate Raspberry Pi devices. They note that another possible approach is to install the Raspbian operating system with MotionEye, rather than using MotionEyeOS (an embedded system version) but found approach unstable.

3.2.1.2 Microsoft IoT Pet Door

This project from Microsoft uses a pre-trained OpenCV image classifier to detect the presence of a preferred pet in frames, allowing or restricting an animal's access via a pet door accordingly. For example, a user may wish to allow a dog to enter the house but deter raccoons and other wild animals. The system uses a Minnowboard Max running Windows IoT Core, a version of Windows 10 optimised for use on smaller devices. The authors note that a Raspberry Pi could also be used in place of the Minnowboard Max. A set of motion sensors connected to a pair of servo motors control the opening and closing of the pet door.

Figure 11 - A dog is granted access by the Microsoft IoT pet door.

When a pet triggers the motion sensor, the webcam is activated, capturing some frames of the animal which activated the sensor. The OpenCV classifier then identifies the type of animal in the frames.

Advantages:

I find the creative use of infrared sensors and servo motors in this project interesting. It may be possible to implement a similar system for our purposes, albeit not with a pet door specifically.

Disadvantages:

This system is only capable of differentiating between animal species and is incapable of classifying an individual cat or dog, for example. For our purposes, this is of little use. Furthermore, the YOLOv7 algorithm is trained on the COCO dataset, and so is capable of differentiating between cats and dogs with no custom training required.

In my opinion, this project isn't particularly impressive regarding its software but would likely be a genuinely useful consumer product for cat or dog owners, especially those living in areas where wild animals like raccoons or opossums are widespread.

3.2.1.3 Cat Identification

This project used a webcam-mounted Raspberry Pi Zero running MotionEyeOS, along with the Amazon's Rekognition image classification service to recognise when a cat is outside his door, waiting to be let inside. When the system detects the cat, it sends a push notification to the creator's phone.

Figure 2 - Raspberry Pi mounted to a window.

Articles written about this system claim that the image classification model was trained to recognise one specific cat only, having been trained on previous images of the cat. Unfortunately, however, after emailing Arkaitz Garro, the author, to ask how he achieved this, he explained that articles about this system have exaggerated its capabilities, and the system, and the system is only capable of recognising cats in general, and not his specific cat. Similar to the YOLO algorithm when using the pretrained COCO dataset version, AWS Rekognition will return a list of objects present in an image and its confidence of each one.

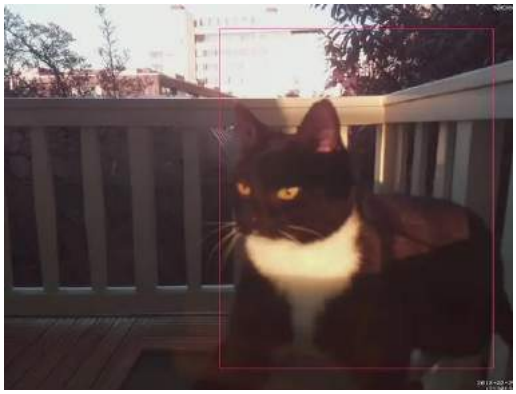


Figure 12 - The system recognises a cat outside the window

Advantages:

The creator felt that AWS Rekognition was very simple to use and did not require any additional training to classify images of his cat. The system also notifies the user via push notification when the cat is detected.

Disadvantages:

Like the Microsoft IoT Pet Door, the system is only capable of classifying images of cats in general, as opposed to identifying an individual cat.

3.2.2 Survey

A survey was conducted in order to inform the design and implementation of the system. 20 questions including diagrams and screenshots of the application's prototype were included in the survey, which was distributed to various animal forums and chatrooms. The survey includes general demographic questions and animal-related questions specific to pet owners, and provided respondents with the opportunity to list the features they believed to be the most important to the system, as well as to provide feedback on the design. 34 responses were received from pet owners in Ireland, Germany, Hungary, Lithuania, the Phillipines and the United States.

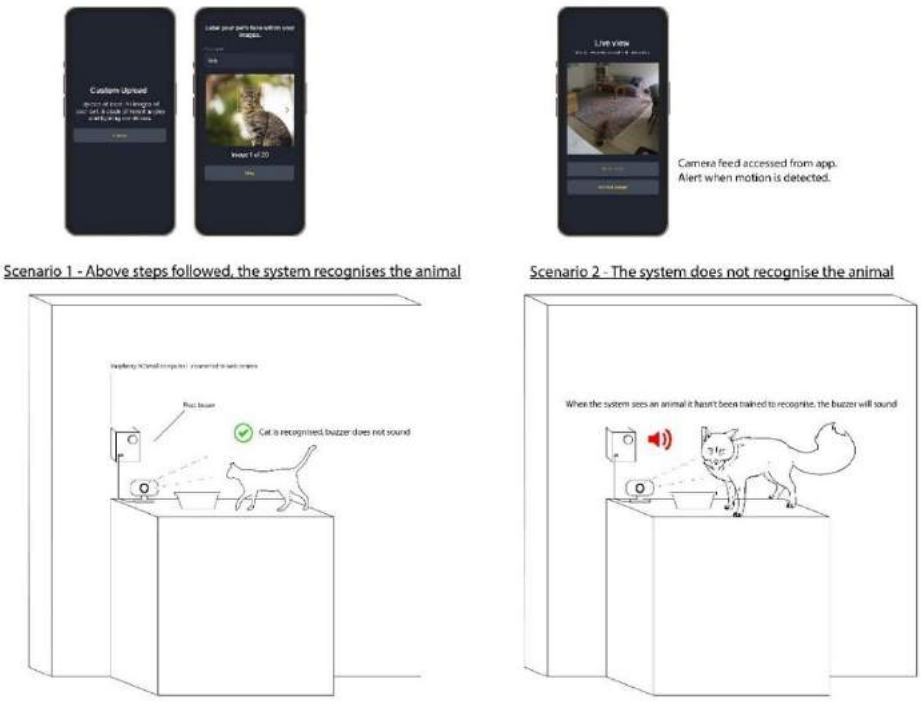
The following description and a link to the Figma prototype were included to introduce respondents to the project:

The proposed system will consist of a MERN stack web application, a Raspberry Pi mounted with a webcam and piezo buzzer, and the YOLOv7 object detection algorithm.

The idea is to allow the user to upload images of their pets to train a custom object detection model capable of recognising their pet specifically, unlike existing models which might only be able to recognise e.g., dogs or cats in general.

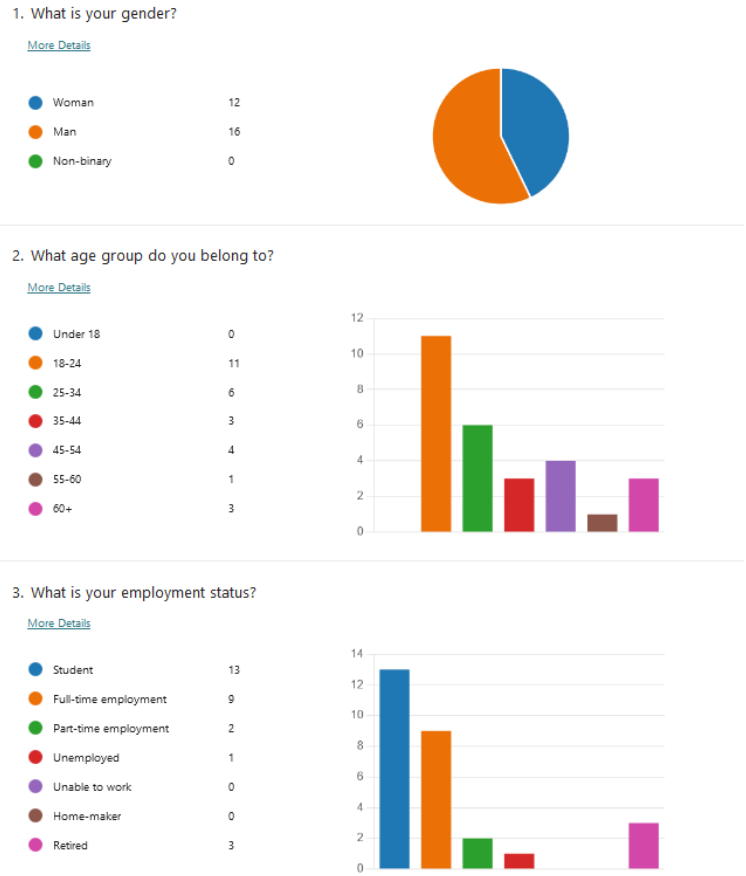
Once a user has trained a model to recognise their pet, the system could identify the objects passing in front of it and react accordingly. For example, the user could train the system to recognise their outdoor cat. By mounting the Raspberry Pi near the cat's food bowl, the system could then be used to deter stray cats or wild animals by activating the piezo buzzer when anything other than the user's pet passes by the camera.

To further ensure that respondents understood the concept of the project, the following diagram was included. In scenario 1, the user has uploaded and labelled images of their cat, which the system can now recognise. When the cat walks in front of the camera, the buzzer is not activated. In scenario 2, a fox passes in front of the camera, which the system does not recognise, and so the buzzer is activated.



Also included was a screenshot from a previous proof of concept using the YOLOv7 algorithm to illustrate the facial recognition aspect of the system:

The survey responses are summarised as follows:



Users were also asked what level of education they had attained. The options for this question were modified to accommodate respondents from the United States who were unfamiliar with the Irish school system, and so the visualisation created by Microsoft Forms was innacurate. The responses to this question were as follows:

- Bachelor's Degree – 5 respondents
- Leving Certificate/High School – 8 respondents
- Master's Degree – 3 respondents
- Ph.D or higher – 2 respondents
- Primary School only – 1 respondent
- QQI level 5/6 course or Associate Degree – 9 respondents

5. What kind of area do you live in?

More Details

Suburban area	12
City or Town	5
Rural area	11

6. What country are you from?

More Details Insights

24
Responses

Latest Responses
"Ireland "
"Ireland "
"Ireland"

Update

12 respondents (55%) answered **Ireland** for this question.

7. Do you have any pets?

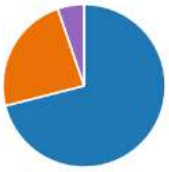
More Details

Yes	28
No	5

8. Which of these do you have?

More Details

Dog	27
Cat	9
Small animal, e.g. rabbit, guine...	0
Bird	0
Fish	2



9. Do you ever see stray cats or wild animals around your home?

More Details Insights

Yes	25
No	3

10. What kind of animals do you see around your home?

21 Responses

ID ↑	Name	Responses
1	anonymous	Fox, raccoon, opossum, songbirds, hawks, squirrels, groundhogs, bears, coyotes, bobcats
2	anonymous	Cats, foxes, pheasants, squirrels, hedgehogs
3	anonymous	Foxes; Deer; Stray cats; Mink; Pine Martin
4	anonymous	cats, fox, groundhogs, raccoons, squirrel, deer
5	anonymous	Fox, hedgehog, cats, deer
6	anonymous	possum, raccoon, skunk, deer, cat, rabbit
7	anonymous	Cats, rodents, birds, foxes
8	anonymous	Cats, foxes
9	anonymous	Cats, Rats, Foxes
10	anonymous	Stray Cats
11	anonymous	raccoons, coyotes, skunks, porcupines, birds, rabbits
12	anonymous	Fox badger hedgehog cow sheep chicken deer
13	anonymous	Many stray cats
14	anonymous	Foxes. Squirrels. Cats.
15	anonymous	Stray Cats
16	anonymous	Cats
17	anonymous	Stray cats, rabbits, hedgehogs
18	anonymous	Cats
19	anonymous	Foxes, Stray Cats
20	anonymous	Birds
21	anonymous	Cats

11. Do these animals ever interfere with your pets? E.g. harass them, eat their food.

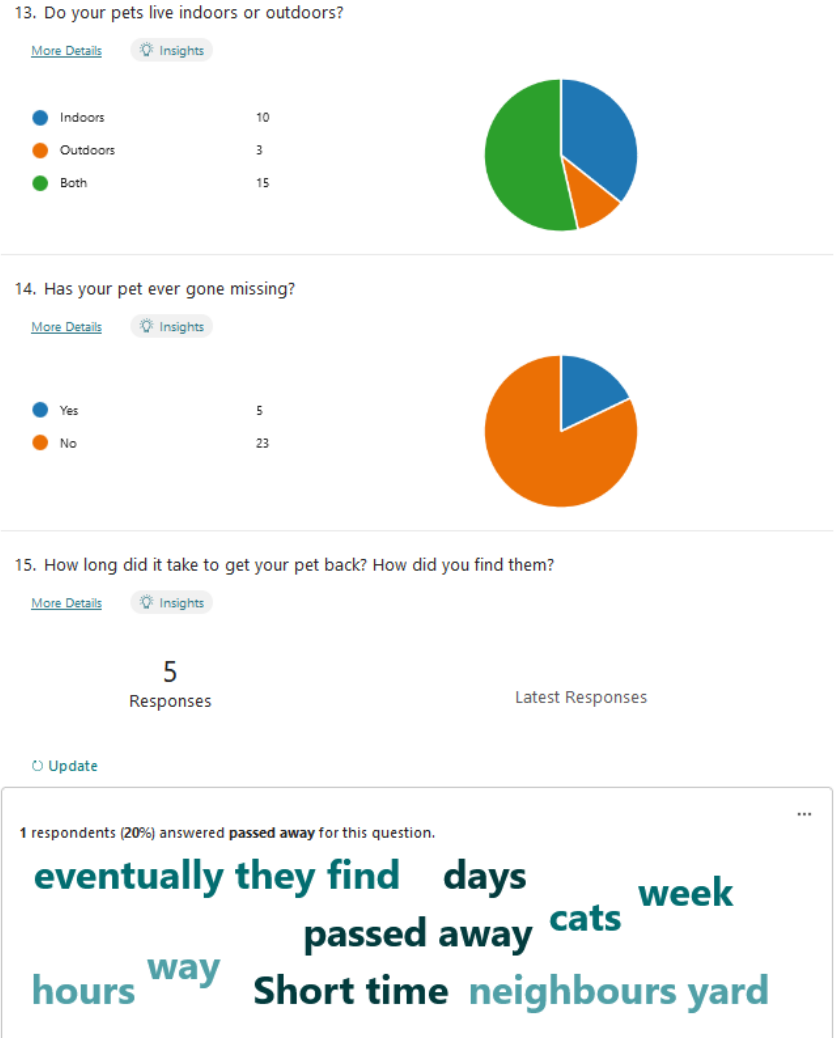
More Details

Yes	13
No	12

12. In what way do they interfere with your pets?

13 Responses

ID ↑	Name	Responses
1	anonymous	Stray cat will chase my cat and two dogs. Causes the dogs to bark loudly.
3	anonymous	Ocassionally harass outside, could eat their food if they got in the house
4	anonymous	Steal food; Get aggressive;
5	anonymous	Eat their food, sleep in their outdoor bed, attack them
6	anonymous	The birds like to tease the dog when she's in the garden, and I suspect that there are rodents travelling through the garden which my dog tries to catch.
7	anonymous	Eat their food
8	anonymous	porcupine spikes in face
9	anonymous	Harassment eat food sleep in their bed
10	anonymous	Eat their food and poop in my yard. Basically use my yard as a toilet.
11	anonymous	Eat their food
12	anonymous	Come into our garden
13	anonymous	Annoy them



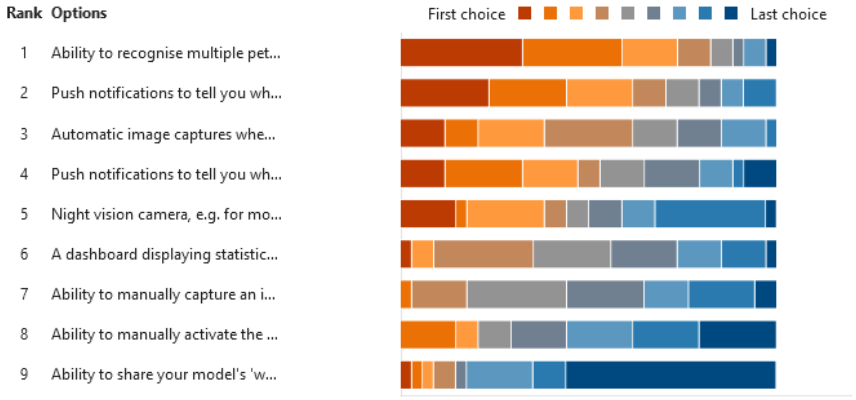
85% of respondents said they own pets, of which 68% (28) were dogs, 24% (10) were cats, and 7% (3) were fish. Fish were not included as an option initially, but I then realised some garden pond owners may be interested in using the system to deter birds and other predators from eating their fish, and so the survey was also shared to a garden pond enthusiast forum.

90% of respondents said they see stray cats and other wild animals around their home. Interesting responses were given by those from the United States, who claim to see raccoons, opossums, coyotes, bobcats, bears, skunk, porcupines and more around their homes. Irish respondents most often mentioned stray cats, but also included foxes, badgers, hedgehogs, and deer, among others.

Most respondents (55%) said their pets lived both indoors and outdoors. Of pet owners who see stray cats and other wild animals around their home, 50% said these animals harass their pets or eat their food. Respondents said their pet cats were chased by strays, who also aggravated their dogs. Several respondents said stray cats and wild animals steal food from their pets and sleep in their beds. One respondent from the United States even mentioned her dog was attacked by a porcupine, leaving him with spikes in his face. Almost 20% of respondents said their pet had gone missing at some point. Some said their pet went missing for a short time only, others for several days, and some never came back at all.

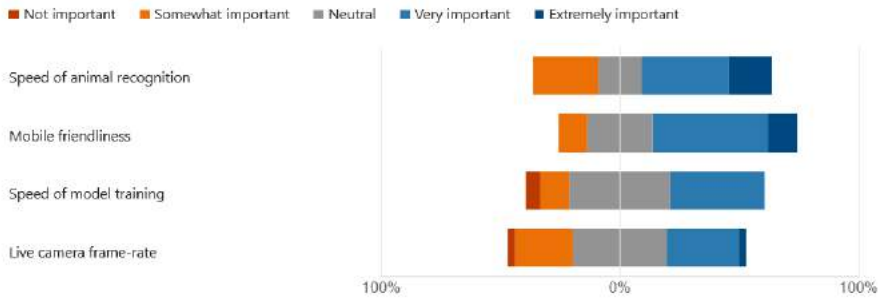
16. How desirable would you consider each of the following features? (Arrange by order of importance)

34 Responses



17. How important would you consider the following non-functional requirements?

More Details



The final three questions asked users to provide feedback on the UI design (A screenshot of all the application's screens was hosted on Imgur), suggest any additional situations in which the system would be useful to themselves or others, and provide their overall thoughts on the project, as well as any further comments they might have.

Most users liked the UI design, and praised the simplicity of the colour palette, the design's overall simplicity and cohesiveness. Users said the design featured 'Easily identifiable options' and liked the dark background.

Suggestions for improving the UI said the text is too small, and the images should be as large as possible. One user suggested differently coloured bounding boxes to distinguish between pets and strays, and another colour to represent a lost animal. The user suggested a notification system to alert users of missing pets in their area. Another respondent noted that it's unclear whether users need to specifically upload images of their pet's face, or simply images of their pet in general. One user commented that the colour is too dark, which suggests users would benefit from a switchable light/dark mode. Finally, with regard to the physical system, one user said the Raspberry Pi and camera should be durable to withstand the elements.

Several interesting suggestions were given for additional situations in which the system could be used. One respondent mentioned the airports have an issue with birds on the runway, causing a major hazard for planes taking off. The system could act as an early warning system for this issue.

Another interesting proposal was to use the system to help forest rangers studying a species' range, where multiple cameras could identify an individual over a large area. Some of the research papers mentioned previously mentioned that such a system is desired by scientists conducting ecological studies. Several respondents suggested the system be used to combat pet theft, in that if several people owned our proposed system. Finally, one respondent mentions that users may find the noise emitted by the device irritating, and suggests finding an alternate method of deterring animals.

The responses included some very interesting suggested features, though I consider most of them to be outside the scope of the project.

3.3 Requirements modelling

3.3.1 Personas

3.3.1.1 Samantha, 62, USA

Samantha is a 62-year-old retiree with a master's degree. She lives in a rural area in the USA and owns several small pups, who live both indoors and outdoors. She often sees stray cats and other wild animals around her home, including foxes, groundhogs, raccoons, squirrels, and deer. However, she doesn't find they often harass or interfere with her dogs.

For her, the most useful application of the system would be as a method for finding her small pups, who sometimes wander out into her back yard. In terms of design, Samantha feels that the images displayed should be as large as possible, and believes the ability to recognise multiple pets, like 'dog 1' and 'dog 2' is the most important feature, followed by push notifications, automatic notification of animal detection, and a dashboard for viewing statistical information about when animals are sighted.

Needs:

1. Large images she can see easily.
2. Ability to train the model to recognise multiple pets.
3. Push notifications when an animal is detected.
4. Dashboard for statistical information on animal sightings.

Key differentiators:

Like other respondents living in the rural United States, Samantha sees a lot of wild animals around her home but is unusual in that they don't interfere with or harass her small pups. Her response suggests she suffers with poor eyesight, as the most important feature to her is large images she can easily see and feels the most useful application of the system for her would be its ability to locate her pups if they escaped into her yard.

3.3.1.2 Joan, 50, USA

Joan is a 50-year-old woman living in the rural United States. She has a Ph.D. and is in full-time employment. As expected for someone living in the rural USA, she often sees raccoons, coyotes, skunks, porcupines, and rabbits around her home. These animals often interfere with her pet dog, who lives both indoors and outdoors. Her dog has even been attacked by a porcupine, leaving painful spikes in his face.

Joan has Google Nest cameras around her home and likes how they can recognize people from photos. She thinks a similar system for pets and animals sounds very interesting.

Uncommon among respondents was the desire for a night-vision camera capable of monitoring nocturnal animals, but Joan feels this feature is very desirable. She also feels a dashboard displaying previously captured images and the ability to train the system to recognise multiple pets are very important.

Regarding non-functional requirements, she feels mobile friendliness is extremely important for the system and rated the speed of animal recognition and model training as very important but didn't feel strongly either way about the live camera's video framerate.

Needs:

1. Night-vision camera for monitoring of nocturnal animals
2. Ability to train the model to recognise multiple pets.
3. Mobile friendly design.
4. A dashboard displaying previously captured images.
5. Fast animal recognition and model training.

Key differentiators:

Almost every respondent with an outdoor pet claims wild animals or stray cats can harass their pet, eat their food or sleep in their bed, but Joan's dog has been attacked by a porcupine, suggesting the system could be of use to her as a deterrent for porcupines particularly. Her dog also sometimes sleeps outside, so Joan is interested in outfitting the system with a night-vision camera for nocturnal monitoring.

3.3.1.3 Mia, 24, Germany

Mia is a 24-year-old woman with a high school education. She's a student living in an urban area in Germany. She doesn't own any pets, but feels the system has potential applications outside the home and suggests it could be useful for forest rangers to use several Raspberry Pi devices to monitor the range in which wild animals move around.

She suggests the facial recognition aspect of the system could be used to focus on specific animals or members of a herd. Mia likes the design of the app and finds the simplicity of the colour scheme pleasing to look at. She feels some clarity is needed in what purpose the buttons serve and suggests adding icons to provide a visual representation of what the buttons do, such as an arrow symbol within the 'upload' button.

If she did have pets, Mia suggests that the Raspberry Pi be connected to a latch which could be used to block a cat flap, ensuring that only your own cat could enter the house.

Among non-functional requirements, she feels the speed of animal recognition is extremely important, and places mobile friendliness and the speed of model training as very important.

Needs:

1. Greater clarity in the design, specifically the purpose of the buttons.
2. The ability to recognise multiple pets.
3. Push notifications and automatic image capturing when motion is detected, as well as a dashboard for viewing past captures.
4. The ability to share trained model weights online for other people to use.

Key differentiators:

One of the proposed features suggested in the survey was the ability for users to share their model's weights to a shared 'marketplace', where other users could access these weights in order to identify another user's pets on their own system. Mia found this feature interesting. One possible use case we can identify from Mia's suggestions would be the ability for park rangers or biologists to use this feature to conduct studies involving the tracking of individual animals over large areas.

In terms of the idea overall, Mia is impressed by the fact that the program can learn to recognise a specific animal face from only a few reference pictures.

3.3.1.4 Bríd, 28, Ireland

Bríd is a 28 year-old Irish woman living in a rural area. She has a bachelor's degree and is in full-time employment. She owns a dog and multiple cats, who live both indoors and outdoors. She often sees foxes, hedgehogs, stray cats, and even deer around her home. She finds that these animals often interfere with her pets and complains that they eat their food and sleep in their outdoor beds and have even sometimes attacked them. She's among of 18% of respondents who've had a pet go missing. She says she has owned cats in the past who never came back or wandered off to die. One of her cats could sometimes be gone for a week before returning.

Like many other respondents, she feels push notifications informing you when your pet was detected is very important, followed by the ability to recognise multiple pets, and the ability to share model weights with other users. For Bríd, mobile friendliness is very important, while the speed of animal recognition and model training only somewhat important.

She recommends displaying different colour bounding boxes to differentiate between pets and strays, and another that represents a potentially missing animal. Furthermore, she believes it would be useful to integrate a notification system informing pet owners that a lost animal may be within a certain distance, so the owner is aware that e.g., a cat is not a stray. While it seems to be outside the scope of this project, the also suggests incorporating a tracking system for pets which is built into their microchip to always track their location.

Needs:

1. Clear design choices to simplify the user experience.
2. Notifications informing the user when their pet is detected
3. The ability to recognise multiple pets

Key differentiators:

Bríd is a proponent of applying this system to reuniting owners with their lost pets. I find her suggestion of using different colour bounding boxes to differentiate between pets, wild animals, and potentially missing pets very interesting.

3.3.2 Functional requirements

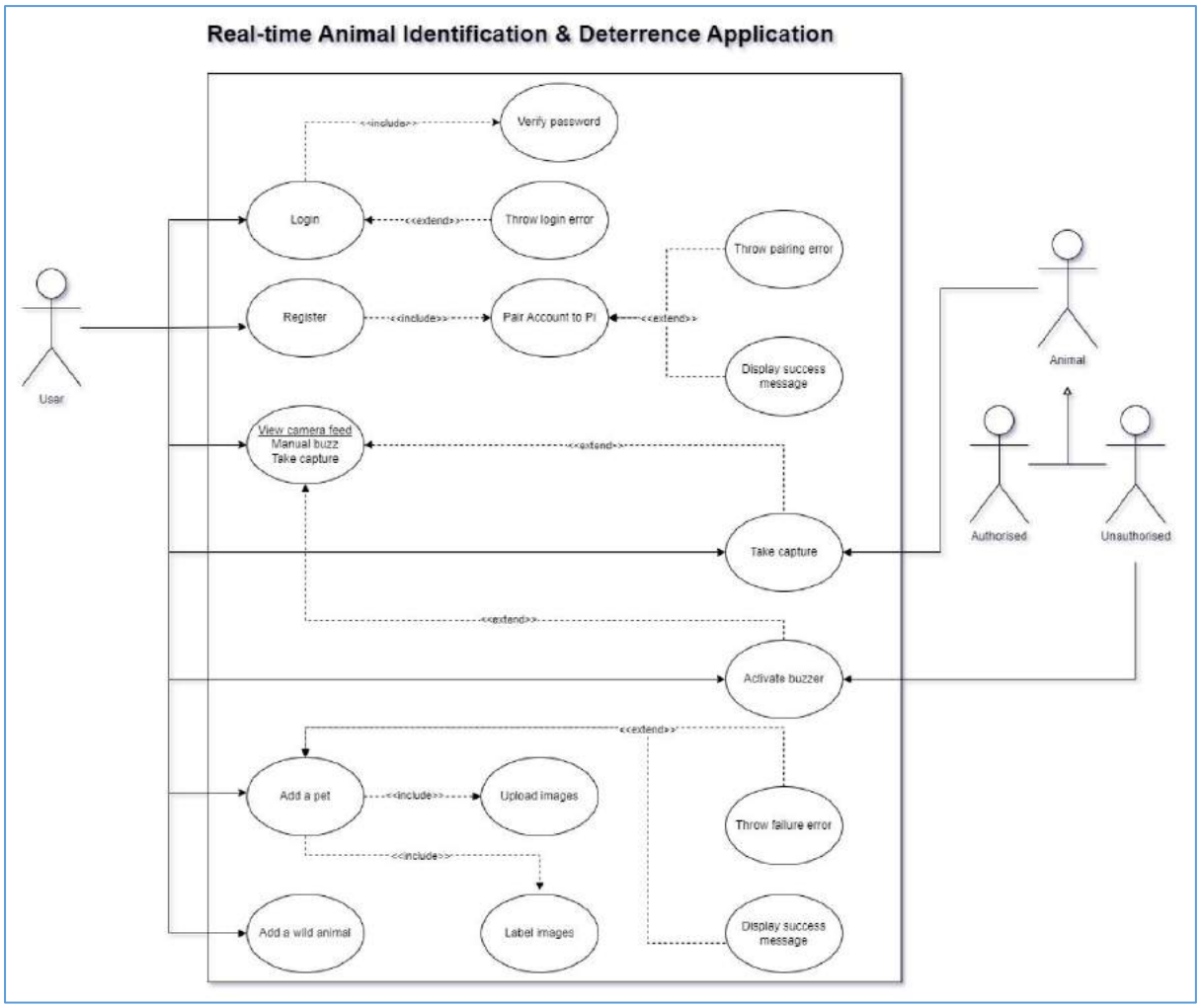
This is a list of the features required by the application, beginning with the most important.

Requirement ID	Requirement Definition
FR1.0.	Login/Register
FR2.0	Display unique QR code per user
FR2.1	Have Pi recognise QR code, pair to user account
FR3.0	Display live camera feed on dashboard
FR4.0	Pi responding to movement
FR4.1	Pi identifying animals in general
FR4.2	Pi can be trained to recognise specific animals
FR4.3	Pi deterring animals using piezo buzzer
FR4.4	Notifying user of animal detection (Push notifications)
FR4.5	Animal whitelisting settings

3.3.3 Non-functional requirements

Requirement ID	Requirement Definition
FR1.0.	Good user experience on mobile as well as desktop
FR2.0	Possible to train the model to recognise multiple animals
FR2.1	Ability to select pre-trained models, e.g. recognise various bird species
FR3.0	Add two factor authentication to ensure security, especially considering this system will have a live camera feed.
FR4.0	Ability to manually activate teh buzzer or manually capture images
FR5.0	Shared database where users can make their model publicly available
FR6.0	Dashboard – detection statistics, frequency of activation, previous captures

3.3.4 Use Case Diagrams



3.4 Feasibility

In terms of technology and functionality, the system can be divided into three segments. The user will interact directly with the progressive web application, which will be written in the React JavaScript framework and styled using TailwindCSS, a CSS framework providing utility classes to speed up development. Motion detection and image capturing will take place on the Raspberry Pi, which will run MotionEyeOS, a Linux distribution used to turn the device into a video surveillance system. Also running on the Raspberry Pi will be the NodeJS backend, which will be responsible for connecting the AWS services and React frontend. A small piezo buzzer will be attached to the device, which will be used for playing sounds to deter animals, similar to an ultrasonic animal deterrent.

Finally, a series of AWS services will power the animal recognition. AWS Simple Storage Service (S3) will be used to store images uploaded by the user for training, and images captured by the Raspberry Pi when motion is detected. An AWS SageMaker endpoint will be used to host the YOLOv7 algorithm. An endpoint will be configured using AWS API Gateway connected to a Lambda function, which will be used to trigger the SageMaker endpoint and run inference on an uploaded image.

In summary, the entire application will consist of a MERN stack progressive web application, a Raspberry Pi single-board computer running MotionEyeOS, the AWS S3, Lambda, API Gateway, and SageMaker cloud services, and the YOLOv7 single-stage object detection algorithm.

In terms of technical feasibility, my previous proof of concept work using YOLOv7 to identify a specific dog face was reasonably successful at nearly 60% recognition confidence, despite the model having been trained for only a short amount of time. Therefore, I am confident that the overall concept is technically feasible and will perform to a much higher standard given more training time. However, an issue may arise in attempting to run the NodeJS server on the Raspberry Pi once MotionEyeOS has been installed. In section 3.2.1.1, the author of the cat identification system notes that she had difficulty streaming the live camera feed and running inference at the same time, and resorted to using two separate Raspberry Pi devices. Should this issue arise in developing my own system, I believe it could be mitigated by using the MotionEye standalone programme, rather than the MotionEyeOS operating system.

3.5 Conclusion

In this chapter, we've discussed similar applications, discussed the results of the survey, determined user needs and key differentiators through their personas, identified the application's functional and non-functional requirements based on user feedback, visualised our proposed system with a use case diagram, and finally summarised the technologies that will be used to create the system.

To briefly reiterate the proposed system, a MERN stack web application, a Raspberry Pi mounted with a camera and piezo speaker, and a combination of several AWS cloud services will allow users to upload facial images of their pets to train a computer vision model to recognise their pet individually. With a model trained, the Raspberry Pi will detect motion, capture an image, and send it to the hosted model's endpoint for inference. If the animal in the image is not recognised as a pet, the piezo speaker will emit a high-pitched sound to deter it.

This chapter has served to provide clarity on the feasibility of this project, and offered insight into what users would expect of such a system. From examining some similar applications, we've found that some authors have had success differentiating between individual animals of the same species, and others have made creative use of the Raspberry Pi and other single-board computers to detect specific animals and to facilitate access control for pets. Studying how other people created smaller scale versions of what we aim to achieve with this project has made us aware of MotionEyeOS, and the possibility of using cloud services.

Distributing the survey forced us to gain a clear understanding of how the system will work in order to explain its premise to survey respondents quickly and simply, and so enhanced our own understanding of our goals in the process. Receiving feedback from pet owners around the world has been invaluable in informing the application's requirements, understanding which features are desirable, and features that users consider unimportant in our proposed system.

4 Design

4.1 Introduction

Given that the system consists of three separate but interconnected components, considerable effort must be put into the program design in preparation for beginning the implementation stage.

Having considered the findings from the research chapter, we will now create a more tangible plan for how exactly this system will be created. This plan will be divided into the program (application) design, and the UI design.

4.2 Program Design

4.2.1 Technologies

The technologies for the application can be considered to span three interconnected domains, consisting of the web application, the Raspberry Pi, and several cloud services. The user will interact directly with the web application, which will be written in the React JavaScript UI library, and styled using TailwindCSS, a library of CSS utility classes.

User data will be stored using MongoDB, a NoSQL, highly scalable database solution. Routing, authorisation, and communication with the AWS services will be handled by the NodeJS and Express backend server. NodeJS is a server-side JavaScript runtime environment, and Express is a web framework written for NodeJS. Express does not enforce any specific structure, but the Model View Controller (MVC) architecture will be used. The Node/Express backend will act as the middleman between the web application and the cloud services which will power the computer vision capabilities of the system.

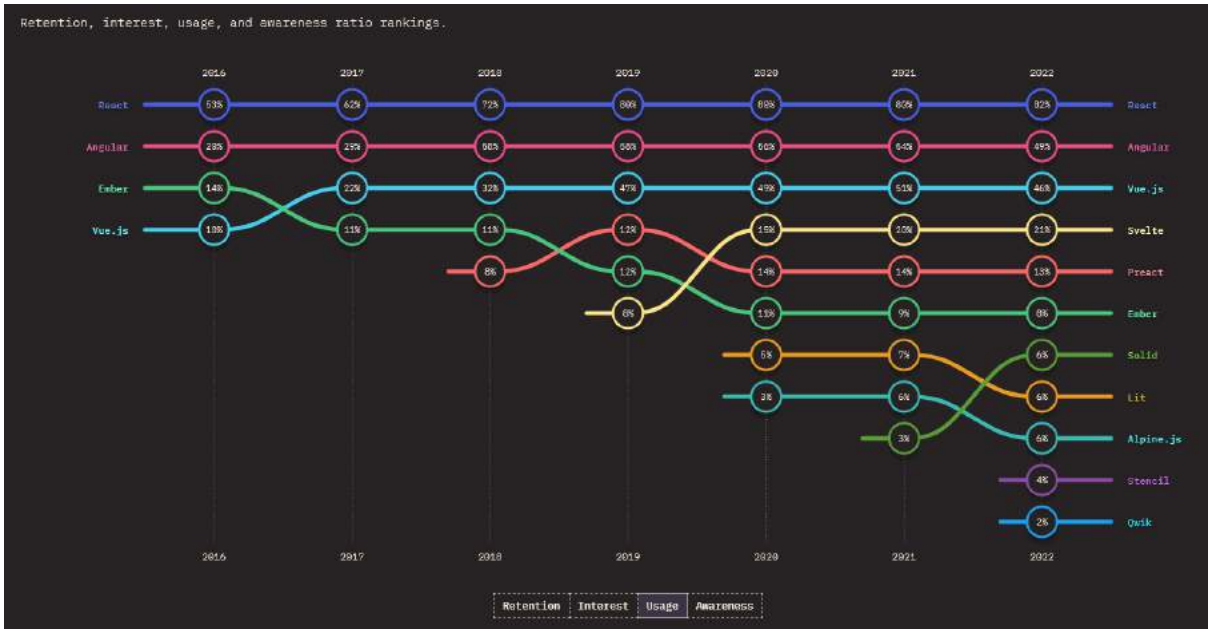
The server will run on the Raspberry Pi single-board computer. To capture live video, the MotionEyeOS operating system will be installed on the Raspberry Pi, which transforms the computer into a video surveillance system, capable of detecting motion and streaming video over Wi-Fi.

To train a custom computer vision model capable of recognising an individual animal, users will upload images of their pet to Amazon Simple Storage Service (S3), along with coordinates specifying the location of the animal's face. When has uploaded the required labelled images, a Lambda function will be triggered, training a new model using the YOLOv7 object detection algorithm, which will be hosted on Amazon SageMaker.

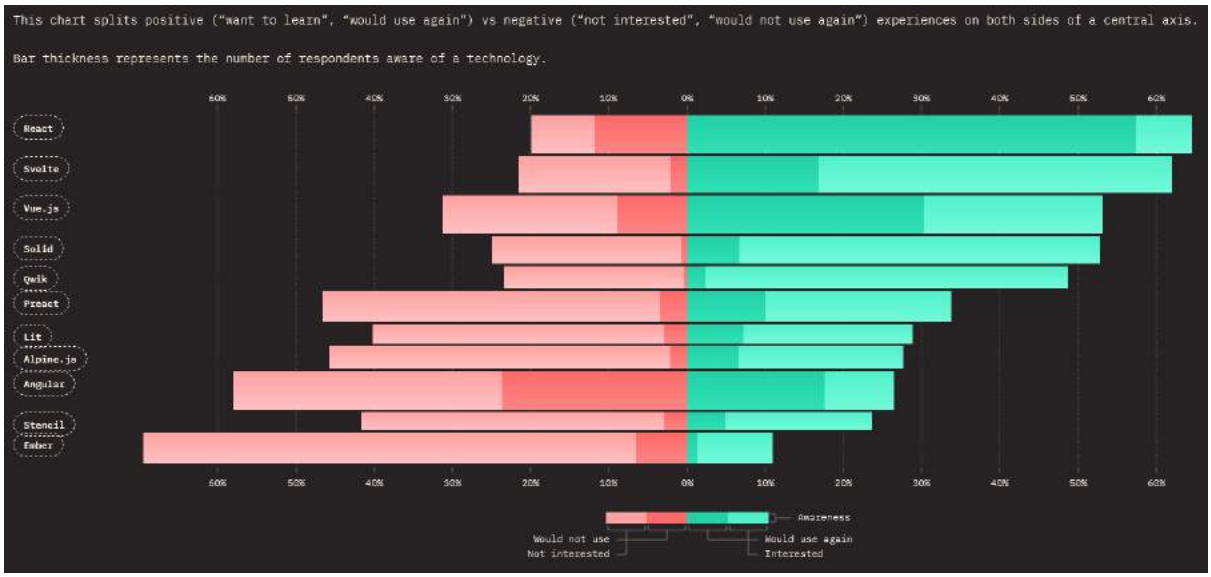
After this point, whenever motion is detected by the Raspberry Pi, a still image will be sent to the server, which in turn will upload the image to S3. Again, a Lambda function will be triggered, using the custom SageMaker endpoint to run inference on the image, and returning the result to the server, and finally providing visual feedback to the user via the UI. In summary, the cloud services powering the animal recognition consist of S3, Lambda, and SageMaker. It is unclear as yet if AWS API Gateway will also play a role in this functionality.

4.2.2 Reasoning for Using These Technologies

Alternatives exist to the MongoDB, Express, React, and NodeJS (MERN) stack, including the MEVN (Vue) and MEAN (Angular) stack, but MERN was chosen because I am most familiar with React. Given the complexity involved in building the entire system, too much time would be spent learning frontend technology. Furthermore, in the 2022 State of JS survey, in which almost 40,000 respondents took part, 82% of developers rated React as the frontend framework they use the most.



React also had the greatest proportion (57%) of developers who had used React before, and would use it again, compared with 30% and 18% for Vue and Angular, respectively.



In terms of the server-side language/runtime being used, there is an argument to be made in favour of alternatives to NodeJS. The below benchmark from Toptal tested the speed of NodeJS, Java, PHP, and Go when reading 64 kilobyte file, hashed it 1000 times using the SHA-256 algorithm, and printed the resulting hash in hexadecimal format. The request for this operation was running 300 times concurrently. The numbers above each column in the diagram below represent the time this took.

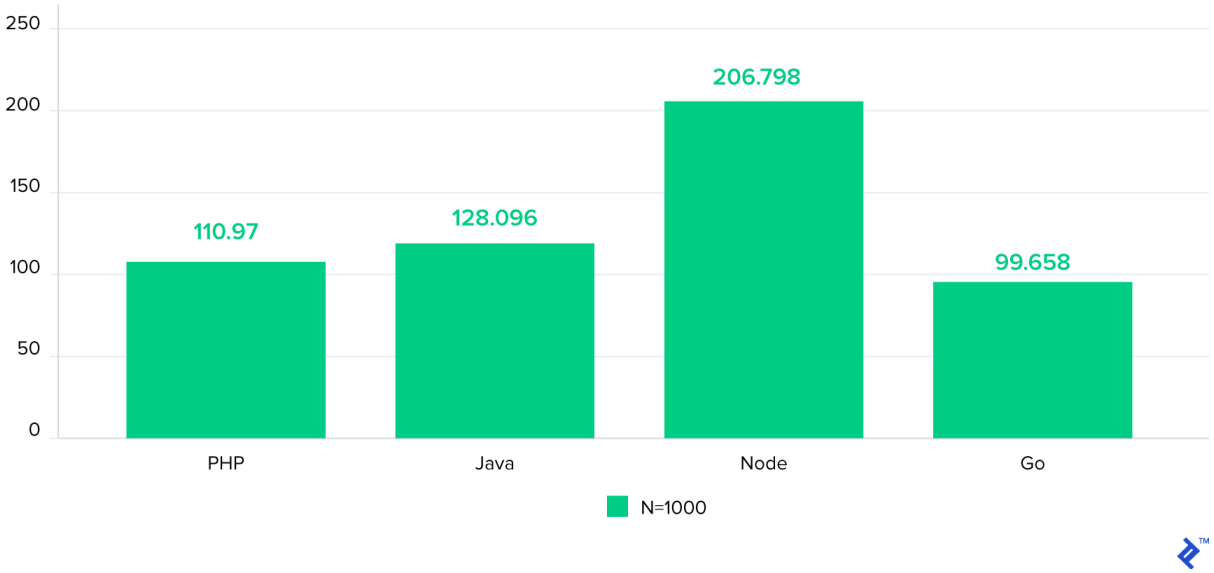


Figure 13 - A speed comparison of popular web server languages (Peabody, n.d.).

It's clear from this benchmark test that in this CPU-intensive operation, NodeJS is outperformed by other popular server-side programming languages. However, as with React, the main motivation for my using NodeJS and Express is my familiarity with JavaScript. Using the same language for the frontend and backend of the web application is extremely convenient. Furthermore, I don't expect CPU-intensive operations to be a significant factor when building this application, given that the most demanding tasks will be delegated to the cloud services, while the NodeJS server exists only as a means of sending data between the user, the Raspberry Pi, and the hosted computer vision model.

Prior to selecting the Raspberry Pi 4 as the single-board computer of choice for the system, two other possibilities were also considered; the NVidia Jetson Nano, and the Arduino Uno.

One important difference to note between the Raspberry Pi, the NVidia Jetson and the Arduino is that the Raspberry Pi and the NVidia Jetson are both single-board computers, while the Arduino is a microcontroller. The Arduino possesses a significantly slower processor, requires far less power, and does not run an operating system. The Jetson Nano and Raspberry Pi are in themselves fully-fledged computers, while the Arduino is designed to be programmed by an external computer running its IDE. Realising this, any advantages the Arduino might have over these two competitors is of no consequence, as it seems unsuitable for the purpose of running a web server or acting as a video surveillance system. (L. Pounder, 2020)

The Raspberry Pi 4B and NVidia Jetson Nano, on the other hand, carry impressive hardware for their small size. The former possesses a quad-core CPU running at 1.5 GHz and up to 8GB of DDR4 RAM, and 40 GPIO pins, while the latter sports a similar quad-core CPU running at 1.43 GHz, up to 4GB DDR4 RAM, and the same number of GPIO pins. One clear advantage the Jetson Nano holds over the Raspberry Pi is its graphics processing capabilities. It possesses an embedded NVidia GPU, unlike the

Raspberry Pi, which relies solely on its CPU for graphics processing. In the below benchmark comparison of the two by Dumi Loghin on Medium, the Raspberry Pi and Jetson Nano were first tested using CPU only to perform inference with TensorFlow Lite's benchmarking tool, using the COCO SSD MobileNet v1 single-shot object detection model. The Raspberry Pi and Jetson Nano ran inference at a somewhat similar speed, 3.38 and 2.55 seconds, respectively. However, when using its GPU, the Jetson Nano needed only 82/7ms per image, on average. (Tan, C. 2022)

```
# On RP4
$ perf stat -e armv7_cortex_a15/mem_access/
./linux_arm_benchmark_model
--graph=ssd_mobilenet_v1_1_metadata_1.tflite --num_threads=4
...
7,062,668,270      armv7_cortex_a15/mem_access/u

3.385302077 seconds time elapsed

# On Nano
$ perf stat -e armv8_pmuv3/mem_access/
./linux_aarch64_benchmark_model
--graph=ssd_mobilenet_v1_1_metadata_1.tflite --num_threads=4
...
3,430,775,018      armv8_pmuv3/mem_access/

2.551278354 seconds time elapsed
```

Figure 14 - A comparison of processing time on the Raspberry Pi 4 and Jetson Nano. (Dloghin, C. 2022)

Ultimately, however, while this difference is significant, and testament to the graphics processing power of the Jetson Nano, I realised that the main factor in favour of the Raspberry Pi 4B was its price. The Jetson Nano 4GB single-board computer alone retails at €295 on Amazon, while I was able to purchase a second-hand Raspberry Pi 4B 4GB, along with a case, and 32GB SD card for only €120. The Raspberry Pi was also chosen over the Jetson Nano due to the availability of Infrastructure as a Service (IaaS) cloud services offering low-cost access to computing resources with powerful GPUs, which this system will use to augment the capabilities of the Raspberry Pi and allow it to perform fast inference on images despite its hardware limitations.

4.2.3 Design Patterns

While Express is unopinionated in the architecture the developer is expected to follow, the Model View Controller (MVC) architecture is the most popular of all software patterns. In this pattern, the structure of any system is divided into the Model, the View, and the Controller, where each layer is responsible for a different aspect of the overall system.

The model and the database are intertwined. We use the model to replicate the structure of our database. For example, a customer table in our database might be represented in the model as a customer class, with the attributes of the customer class matching the columns used in the customer table. Any operation performed on the customer table is then reflected in the customer class through its methods, such performing create, read, update, or delete (CRUD) operations. The model is the lowest level of the MVC architecture in that it responds to requests from the controller layer.

The model does not communicate directly with the view layer. Instead, a request is made from the controller to perform some operation on the database, and this is handled by the model, which then sends its response back to the controller.

The view layer is simply the user interface. In terms of a web application, the view is the HTML, CSS, and JavaScript. Data is first received by the model and passed via the controller to the view. For example, a user might upload some data to a form in the view with the aim of adding a new row to the database. The form data is passed from the view to the controller, with then calls the corresponding class/method in the model layer. In the same way as the model does not communicate directly with the view, the view does not communicate directly with the model.

The controller, then, is the layer acting as the middleman between the other two layers. The purpose of the controller is to receive data or some instruction from the view and tell the model what needs to be done. Once the controller receives a response from the model, it then sends it back to the view for visual representation to the user.

The architecture overall is a means of applying the principle of 'separation of concerns', whereby a software system is 'decomposed into parts with as little overlap in functionality as possible.' (Makabee, H. 2022)

The advantages of using the MVC architecture is seen in that the application is easier to read and understand, the user interface and business logic are separated, the code is easier to maintain, and each component in the architecture can be individually tested and maintained.

4.3 Program Design – Version 2

As this chapter is written in the final week of implementation, the program design of the project has changed dramatically. At the project's inception, it was anticipated that cloud services would form the backbone of the object detection system, and the MotionEyeOS operating system would be used on the Raspberry Pi for detecting motion in order to forward images of detected objects to the cloud service for inference. As the project's development progressed, it was realised that popular cloud-based computer vision services such as Amazon Rekognition are prohibitively expensive for a college project. After simply experimenting with AWS Rekognition for a week, costs of €40 were incurred, despite having only created a cloud-based model for testing and having never run inference using the model.

MotionEyeOS was also quickly determined to be unsuitable for the application, upon realising the difficulty of decoupling the MotionEyeOS user interface from its supporting object detection software. Considering that RAID required a completely custom-built user interface and set of features, MotionEyeOS proved far too restrictive an option for performing the initial motion detection.

Since then, as the development progressed, the architecture used by the Raspberry Pi has developed into a custom-trained YOLOv4-Tiny model supported by OpenCV4NodeJS, running on top of a SocketIO client. The REST server functionality has been decoupled from the Raspberry Pi itself, and is now a separate 'middleman' server, which acts as the go-between for the React client and the

Raspberry Pi. A comprehensive computer vision model training flow has been implemented based on Roboflow, a cloud service designed to support object detection and image classification models.

In summary, while the technologies underpinning the project, React, Vite, TailwindCSS, Express, and NodeJS have remained the same, several core technologies have been added, the functionality of the Raspberry Pi has been completely decoupled from the REST services, authentication functionality, and communication with the client, and the cloud services originally expected to underpin the most fundamental aspect of the project, object detection, have been abandoned in favour of Roboflow.

4.3.1 Program Design

Most of the core technologies described in version one of the program design have remained the same. React, ViteJS, and TailwindCSS are still used on the client-side, and the server is still underpinned by Express and NodeJS. However, it is necessary to provide a high-level explanation of the existing technologies, as well as those which have been added since the project's initial conception.

4.3.1.1 Technologies

4.3.2 React

ReactJS is a JavaScript library for developing user interfaces, developed by Facebook. React has been the most popular frontend UI framework for several years. React, conceptually, is built on components. Every React component is built on at least one component, which then has 'child' components, forming a 'tree' of components. We could generate a UI just the same using vanilla JavaScript but React makes the task less arduous and more manageable. React's component-oriented design means that we can use components for creating reusable pieces of code and combining them together to form our entire user interface.

4.3.2.1 The DOM & The Virtual DOM

When we interact with a webpage using JavaScript, the interaction takes place via the Document Object Model (DOM). The DOM represents a web-page's hierarchy in terms of a tree-like structure of objects. JavaScript can interact with a webpage via the DOM to add, modify, and remove elements from the webpage, in response to user events such as clicks or mouse events. In plain JavaScript, we can technically do anything React can do, only it is far more arduous for the developer to write this code. React simplifies this using a 'virtual DOM'. While using vanilla JavaScript, changes to the 'real' DOM would result in re-rendering the entire DOM tree, whereas React generates a new virtual DOM whenever changes are detected, and compares it with the previously saved virtual DOM to determine the fastest and most efficient way to apply its updates. (Koffer, P. 2022)

When we perform some manipulation on the DOM, the UI changes, and the DOM is rebuilt to render our change. For example, in the following code, a simple unordered list is rendered with three child list items. JavaScript is used to target the first list item and change its inner HTML when the button is clicked. In vanilla JavaScript, this change would cause the entire webpage to re-render, despite us

having changed only a single list item. In a complex web application, rebuilding the entire DOM structure after every change would severely degrade performance.

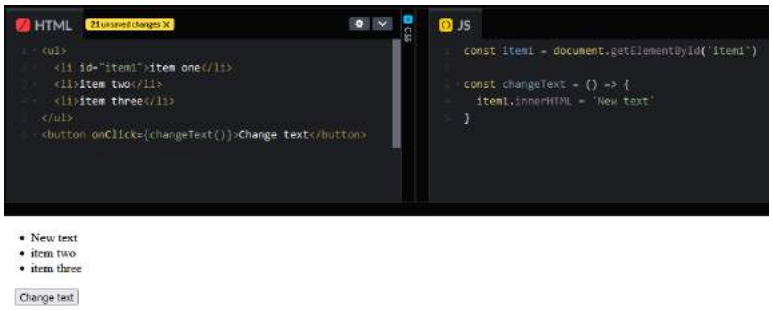


Figure 15 Using vanilla JavaScript to update a list item.

To solve this problem, React stores two copies of the DOM in memory. One is a snapshot of how the DOM structure looked before any changes were applied, and another is the new updated representation. React will the two determine what needs to change, updating the virtual DOM accordingly. No changes are made to the real DOM during this process, which is very efficient. (White, T. 2021)

When the necessary changes have been determined, the real DOM is finally updated and only the portions of the UI which have changed are rebuilt and re-rendered. This difference can be illustrated in another example. Here we use React to create the same page shown above in vanilla JavaScript.

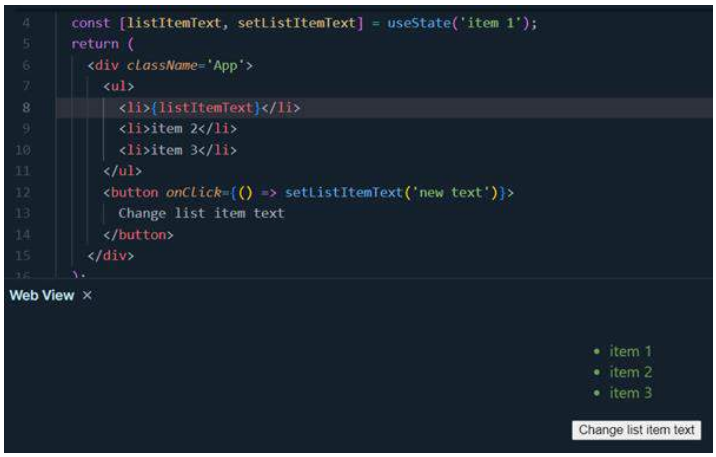


Figure 16 - Using React to update a list item.

The value of a list item is stored in state. When the user presses the button, the state is updated, and the new text is rendered on-screen. In the first example, the DOM is manipulated directly by vanilla JavaScript, and the entire DOM structure is re-rendered when a single list item changes. In this example, React will use its 'diffing' process to compare an original copy of the DOM with another copy where our updates have been applied, allowing it to determine the most efficient way to update the UI.

4.3.2.2 JSX

In vanilla JavaScript, we could create a new DOM element in the following way:

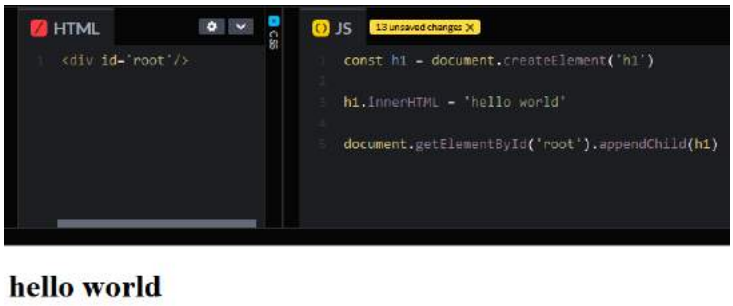


Figure 17 - Using vanilla JavaScript to render a H1 tag on-screen.

We first use the createElement function to generate a new <h1> tag, then set its innerHTML attribute to give it a text value, and finally use appendChild to append the H1 as a child element of the tag in the HTML.

While React's structure is based on components, we can use React's createElement function in a similar way to the DOM's createElement. React.createElement allows us to create a new HTML element, which can then be added to a React component to be rendered to the DOM. The React equivalent of the above example using React.createElement is far less verbose.

Figure 18 - Using React.createElement to render a H1.

However, React provides a convenient extension on JavaScript known as JSX (JavaScript Syntax Extension/JavaScript XML), which is syntactic sugar for simplifying the use of React.createElement. Where React.createElement appears more concise than using the native DOM createElement method, the same could be achieved in a single line using JSX:

```
export function App() {
  return <h1>Hello world</h1>
}
```

Figure 19 - Creating the same H1 element as above, using JSX.

JSX allows us to seamlessly integrate our JavaScript methods with our HTML. It benefits the developer by makes React code more readable and concise and is readily understandable by anyone already familiar with HTML, CSS, and JavaScript. Creating elements using React.createElement would quickly become unwieldy in a complex UI structure, and JSX makes it easier for the developer to structure their code and create reusable UI components. (Burnett, M., & Jelisejevs, P., 2020)

4.3.2.3 State

One of the most fundamental concepts in React is state. State is simply the data we need to manage over time in our application, and any application built with plain JavaScript uses state too. If we imagine a simple counting application, our UI could consist of a number and two buttons, one to increment the count, and the other to decrement the count. In this case, the state in this application is the counter number. The user updates the state value using the buttons, and we render the result of the state change on-screen.

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Counter App</title>
    <meta charset="UTF-8" />
  </head>

  <body>
    <div>
      <button id="increment">+ 1</button>
      <span id="count">0</span>
      <button id="decrement">- 1</button>
    </div>

    <script>
      const increment = document.querySelector("#increment");
      const count = document.querySelector("#count");
      const decrement = document.querySelector("#decrement");

      increment.addEventListener("click", () => {
        count.innerText = Number(count.innerText) + 1;
      });

      decrement.addEventListener("click", () => {
        count.innerText = Number(count.innerText) - 1;
      });
    </script>
  </body>
</html>
```

```jsx
export default function Counter() {
  const [count, setCount] = useState(0);

  function incrementCount() {
    setCount(count + 1);
  }

  function decrementCount() {
    setCount(count - 1);
  }

  return (
    <div>
      <button onClick={incrementCount}>+ 1</button>
      <span>{count}</span>
      <button onClick={decrementCount}>- 1</button>
    </div>
  );
}
```

Figure 20 - A simple counter app in vanilla JavaScript (L) and the same app in React (R)

In the first example we can see the HTML stucture and JavaScript logic needed to implement this. There are several unintuitive and verbose steps required to update our counter. In React, all this boilerplate is abstracted away from us as the developer. In vanilla JavaScript, we need to manually acquire a reference to the DOM element containing our 'counter' state value to update it, while React allows us to simply store our state value in a variable which can be updated and immediately changed within the UI.

Our state variable can be declared in React using the 'useState' hook, which is similar to an ordinary JavaScript function. We call the useState hook at the top of a React component, providing it with our state value. When we create the same counter app as a React component, React removes the need for HTML elements to have unique IDs, manually selecting elements from the DOM, and adding event listeners for receiving user input. The React counter's use of the useState hook makes the management of our counter state far more concise and manageable. React makes it simpler to manage the state variables affecting what the user sees on-screen, and abstracting away all the boilerplate code required by a vanilla JavaScript app makes the app more robust and less error-prone, as it is far easier for the developer to debug.

4.3.3 Vite

4.3.3.1 What is a bundler?

If we were to create a web application in the 'traditional' sense, our app would consist of HTML, CSS, and JavaScript files, with JavaScript files imported using <script> tags. As the application grows in scale, understanding code written this way can become unwieldy. Modules were introduced to JavaScript as a means of organising large bodies of code, splitting them into modular elements,

importing and exporting files from one another as needed. Two systems used to support these modules are CommonJS and ES modules. Of the two, ES modules is the newer system, and supports more efficient tree-shaking and faster loading times.

Regardless of the module system used, however, importing all of these modules separately using <script> tags would be a very cumbersome task for the developer. This would be even further exacerbated when third-party libraries with their own modules are installed, all with their own dependencies. A bundler's purpose is to package our code up cleanly, generating a dependency graph to manage the order of dependencies between our JavaScript packages and modules, and ensuring everything is installed in the correct order. The bundler will minify our code, shorten variables, remove whitespace, perform 'tree-shaking' to remove unnecessary dependencies, and finally bundle up our files into a few files, or even a single file. (Dechalert, A. 2021)

4.3.3.2 Why Vite?

In terms of which bundler system to use, CLI tools for scaffolding web applications come with default bundlers installed. Create React App (CRA) and Vue CLI both ship with Webpack as their bundler. ViteJS was chosen as an alternative. Vite is a framework-agnostic build tools, providing a development server and a build command using esbuild as its bundler. Vite describes itself as "next-generation frontend tooling".

Webpack is based around the CommonJS module system, while Vite is designed to leverage the efficiency of ES modules. Vite's use of ES modules makes it *significantly* faster than Webpack, or other alternative bundlers. Another speed advantage is afforded to Vite by its use of esbuild, written in Go, for pre-building the dependencies during development. In the below example, esbuild was compared against other bundlers in creating a production bundle of 10 copies of the Three.js library. The advantage of esbuild is clear. (Findlay, T., 2022)

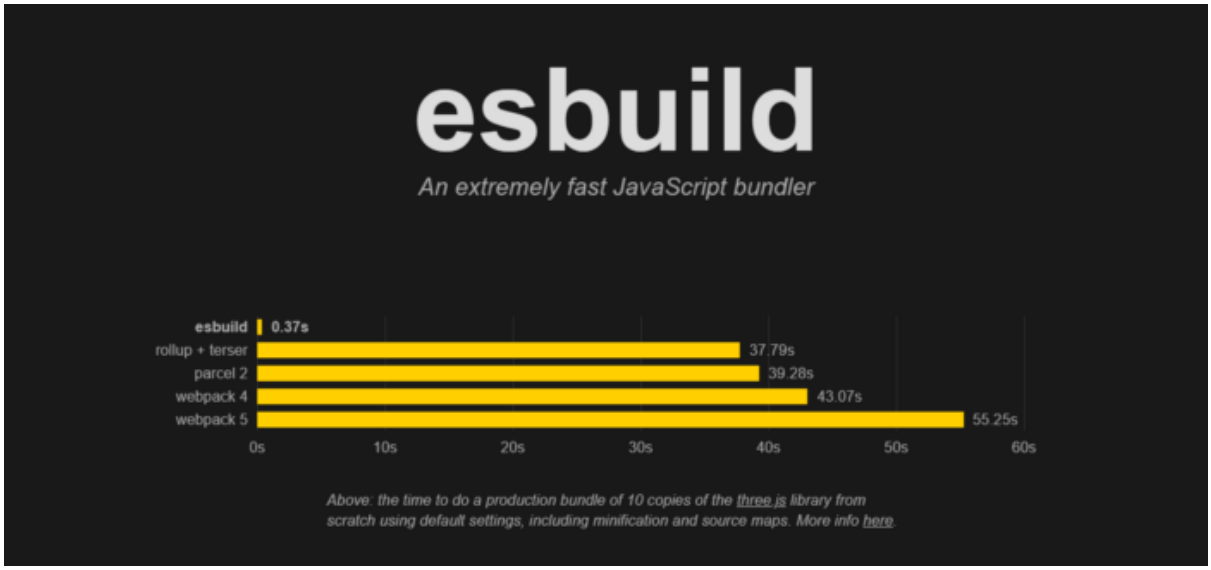


Figure 21 - esbuild speed comparison. Source: Findlay, T. 2022.

Another reason Vite was chosen was the availability of start templates. In RAID's case, an ideal starter template from the 'Awesome-Vite' GitHub repository was used, which allowed for the start application to be immediately bootstrapped with React, Vite, and TailwindCSS.

4.3.4 TailwindCSS

TailwindCSS is built on the concept of atomic CSS. With atomic CSS, as described by CSS tricks, the approach favours "small, single-purpose classes with names based on visual function". A developer could choose to write their own library of CSS utility classes in this format without using a library like Tailwind by writing simple CSS classes designed to serve a singular purpose. For example, a developer might write a series of CSS classes dedicated to controlling text attributes, such as 'text-underline', 'font-weight-400', 'color-red'. While certainly possible, however, the task of writing an entire library's worth of these atomic CSS classes would be incredibly time-consuming. Furthermore, sensible design choices need to be made, the CSS classes should support responsive design, and many domains need to be covered by the classes, such as text utilities, colour palettes, positioning, flexbox and CSS grid utilities. (Mopkar, K. 2021)

TailwindCSS is described as a "utility-first CSS framework". Tailwind provides hundreds of CSS classes for sizing, colours, typography, shadows and more, and simplifies making consistent choices with regard to UI design. Tailwind also uses a Just-in-Time (JIT) mode for purging any unused styles when building for production, helping to make the build as small and as fast as possible. Tailwind's website states that most builds use less than ten kilobytes of CSS. Note that TailwindCSS is a utility-first CSS framework, as opposed to other frameworks such as Bootstrap or Material UI, which are component-based frameworks. Tailwind does not provide pre-built components, only CSS classes.

4.3.4.1 Why Tailwind?

It could be argued that using a component-based framework like Bootstrap would speed up the UI development process, providing the developer with more time to focus on the application's business logic. However, in RAID's case, a complex user interface is not an anticipated requirement. The application's UI will be minimal, in that the main focus of the application is its ability to detect and differentiate between individual animals, a process which will be handled entirely by the server and the Raspberry Pi. The React client simply acts as a means of authorising users, viewing the camera feed, uploading and labelling images, and initiating model training. As such, a component-based CSS framework would simply be unnecessary, as only a relatively small number of components are required to build the RAID UI.

4.3.5 MongoDB

MongoDB is a relatively minor part of the application, but a simple database was necessary for storing user information in order to register and authenticate users. Furthermore, data related to user pet models and saved images needs to be maintained, and MongoDB was a perfect candidate considering the small amount of data being stored. Traditional SQL databases such as MySQL and Microsoft SQL server are known as relational databases, and use a model whereby data is stored in table-like structures, and primary and secondary keys are used for forming relationships between tables. MongoDB is a NoSQL database, meaning it is non-relational, and instead uses an entirely different structure for storing and retrieving data, known as BSON, which is similar in structure to JSON.

MongoDB is designed for scalability using a concept known as 'sharding', meaning data is partitioned across multiple physical servers. By creating multiple copies of the same data across different servers, MongoDB provides redundancy against hardware failures, and its data distribution allows for high availability. MongoDB is a suitable solution for distributed data, where it's necessary to have multiple copies of the same data across a range of servers to mitigate against hardware failure. It is also suitable for very large amounts of data, considering its built-in sharding solution for distributing data in chunks across servers. Finally, since relational databases are based on pre-defined schema, adding new columns in a relational database is difficult by comparison to the schema-less MongoDB.

Despite all the advantages offered by MongoDB over traditional relational databases, MongoDB was primarily chosen as the the database solution for RAID due to its simplicity and familiarity. MongoDB integrates easily with NodeJS using the Mongoose NodeJS driver, and considering how relatively unimportant the database is with regard to the rest of the project, MongoDB was an ideal solution, considering that I, as the developer, am already familiar with and comfortable using MongoDB. In a development project with so many unknowns, in an area largely unfamiliar to me, it is only strategic to use something already familiar to ease the burden of research in developing the project.

4.3.6 Express and NodeJS

4.3.6.1 What is NodeJS?

When we write JavaScript code for the computer, the browser's JavaScript engine needs to convert our code into machine code for interpretation. All the different browsers available use different JavaScript engines. The most popular today is Google Chrome's V8 engine, which is a very fast engine. Because of the V8 engine's speed, developers began to consider the possibility of using JavaScript outside the browser, and so NodeJS was developed as a JavaScript runtime environment for running on a local machine, completely independent from the browser. Since NodeJS has a HTTP module, it is possible to build entire servers using NodeJS. This means that JavaScript can be used on both the client-side, and the server-side. However, NodeJS is a runtime environment, not a framework, which is where ExpressJS comes into play. ExpressJS is a server development framework. This makes it possible to create an entire development stack using JavaScript, in the form of the MERN stack. (Reddy, N. 2020)

The heart of the NodeJS architecture is the event loop. Unlike other languages like Java, JavaScript is single-threaded. JavaScript cannot run tasks in parallel, only concurrently. Especially on the server, we don't want to block our only thread. Events like database calls should be delegated off 'somewhere else' to be executed.

JavaScript uses a data structure known as a 'call stack'. When we ask JavaScript to do something, one item is 'stacked' onto the other, and events are processed in that order. Within NodeJS, the callstack keeps track of the events which need to be processed, and in what order to process them. (Reddy, N. 2020)

4.3.6.2 Express

It might be possible to handle HTTP requests, serve static files, or use templating to create responses using pure NodeJS, but to remove the burden of writing all this from scratch, ExpressJS provides a large number of features for developing single-page, multi-page, and hybrid web applications. As explained, the popularity of Express for JavaScript developers is that it runs on NodeJS, allowing developers to create their entire stack using the same language. Packages exist to extent Express to handle functionality such as cookies, sessions, user authentication, and many more. Similar to the reasoning for using MongoDB, NodeJS and Express were chiefly chosen for their familiarity, especially considering the unfamiliar ground much of the computer-vision related functionality of this project lies on. In RAID, the entire middleman server is created using NodeJS and Express. (MDN Web Docs, n.d)

4.3.7 YOLOv4-Tiny

To understand YOLO, we first need to understand Convolutional Neural Networks (CNNs), and to understand CNNs, we first need to understand neural networks.

4.3.7.1 Artificial Neural Networks

The design of a neural network is conceptually similar to the organisation of neurons in the brain. The network can consist of dozens, all the way up to millions of layers. The first layer is always known as the input layer, which receives data from the outside world which the network will use to learn to recognise a subject. The final layer is then known as the output layer, which signals how the network responds to the information it has learned.

Between these two layers can be any number of hidden layers, which form the majority of the network. A neural network is known as 'fully-connected' when all layers are interconnected to either side. Most neural networks are fully-connected. The influence one layer has on the next is determined by a 'weight' value, which will be either positive or negative. A higher weight value connecting one layer to the next will result in the first layer having a higher influence on the next. This structure corresponds to the influence synapses have between neurons in the brain. In summary, a neural network operates by receiving information through its input layer, passing the information through its many hidden layers for transformation, and passing a result through the output layer.

The network can be thought of as operating in two distinct ways; training and running inference. When the network is being trained, it learns using a feedback process known as 'backpropagation'. This process seeks to reduce the difference between actual and intended output, to make the network's predictions more accurate. The difference calculated by the backpropagation can be used to adjust the weight values determining the influence each layer has on the next, simulating the way humans learn from experience. When the network has been trained with a sufficient number of examples, the goal is for it to reach the point where it can make predictions on an entirely new set of inputs which it has never seen before, and still receive an accurate response. **(**Woodford, C. 2021**)**

4.3.8 Convolutional Neural Networks

A CNN is a specialised type of neural network for performing pattern detection. A CNN is differentiated from a standard neural network by its use of hidden layers, or convolutional layers. A CNN will also have other non-convolutional layers, but the convolutional layers form the basis of the network. The job of the convolutional layer is to receive some input, perform a transformation on it, and send the transformed output to the next layer. Within a convolutional layer, this type of operation is referred to as a 'convolution'.

In any given image, there will be multiple shapes, objects, or textures. In terms of the patterns a layer could detect, one might filter for discerning object edges, others might identify simple shapes. The patterns detected by a layer are determined by its 'filters'. As the network progresses through its layers, it will become more capable of identifying specific, sophisticated objects, such as eyes, or hair, and as the network approaches the output layer, it will be able to identify entire objects, such as a face, or a dog.

A common example used for illustrating how a CNN works is recognising hand-written digits. In this example, a CNN would categorise images based on the type of number thy contain. If the network's first hidden layer were a convolutional layer, we would also need to specify the number of filters our convolutional layer should use. A filter can be represented as a small matrix containing a number of rows and columns initialised with random numbers. For example, let us imagine our convolutional layer has a single filter of size 3x3. When the convolutional layer receives an image as input, the filter matrix will slide, or 'convolve' across every 3x3 block of pixels in the input image.

We allow our 3x3 filter to convolve across every 3x3 block of pixels, calculating the dot product of the filter matrix and the 3x3 block of pixel values (which will be numbers 0 and 1).

In practice, if our filter were a 3x3 matrix of the following numbers:

0.979	0.278	0.940
0.713	0.048	0.564
0.604	0.327	0.853

And our filter convolved over a 3x3 pixel grid containing these values:

0.5	0.9	0.9
1.0	1.0	1.0
0.8	0.8	0.8

The dot product calculation would multiply each row in matrix one with each column of matrix two, returning the product of these values. In this case, the calculation would take place as follows:

(0.979 x 0.5) + (0.278 x 1.0) + (0.940 x 0.8) = 1.317

(0.713 x 0.9) + (0.048 x 1.0) + (0.564 x 0.8) = 1.096

(0.604 x 0.9) + (0.327 x 1.0) + (0.853 x 0.8) = 1.135

Resulting in a dot product of [1.317, 1.096, 1.135]

Running this operation on every 3x3 grid of pixels, we will create an entirely new representation of our original grid of pixel values, where each pixel now contains the dot matrix value of the previous representation's 3x3 pixel blocks multiplied by our filter. This new set of values is then passed to the next convolutional layer, and the process takes place again with a new set of filters. This process is visualised below:

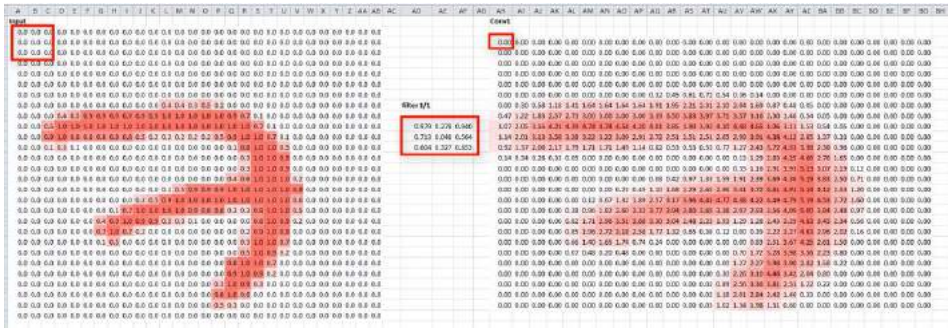


Figure 22 - Convolving a filter matrix over our image. (deeplizard, 2017)

The image on the left is our original hand-written digit, where each pixel value is represented as a value between 0 and 1, where 0 is ‘dark’ and 1 is ‘bright’. Our filter is shown in the centre, and we can imagine the filter ‘sliding’ across every 3x3 grid of pixels in the original image, calculating the dot product value, and assigning it to a single pixel in the output layer. We can't observe any specific pattern found using this filter value, but imagining we have four separate filters for our convolutional layer instead of one, we can represent the following filter values visually like this:

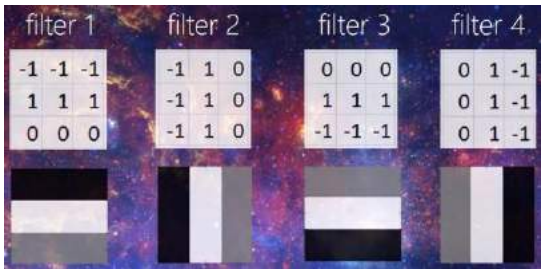


Figure 23 - Filter values represented visually. (deeplizard, 2017)

Where -1 corresponds to black, 1 corresponds to white, and 0 corresponds to grey. If our original image of the hand-written digit were convolved with each of these four filters individually, the following output would be produced:

Figure 24 - Edges detected by the filter. (deeplizard, 2017)

Our filters, in this case, have detected edges. This simple result would be seen at early stages in our network, but as we progress deeper into a CNN's layers, more sophisticated shapes can be detected, such as entire dog faces:

Figure 25 - An example of more advanced detections made at a later stage of the network. (deeplizard, 2017)

4.3.9 YOLO

Having discussed artificial neural networks in general, followed by convolutional neural networks, we can discuss how the YOLO algorithm works. An overview of CNNs was necessary first, as YOLO uses a CNN as its backbone. YOLO is an acronym for "You Only Look Once", which is indicative of its approach as a single-shot detector algorithm, capable of very quickly performing object classification, localisation, and detection on an image.

Classification is simply determining what kind of object is the subject of an image. Localisation is determining where exactly in the image the object is. Object detection then is the ability to perform classification and localisation on multiple objects within the same image.

We've discussed how a CNN can be used to classify the subject of an image, as demonstrated in our hand-written digit recognition example. The YOLO algorithm expands on this concept to not only identify what kind of object is in an image, but where in the image it is located.

For example, if we had trained a CNN to recognise cows, and provided it with an image containing four different cows, a successful detection would classify the entire image as containing a cow. The CNN is not capable of detecting where in the image the cow is and cannot distinguish one cow's position from another's.

YOLO works by first passing an image through a CNN, and then diving the image into a grid, where each quadrant in the grid makes a prediction as to whether it contains the object or not. From all the class predictions made by each box, YOLO chooses the highest overall class probability. (Tatan, V. 2021)

Before outputting its final prediction, YOLO will use a technique known as non-max suppression to remove redundant or overlapping bounding boxes, retaining only the most accurate one. To explain now non-max suppression (NMS) works, we can imagine we have an image with three predicted bounding boxes, each predicting the same class label of 'cat'.

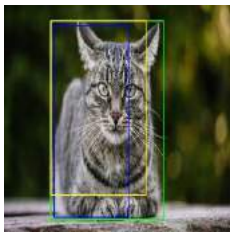


Figure 26 - Three similar bounding boxes detecting the same object. (Subramanyam, V.S. 2021)

YOLO's predictions will be in the format [x1, y1, x2, y2, class, confidence]. At the first stage, NMS will sort these three bounding boxes in descending order according to their prediction's confidence value. After this sort, any bounding box with a confidence value below a pre-defined minimum confidence threshold will be discarded. For example, if we imagine we had a confidence threshold of 0.8, and our sorted confidence values were [0.9, 0.85, 0.75], the last bounding box would be removed, as its confidence score is below the minimum threshold.

NMS will then perform an intersection over union (IOU) comparison of the bounding boxes to determine the degree of overlap between them. It is safe to assume that if two bounding boxes making the same class prediction have significant overlap with one another, they are making a prediction on the same object. The IOU calculation will first find the area of overlap between two bounding boxes, then the combined area of the two. The overlap is then divided by the area of the union, returning a value between zero and one, where a value closer to one means a more similar set of bounding boxes. Starting from the bounding box with the highest confidence value, we move down the list of bounding boxes, repeating the process of IOU calculation until only a single box remains.

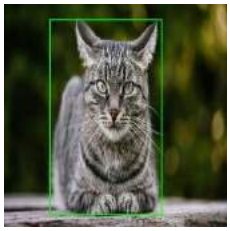


Figure 27 - The result of NMS. A single bounding box remains. (Subramanyam, V.S. 2021)

We've now covered a high-level overview of the YOLO algorithm, artificial neural networks, and convolutional neural networks. The YOLOv4-Tiny algorithm, then, is simply a smaller architecture of

YOLOv4, the fourth iteration of the original YOLO algorithm. YOLOv4 is the same algorithm in principle, but with improvements made to its speed and detection accuracy. The 'tiny' variation of YOLO was chosen due to its smaller size and relatively higher speed. YOLOv4-Tiny is suited for edge devices, making it an ideal choice for the Raspberry Pi used by this application.

4.3.10 MQTT

The MQTT protocol is a relatively minor part of RAID. It is used to allow for communication between the client and the Raspberry Pi, despite the two having no knowledge of one another. Despite its minor role, MQTT is an interesting protocol which proved invaluable, as no other straightforward solution seems to be available for facilitating communication between a server and an 'IoT' device, which the Raspberry Pi could be considered to be, in RAID's case.

MQTT (MQ Telemetry Transport) is based on a publish/subscribe protocol designed for facilitating simple communication between devices, designed for use in environments where devices have poor hardware resources, or where little network bandwidth is available.

The basic concepts necessary to understand MQTT are its publish and subscribe system, messages, and the MQTT broker. When we refer to MQTT's publish/subscribe model, we mean that any device can publish or subscribe to MQTT 'topics'. In our case, the RAID middleman server publishes to the same MQTT topic as the Raspberry Pi is subscribed to. This means that 'messages' can be passed between the two MQTT clients. Any kind of data can be passed between the two. A client's interest in a type of incoming message is determined by 'topics'. A 'publishing' client's message is also sent to a specific topic. In RAID's case, we use an 'api/buzz' topic. We pass a 'true' value to this topic from the middleman server, notifying the Raspberry Pi server that we want it to trigger its buzzer.

In MQTT, the 'broker' is responsible for receiving mesages, filtering the message by topic, and publishing the message to all clients subscribed to that topic. Many MQTT clients are available, but in RAID's case, we use HiveMQ, a free cloud MQTT service which allows for up to one hundred MQTT client connections. (Santos, R. 2017)

The use of the HiveMQ server, and MQTT's publish/subscribe model whereby any connected device is regarded as a client facilitates communication between the React client and the Raspberry Pi via the middleman server. The client can send a request to a REST endpoint on the middleman server, which in turn publishes a message to the 'buzz' topic on HiveMQ. Meanwhile, the Raspberry Pi is subscribed to this topic, and so receives the message notifying it to trigger its buzzer, despite the middleman server and Raspberry Pi having no direct knowledge of one another.

4.3.11 Application architecture

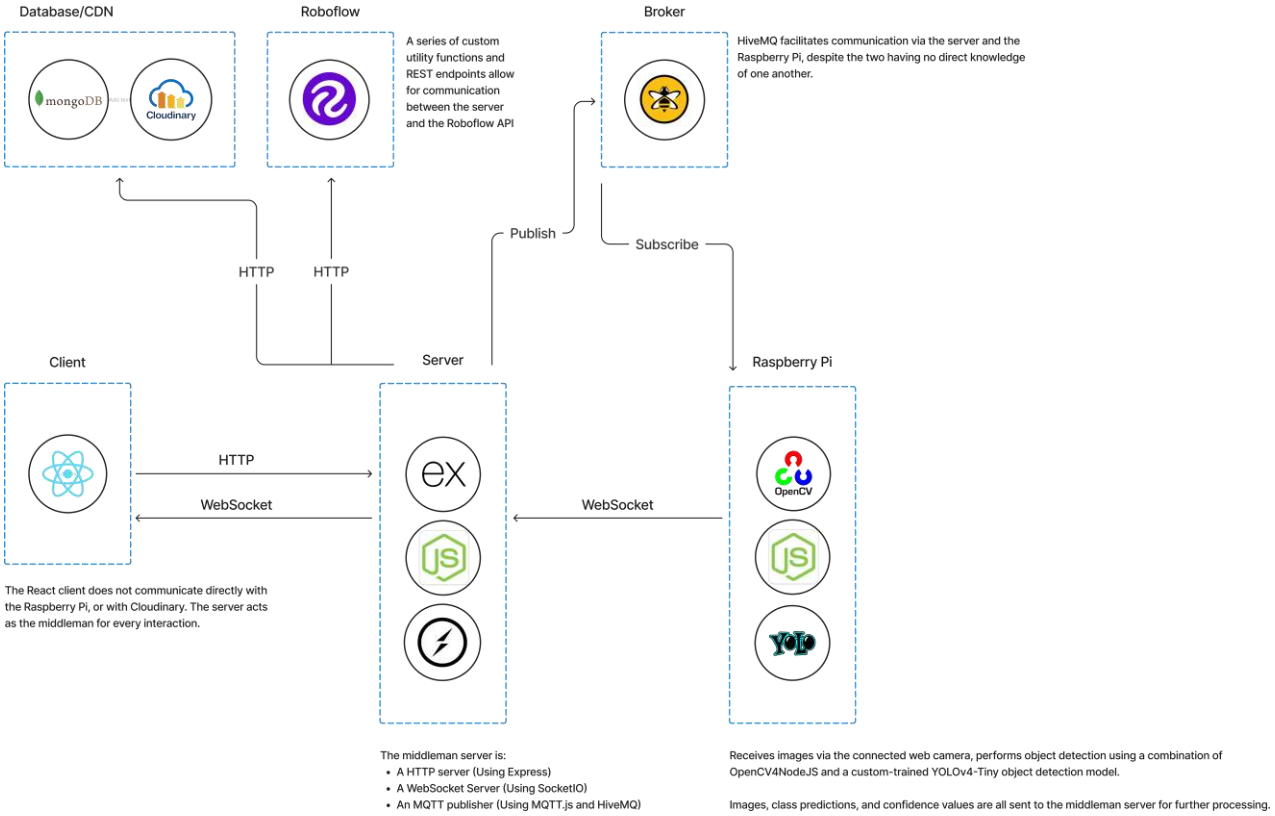


Figure 28 - A block diagram of RAID, demonstrating how each of the three distinct aspects and the cloud services are interconnected.

The architecture of this application generally follows a 3-tier MVC model, where the Client is represented by the React application, the Express server acts as our Controller, and MongoDB and Cloudinary both form our model layer. The design mainly aligns with the standard MongoDB, Express, React, NodeJS (MERN) stack architecture, with some differences. The React client (View layer) is unchanged, and consists of a React application written in JavaScript, styled using TailwindCSS, and is supported in its core labelling functionality by a custom fork of React-BBox-Annotator combined with React-Dropzone-Uploader.

The business logic (controller layer) of the application can be said to consist of two distinct but interdependent parts, the Express HTTP server and SocketIO server running on NodeJS, and the Raspberry Pi running OpenCV4NodeJS and the custom YOLOv4-Tiny model. The model layer contains both the Cloudinary image directory structure, and the MongoDB user database. Finally, the HiveMQ server does not fit neatly into the MVC paradigm, as its purpose is for facilitating the exchange of information between the Express server and the Raspberry Pi, both parts of the controller layer.

Though Roboflow is an external service and not part of the RAID application, RAID depends heavily on Roboflow to facilitate the training of custom pet detection models. A large number of utility functions and REST endpoints exist within the Express server for facilitating interaction with the Roboflow API, specifically creating user projects, uploading images and annotations, generating dataset versions and specifying image pre-processing and augmentation steps, exporting models for training, and finally training the model itself. As such, Roboflow has been included in the diagram above due to its importance to RAID's core functionality, but it does not fit into the overall architecture.

The architecture diagram details multiple channels of communication being used between the client, the server, the database, and other third-party services, as well as within the server itself, between the Express server and the Raspberry Pi. On the client-side, a SocketIO client receives base64-encoded images from the server, and they are displayed on-screen in Motion JPEG (MJPEG) format, as if they were a video stream. This SocketIO client also receives notification events from the SocketIO server. When the user presses a button to capture a manual screenshot or activate the buzzer, uploads annotated images, registers and account or logs in, a request is sent to the server over HTTP, to one of the server's REST endpoints.

The React client does not communicate directly with any of the third-party services or Raspberry Pi. The server acts as the middleman, facilitating communication in any case. The NodeJS middleman is both an Express server and a SocketIO server and uses its REST endpoints to read data and make changes to the MongoDB and Cloudinary databases over HTTP, just as it does for any Roboflow-related request. Its SocketIO server receives images, object labels, and confidence values from the Raspberry Pi, and sends them to the React client, all over WebSocket. Finally, both the middleman server and the Raspberry Pi are MQTT clients, where the middleman is an MQTT publisher, and the Raspberry Pi is a subscriber. Both clients connect to the HiveMQ cloud based MQTT broker, and publish and subscribe to the same MQTT topic, allowing the two to communicate, despite having no direct knowledge of one another. In this way, the user can press the 'buzz' button on the client-side, sending a HTTP request to the middleman server, which in turn publishes a message to the MQTT broker, instructing the Raspberry Pi to activate its buzzer.

In terms of security, RAID uses the PassportJS library to handle user authentication, which supports several strategies such as local (email and password), Google OAuth 2.0, and JWT. Normally, when using Google OAuth 2.0, the user data is serialised into the session, but in RAID's case, the client is a single-page application, so it is simpler to use JWT instead. Therefore, the user data is not serialized into the session, and instead, the Google profile data is encoded into a JWT and sent to the client. The client then stores the JWT in a cookie, and subsequent requests to the server retrieve the cookie and pass it through the request headers for authentication by the Passport JWT strategy.

The local strategy is also set up to send JWTs to the client, and for consistency, the client stores the resulting token as a cookie in any case. This way, regardless of the authentication strategy used, the client stores the token in the same way, making it easier to manage and use.

4.3.12 Database design

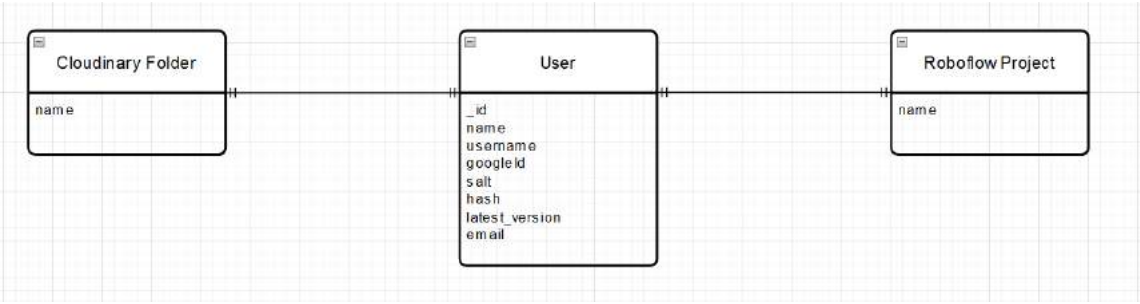


Figure 29 - Fields used by the User table, and the relationship between the MongoDB user database, Cloudinary folders, and Roboflow Projects.

As described in the application architecture section, RAID's design largely follows the MVC pattern, but with some differences. In terms of database design, the model layer of the architecture can be considered to contain two, or three different parts. While the application only uses a single true

database for storing user information, each user is linked to a unique Roboflow project and Cloudinary folder based on their MongoDB _id attribute. When a new dataset version is generated within Roboflow, the user's latest_version attribute is updated to accordingly, so a model is always trained using the most recently created dataset version. The Cloudinary folders will contain subfolders, named according to the type of animal image they contain. As shown in the diagram above, a one-to-one relationship exists between each user and their respective Cloudinary folder and Roboflow project.

In terms of the user database itself, many of the fields are optional. Depending on whether a user has registered using Google, or with email and password, the googleId, name, salt, hash, and email fields may or may not be provided. In any case, the username field will always be present. When a user logs in with Google, the username field is automatically set to match their GoogleID.

4.3.13 Process design

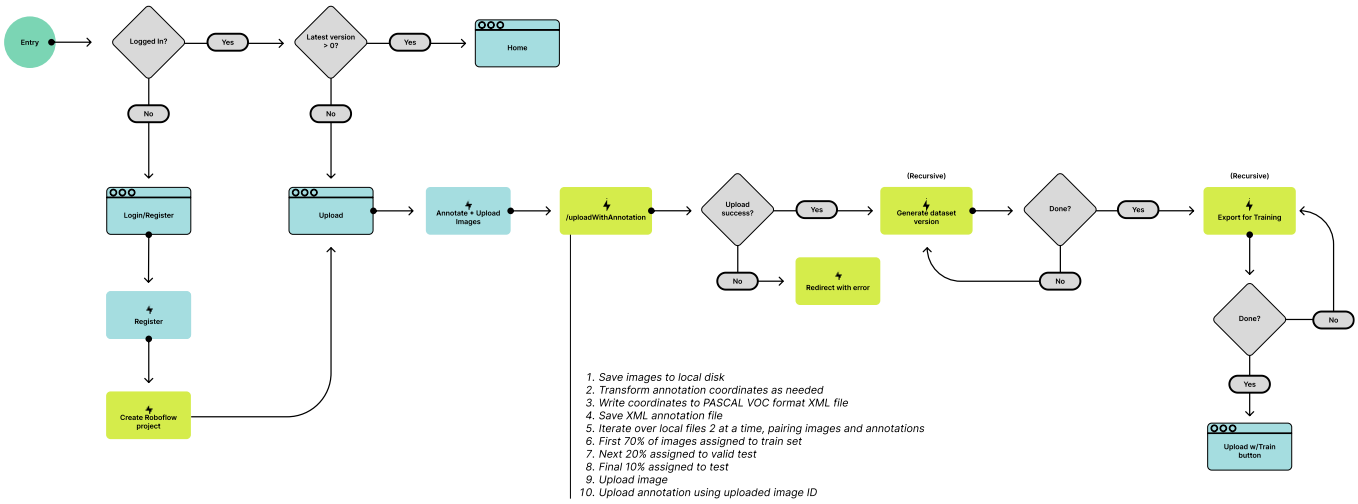


Figure 30 - The Roboflow functionality flow

The diagram above demonstrates the process design relating to the application's Roboflow functionality. When a user vists the React client in their web browser, we first check if there is an 'x-auth-token' cookie stored in the session, and if so, check if it's valid by using it to make a request to the server for user information. If a valid cookie is found, we check the user information retrieved from the server to see if the user has generated a dataset version. The default version number is 0, so we simply check if the user's latest version is greater than 0. If so, they have created a dataset version already, and are directed to the homepage. Otherwise, they have yet to create a dataset, and are directed to the upload page to label and upload a dataset.

If there is no cookie to begin with, or an invalid cookie is found, the user is not logged in, and is directed to the login/register page. Once the user registers, a Roboflow project is automatically created for them based on their unique MongoDB ID, and they are redirected to the upload page. The user uploads their images, where they are stored temporarily in state by React-Dropzone-Uploader, along with the coordinates, file metadata, and labels stored in a custom fork of React-BBox-Annotator. When the user presses 'finished', their uploaded images and corresponding annotation data is sent to the server via the 'uploadWithAnnotation' endpoint, where the files are processed by Multer. From here, the images are saved temporarily to the server's local disk. Some transformations are applied to the annotation coordinates, specifically, the 'left' and 'top' values received from React-BBox-Annotator are rename to 'xmin' and 'ymin', and the 'xmax' and 'ymax' values are calculated by adding the 'left' and 'width' values, and the 'top' and 'height' values,

respectively. These values are then converted to PASCAL VOC format in preparation for annotating their respective images and are saved to the local disk in XML format, using the xmlbuilder2 library.

The server will then iterate over the local files two at a time, first uploading an image, then uploading its corresponding annotation using the image ID generated by Roboflow after a successful image upload. As the server progresses through the image, a percentage counter is incremented. The first 70% of the files will be assigned to the 'train' split value of the dataset. The following 20% will be assigned to 'valid', and the final 10% assigned to 'test'. Providing the upload succeeds, a recursive 'dataset generation' function will be called to ask the Roboflow API to generate a dataset version based on the annotated images it just received. In this step, images will have their orientation corrected automatically by Roboflow, unnanotated images will be filtered entirely from the dataset, and images will be resized to 640x640 pixels. This function is recursive, meaning it will continually call itself until Roboflow completes the dataset version generation. When the version generation finishes, a final recursive function is called in order to prepare the data for model training. Internally, Roboflow's automatic training API uses data in YOLOv5PyTorch format, so the dataset version is exported to this format in preparation for training. When this succeeds, the user's dataset is ready for training. In the client, the upload page will change format, displaying a 'train' button to begin the training process.

With a custom model trained, the middleman server will be able to make an inference request to the user's Roboflow project's trained version and receive a resulting prediction.

4.4 User interface design

The UI design for this application sought to provide a clear frontend experience for the user, creating a layer of abstraction from the comparably complex business logic. A limited, muted colour palette, and the Inter font, a font classified as in a 'geometric non-grotesque' style, which is designed to work well at small sizes.

A design was drafted prior to the distribution of the survey with a view to providing users with a preliminary guide as to how the proposed application is intended to function. This prototype is displayed below:

After conducting the survey, many respondents provided valuable feedback regarding the UI design, which informed the design decisions made thereafter. A second version of the UI was drafted to reflect these suggestions.

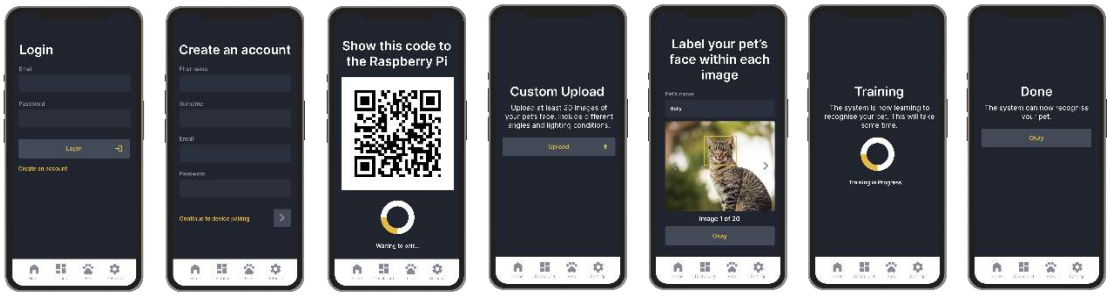
In terms of design choices, the changes suggested by survey respondents included resizing images to be as large as possible, increasing the text size, explaining clearly that facial images were required specifically in order to train the system in recognising a pet, icons to provide clarity on the purpose of certain buttons, and a light mode.

Respondents also suggested additional features, some of which have been considered in the UI design. Others, though conceptually interesting, were considered to be outside the scope of this project and were not included in the UI design. Of those included in the design were the ability to recognise multiple pets, and push notifications to inform the user when their pet, or a wild animal were detected.

Finally, users were interested in the prospect of the system automatically capturing images when an animal is detected, along with a dashboard for viewing previously captured images. Accommodations have now been made for all these features in the updated UI design.

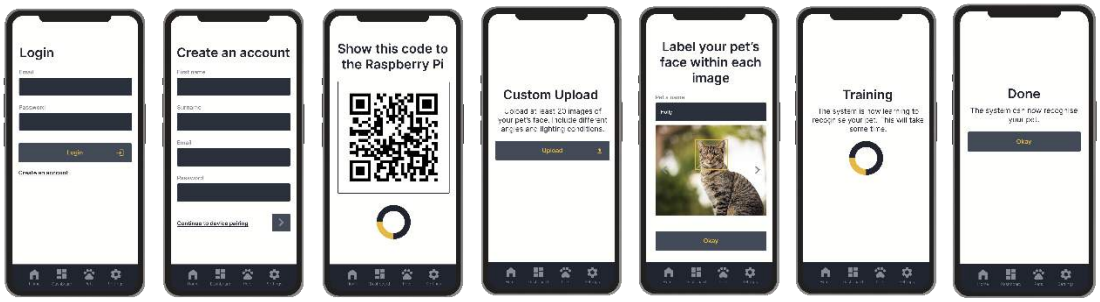
Some users also expressed interest in an attached night vision camera for the monitoring of nocturnal animals, but this feature is considered ancillary to the UI design and may be included at a later point. Further detail was also added to the design regarding the screens displayed after the user has trained a custom detection model. Clarity has now also been provided in highlighting the distinction between the registration and model training stage of the installation and registration of the application, as opposed to the main ‘usable’ stages. Overall, though a preliminary UI design had already been created, the user feedback was invaluable in informing the necessary features and design decisions for the system.

The updated UI design is included below:

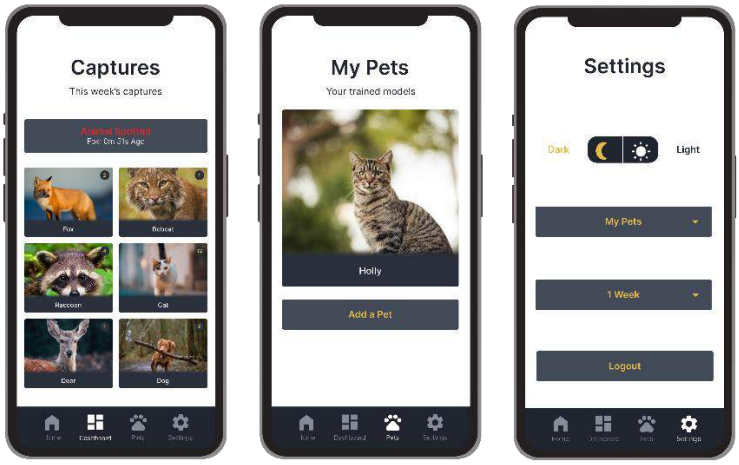


Of the considerations made, the only one which I felt could not be addressed was the request for larger images. On mobile devices, images already occupied as much space as possible, and so were left at the same dimensions as in UI version 1. Heading font sizes were increased from 24px to 32px, the standard font size increased from 12 to 14px, the H2 size increased from 14 to 18px, and the text size used in buttons increased from 12 to 15px. Another change was the addition of icons to buttons to provide further clarity as to their purpose. For example, the button used to upload pet images was given an 'upload' icon.

A light mode was also added, inverting the existing colour palette by using the original 'white' text colour as the background colour, and the 'Dark navy' background colour from the original dark mode as the main font colour. The original 'yellow' was kept as the main highlight colour.



Of the proposed features, most survey respondents felt the ability to recognise multiple pets was the most important, followed by push notifications to notify the user when a wild animal, or their own pet, was detected by the system. The second version of the UI sought to provided further clarity as to how the proposed system will work. As such, three more screens were added to illustrate the system's functionality, especially including features requested by survey respondents:



4.4.1 Wireframe

Wireframes were drawn in the first instance to get a general feel for how the UI might look, in preparation for creating a higher fidelity Figma prototype.

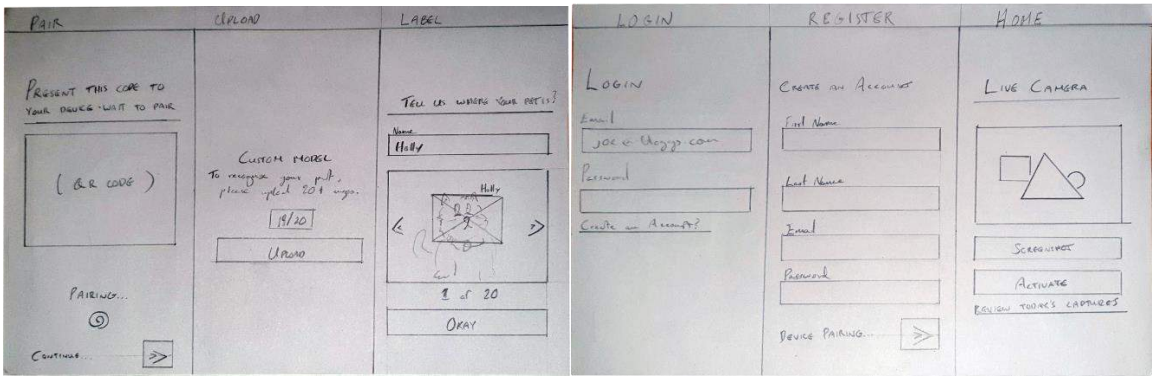
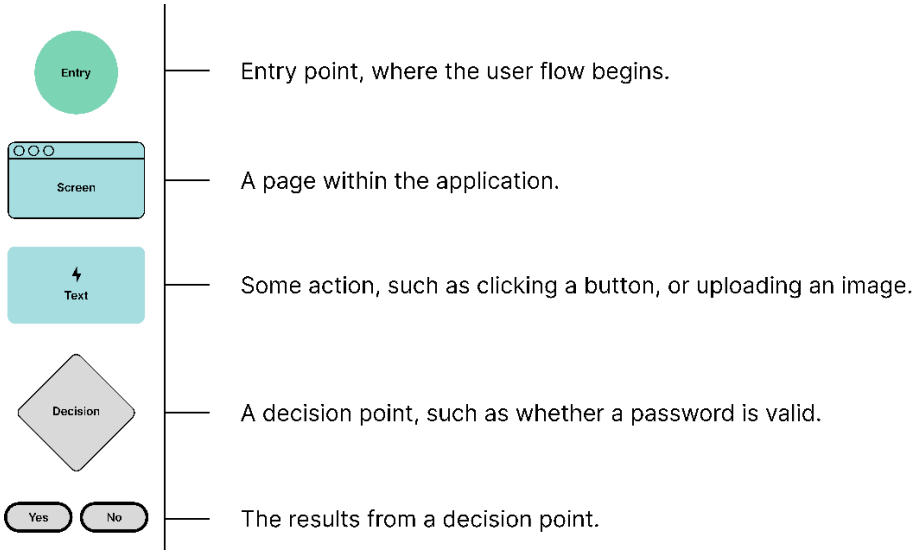


Figure 31 - Wireframes showing Login, Register, and Home (L) and Pair, Upload, and Label (R)

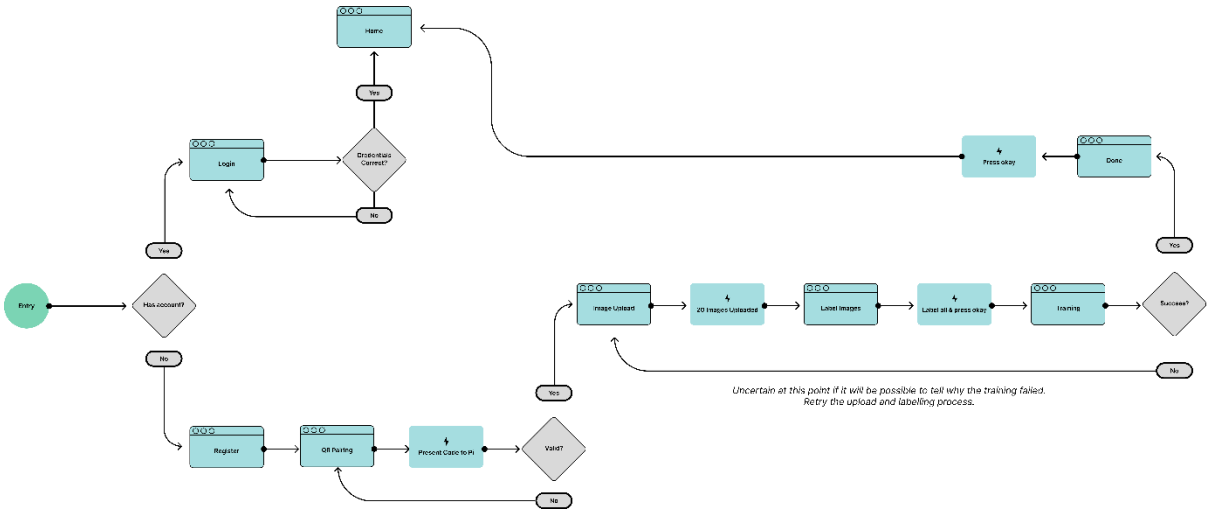
4.4.2 User Flow Diagram

The purpose of a user flow diagram is to present a clear idea of how the user will get from point A to point B within the system, beginning with an entry point and reaching some goal through pages and actions. In our case, the entry point will be either the login page, or the home page, depending on whether the user is already logged in.

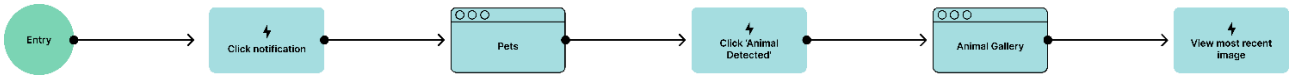
4.4.2.1 Legend



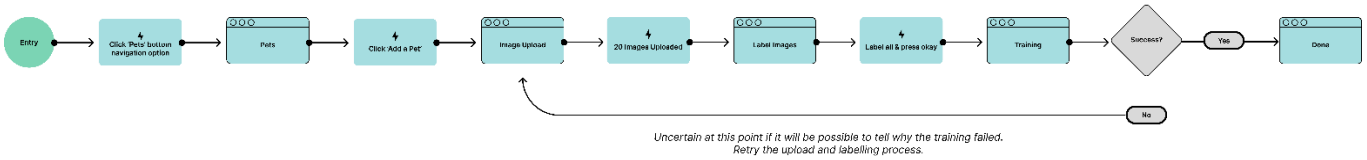
4.4.2.2 Login and Register (Includes Add Pet)



4.4.2.3 View a Detected Animal



4.4.2.4 Add a New Pet



4.4.3 Style guide

4.4.3.1 Fonts

At this point in the project's development, the application is intended to be used primarily on mobile devices for convenience, given that survey feedback suggested that receiving push notifications for

animal detections, and mobile friendliness are both very important to users, and so the Inter font by Rasmus Andersson was chosen because this font is designed to work very well at small sizes.

Inter features a tall x-height to aid in readability and 'adjusted punctuation depending on the shape of surrounding glyphs', which aids in, for example, disambiguating between '0' and 'o' characters. Rather than using multiple fonts, several font weights were used to represent the native HTML H1 to H4 tags, paragraph tag, and some additional special cases, as illustrated below.

Heading 1	Heading 2	Heading 3	Heading 4	Paragraph	Card caption	Bottom navigation item
32px	18px	16px	16px	14px	10px	10px

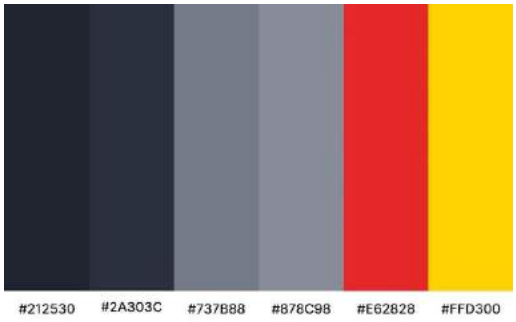
4.4.3.2 Color Palette

In general, muted colours were chosen, specifically shades of grey, with the aim of keeping the user interface as simple as possible, given that the UI is, in this case, secondary to the primary purpose of the application - animal recognition, which is handled by the server-side in combination with a series of AWS cloud services.

Despite this, we should keep in mind the initial impression the application may have on users, and in order to highlight the application's intended purpose, a dark shade of yellow known as 'cyber yellow' was chosen both as the main colour in the logo, and as the highlight colour in the UI prototype. It was chosen due to yellow's positive evocation of spontaneity, but potentially also a sense of caution.

I feel this reflects the application's overall aim as a novel concept (spontaneity), but also its purpose in deterring pests or even potentially dangerous animals from harming pets (caution). Yellow is familiar to us as a shade representing caution. Branding Compass describes yellow as a colour which 'raises the alarm and helps us to avoid danger.' Through this association, they say, yellow is seen as a bold protector, which is reflected in some other brands using yellow in their logos, portraying themselves as a brand to be taken seriously. (Branding Compass, n.d)

With this in mind, the final palette selected for the application is as follows:



RAID

Real-Time Animal Identification & Deterrence

The highlight colour, #FFD300 is known as ‘cyber yellow’, and is the main colour used in the project’s logo.

4.5 Conclusion

In summary, significant thought was put into both the program design and user interface design aspects of the project. This chapter has detailed the reasoning behind the technologies chosen or the project's development, and provided a high-level overview of how they work. The chapter has also provided an overview of the very deliberate design choices made for the RAID user interface, minimal though it may be.

5 Implementation

5.1 Introduction

In this chapter, we discuss the project's progress in terms of each sprint and explore how the project was created. For each project iteration, we'll explain the goals which were set out, how they were achieved, and any shortcomings preventing a goal's completion.

5.2 Scrum Methodology

The SCRUM methodology is a project management framework particularly popular for software development projects. In this methodology, the project workload is divided into iterative two-week blocks known as "sprints". Before the start of each sprint, a sprint planning session is held to create a backlog of tasks the developer intends to complete before the sprint's conclusion. In a similar way, each sprint is followed by a sprint review, which provides the developer with an opportunity to reflect on the successes and failures of the sprint, with a view to planning the necessary changes to improve the next time.

SCRUM is designed as a project management framework for teams, but it can also be adapted to a software project with only a single developer. In a typical team setting using the SCRUM methodology, the team would consist of a product owner, a SCRUM master, and the software development team. The product owner's role is to think in terms of the 'big picture', defining the feature requirements and deciding the priority of backlog items. THE SCRUM master focuses on a lower level than the project owner, and is there to facilitate development, assigning tasks to team members, and identifying solutions to problems which arise during a sprint.

The development team, then, are simply those writing the code. In translating the roles of each team member to a solo project, the role of developer is unchanged in that the solo developer is doing the work the development team would do in any case.

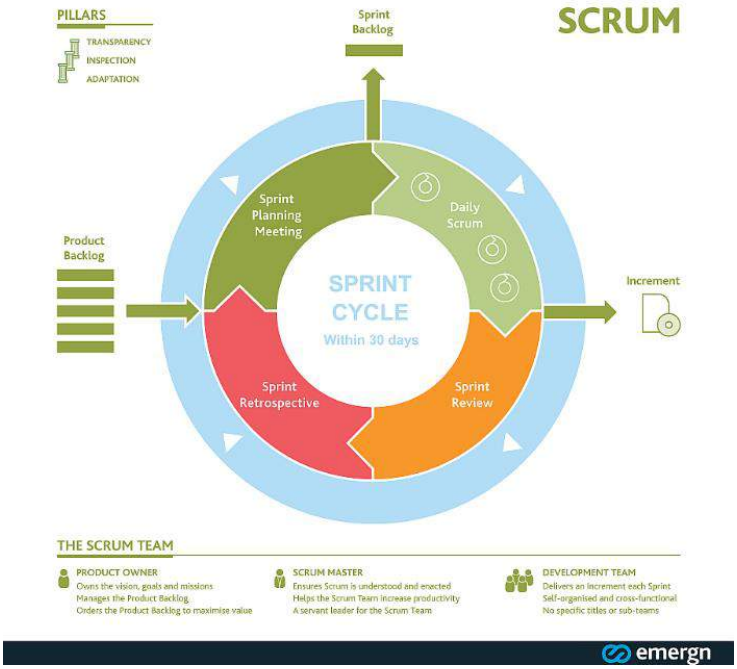


Figure 32 - The SCRUM Methodology. (Emergn, n.d)

In practical terms, SCRUM was implemented in this project by conducting a meeting between the developer and the project supervisor every two weeks, in order to update on progress and plan the next sprint. A retrospective occurred every sprint in the sense that a new sprint review was added to the implementation chapter at the end of each sprint, and the backlog was initially supported by the creation of a Kanban board using the Jira project management software. (Lucidchart, n.d)

5.3 Development environment

5.3.1 WebStorm

The project's development environment consists of a number of tools, the most important of which is the IDE. Both WebStorm and Visual Studio Code are very popular choices for web development, but Visual Studio Code is a text editor, and not a true full-featured Integrated Development Environment. Visual Studio Code has an extensive ecosystem of plugins available, but much of the functionality Visual Studio Code requires plugins for is included in WebStorm out of the box. WebStorm includes advanced coding assistance features, including features specific to React, such as the ability to extract React components out of their parent and into their own file, simply by pressing F6. WebStorm has far more involved version control integration than Visual Studio Code and is fully integrated with GitHub without the need for plugins. It includes the ability to manage branches and commits, as well as support for creating, reviewing, and merging pull requests. While there are many reasons WebStorm could be considered a superior editor to Visual Studio Code, the developer of this project particularly favours WebStorm's built-in debugger and found it easier to configure and use than Visual Studio Code's. Despite this, WebStorm is a demanding software to run in terms of hardware resources, and so when developing directly on the Raspberry Pi, Visual Studio Code was used instead, as it is far more lightweight.

5.3.2 Insomnia

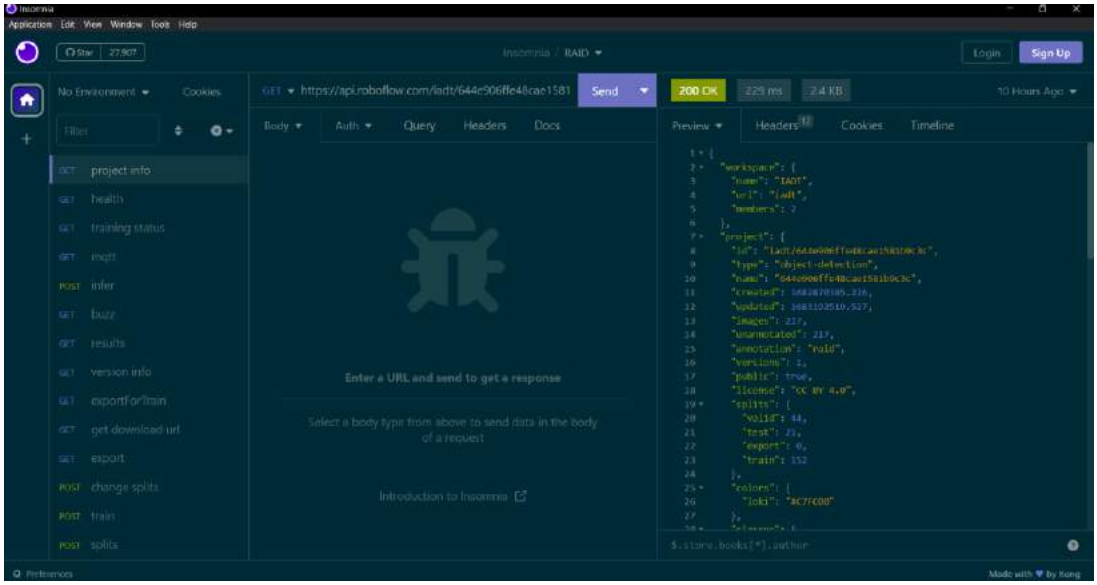


Figure 33 - The Insomnia client featuring some REST endpoints.

For testing the various REST endpoints, the open-source Insomnia API client was used to aid in designing, debugging, and testing, particularly in relation to the middleman server's REST endpoints, and the Roboflow API. Ultimately, the Insomnia client proved a valuable tool or testing and documenting the REST API endpoints.

5.3.3 Doppler

Typically, the developer will create a '.env' file in their server for storing secrets such as API keys or any other sensitive information as environment variables accessible to the server only. This file will be added to a '.gitignore' to ensure the secrets are not pushed to GitHub when a commit is made. However, this approach is cumbersome in that the developer needs to keep a copy of the environment variables on their local machine. This is particularly inconvenient when changing machines, as a copy needs to be kept available at all times. Furthermore, when the server and/or client are hosted on the Internet using Heroku and Vercel, it would be necessary to manually provide the hosted version with a copy of the secrets.

The Github Student Developer plan provides free student access to Doppler, a secret management service which includes the ability to seamlessly integrate with hosting services like Vercel and Heroku and utilises its own command line interface for integrating with the local development environment. Doppler's ability to integrate across platforms means it can synchronise secrets everywhere they are needed, simply by changing them in one place.

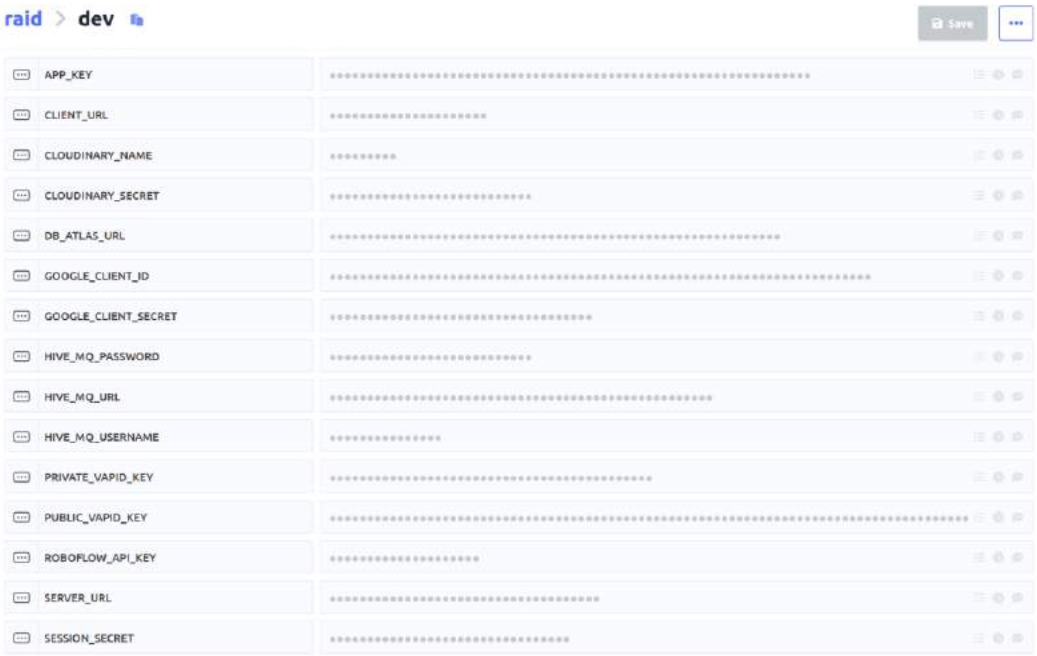


Figure 34 - The list of secrets stored in Doppler.

The figure above shows the entire list of secrets stored using doppler. Heroku and Vercel are then integrated with Doppler, and this entire set of secrets becomes available to both. The local development server then becomes aware of these secrets by installing and authorising the Doppler CLI globally, and prefixing the server's 'run' command with the Doppler CLI start command:

```
"dev": "doppler run -- nodemon server.js ",
```

Figure 35 - Starting the server, prefixed by the Doppler run command.

5.4 Sprint 1

5.4.1 Goal

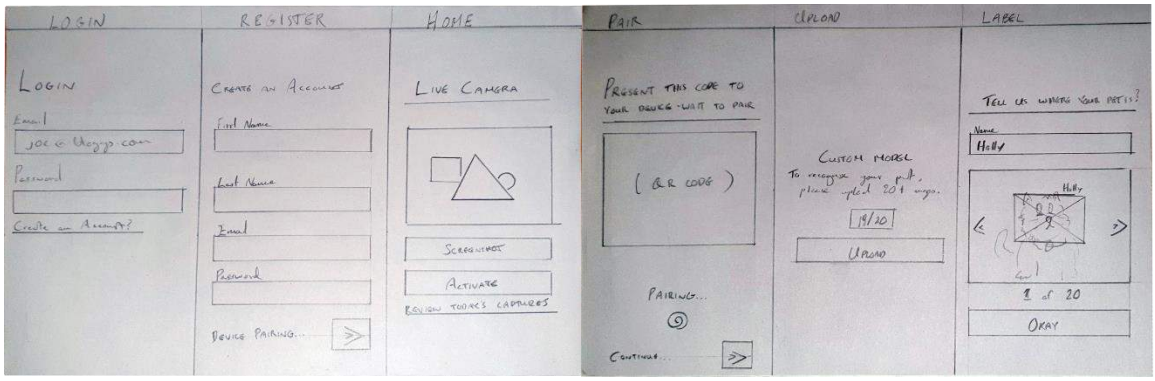
The goal for the first sprint was to finalise the research document and develop a clearer understanding of how the project is intended to work by creating initial prototypes, helping to plan the intended features and screens for the application.

5.4.2 Item 1 – Literature Review & Research

The most important goal for the first sprint was to finalise the literature review, which served to identify what applications existed literature and the public domain which would prove it's possible to implement RAID's features. Eight different studies were consulted to determine how other computer vision practitioners had applied technologies such as object detection algorithms to create applications with similar features to RAID. The results of the research determined that it is feasible to create this application in principle and gave me an awareness of the possibility of using cloud services to augment the computing capabilities of the Raspberry Pi.

5.4.3 Item 2 – Paper Prototype & Figma v1

Having determined that animal recognition could be done using the Raspberry Pi, the next goal for the first sprint was to develop a high-level understanding of what the application should look like. To begin with, a paper prototype was drawn illustrating the pages and basic functionality I expected the app to need.



An initial hi-fi prototype was also created in Figma, a preliminary font scale and colour palette selected, and the logo designed.

5.5 Sprint 2

5.5.1 Goal

The goal of the second sprint was to clear the backlog from sprint one, complete the first iteration of the requirements document, and create the design document.

5.5.2 Item 1 – Requirements Gathering

While the literature review highlighted some successful implementations of animal facial recognition, in the requirements chapter, we looked at some more practical attempts at creating systems featuring similar functionality to those we hope to implement for RAID and gave some insight into the difficulties other creators faced in trying to recognise animals.

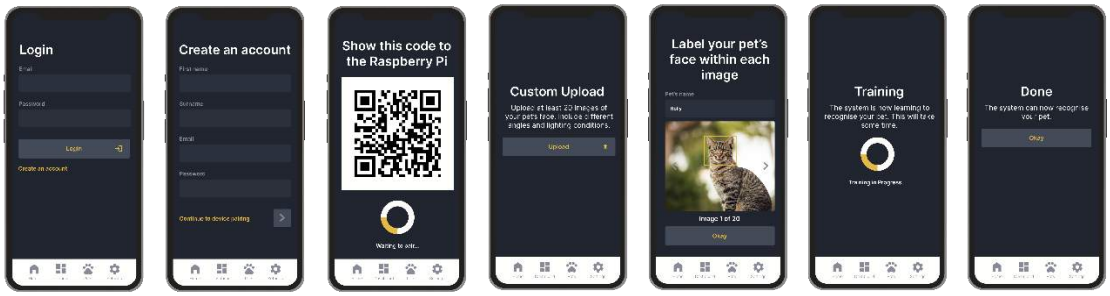
The most informative part of the requirements gathering aspect of sprint two was the response received from the survey. Receiving feedback from pet owners the world over provided valuable insights into the issues they faced. For example, I hadn't considered that the system, for some people, could serve as a deterrent to animals posing a real danger to their outdoor pets, such as the

survey respondent from the United States whose dog was attacked by a porcupine, or others who deal with wild animals sleeping in their pet's bed, harassing them, or eating their food.

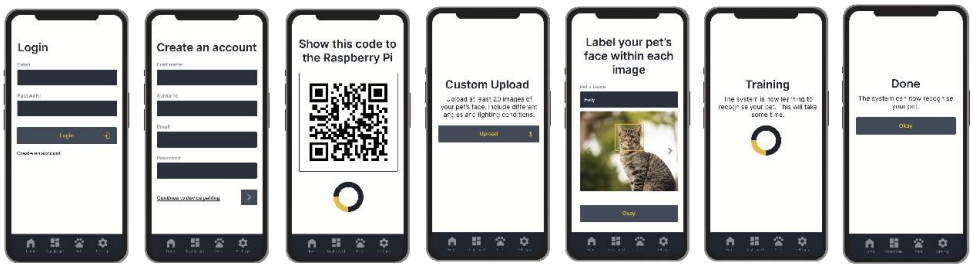
Respondents rated functional and non-functional requirements for the application by order of importance and offered feedback on their overall thoughts on the design of the prototype, or any features they felt were missing.

5.5.3 Item 2 – Prototype Updates

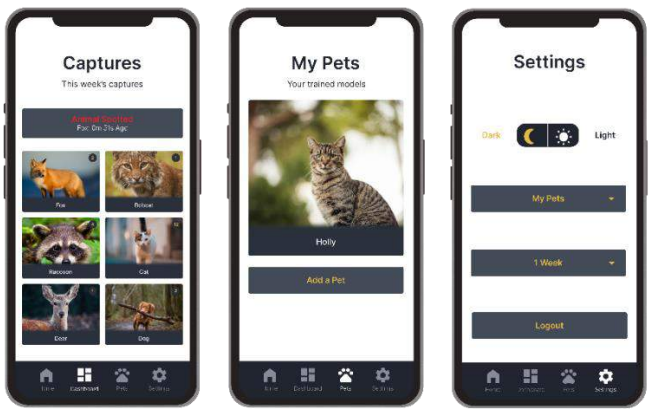
The result of the survey feedback was the development of version two of the Figma prototype. Some users felt the font needed to be larger, and a light mode would be easier on the eyes. Many users said the ability to train the system to recognise multiple animals was of great importance to them, so the prototype was modified to accommodate request. Users also felt a gallery should be added for viewing previous pet captures, and believed push notifications should be implemented to alert users to an animal sighting. To reiterate the changes highlighted in the design chapter, the updated dark mode design is below:



A light mode was also added, with colours modified to allow for higher contrast:



As requested, additional pages were added to facilitate features users felt were important:



A bottom navigation menu was also added to simplify the user flow. A collapsible burger menu was considered, but I felt this was impractical, as it would only serve to add an unnecessary click to the steps leading between the pages.

5.6 Sprint 3

5.6.1 Goal

Sprint three was the beginning of the implementation phase of the project's development. The goal for this sprint was to clear the backlog of the requirements and design chapter in order to begin implementation. V2 of the Figma prototype was to be finalised, and previous documents including the design, requirements, and research were all to be updated.

5.6.2 Item 1 – Additions to Previous Documents

For this sprint, I feel I spent an inordinate amount of time adding to the previous documents and tweaking the design, to the detriment of the project's implementation. In the previous sprint, the results yielded by the survey were very helpful, but too much time was spent trying to increase the number of responses.

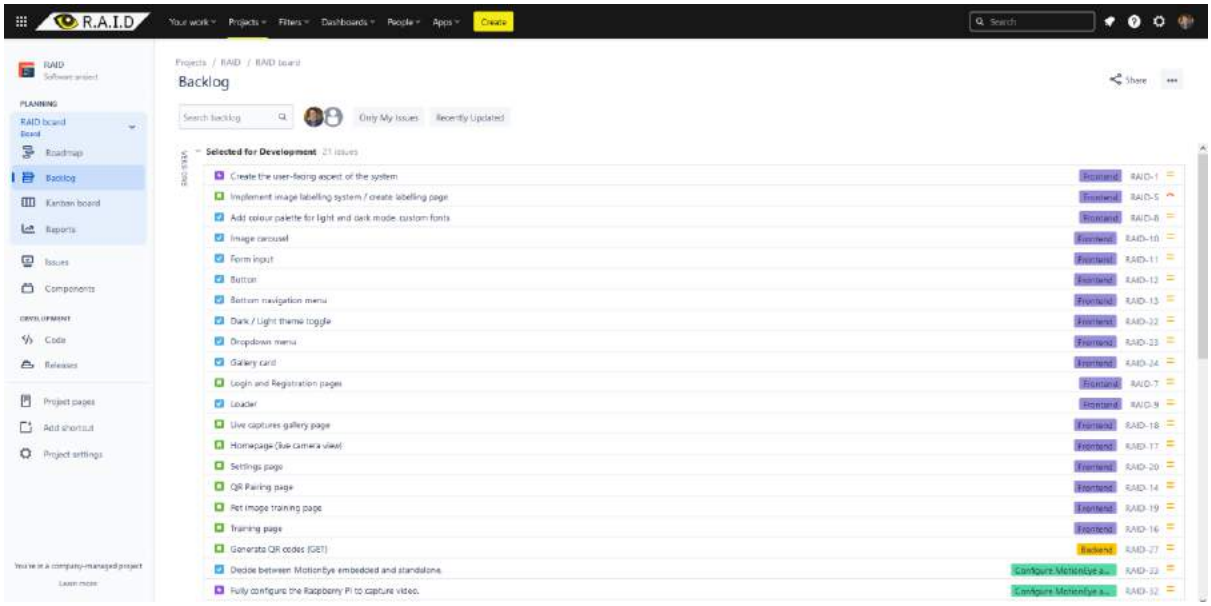
The survey was posted to four animal-related Discord servers, two animal-related subreddits, a garden pond forum site, various social media websites, and sent to the Drogheda Animal Rescue Centre. 35 responses were received, but too much time was spent explaining the survey to people, and updating the questions to accommodate US respondents who were unfamiliar with wording I had included which might be considered specific to an Irish audience.

With the survey complete, most of my remaining time during sprint three was spent updating the Figma prototype and adding interactivity. The highlight colour was changed only slightly, despite significant spent time researching whether yellow was an appropriate colour choice for the RAID branding. It was during this sprint that the image gallery and 'my pets' pages were added, after I realised these pages would be necessary in practice, but were missing from the design.

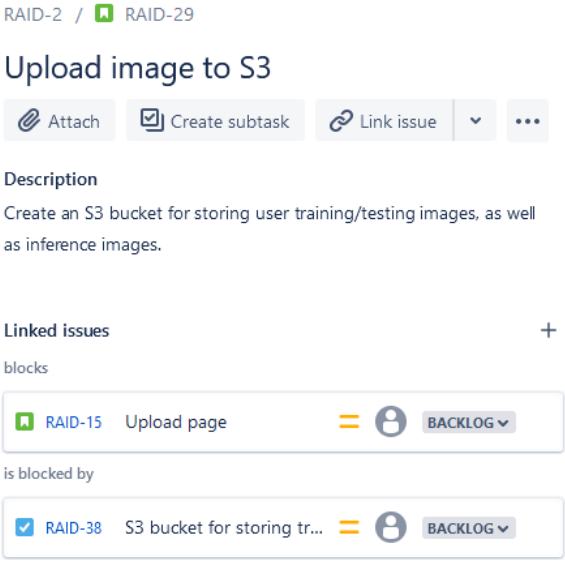
Despite my poor time management during this sprint, the benefit of my time spent focused on prototyping and adding to the design chapter has meant the UI design section of the design chapter is complete, and the technologies section very nearly complete, along with user flow diagrams.

5.6.3 Item 2 – Initial Project Setup, Jira Setup

In preparation for beginning the implementation phase, a Jira board was created. Jira is a project management solution designed specifically for agile software development. I chose to use Jira because of its support for agile workflow practices such as dividing tasks issues into tasks, stories, and epics, assigning difficulty and time estimations to tasks, and its ability to integrate with version control systems and IDEs. Jira features a GitHub integration to create associations between issues and commits and allows the developer to create new issues right from the IDE. Below is a screenshot of the backlog from my Jira dashboard.

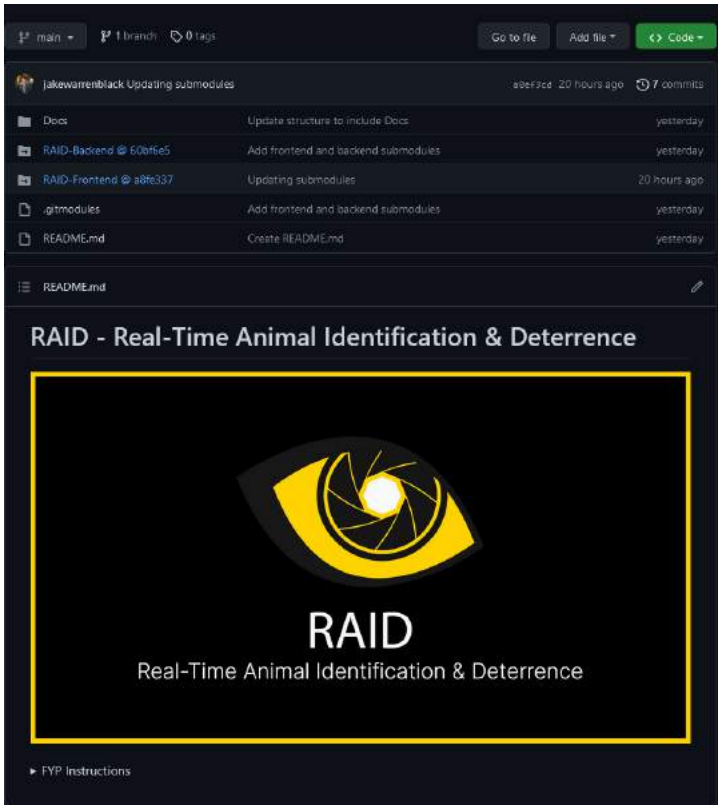


40 tasks have been added to the backlog, divided between epics, stories, and tasks. Some issues have been marked as being blocked by others, meaning one issue can't be resolved until another is. Other issues have been tied to related issues where necessary.



The frontend for the project was initialised locally using Vite, a development tool for scaffolding and bundling web applications. Vite allows developers to make use of the degit project-scaffolding tool to copy another Git repository as a starting template. ViteJS community on Github maintains a list of hundreds of starter templates for popular frameworks and tools. The 'vite-react-tailwind-v3' template was used to create a React application with TailwindCSS preinstalled. The screenshot below shows the starting page provided by the template, which features TailwindCSS styling with no additional configuration required from me.

Finally, the GitHub repository was also configured to contain two other repositories as sub-modules, a RAID-Frontend and RAID-Backend repository.



Submodules allow us to include one repository as a sub-directory of another, which appears visually on GitHub as a link.

5.6.4 Item 3 – AWS Rekognition Evaluation

In terms of actual implementation, a model was trained using Amazon Rekognition Custom Labels, which allows for training a model to recognise specific objects, which distinguishes it from other AWS Rekognition services, which are pre-trained and can only recognise objects from the datasets used by AWS or perform (human) facial recognition.

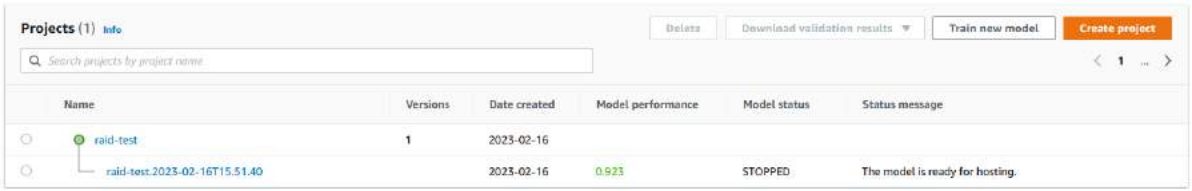


Figure 36 - Dashboard from AWS showing a trained Rekognition Custom Labels model

We saw the implementation of a cat recognition system by Arkaitz Garro in the requirements chapter. Arkaitz used Rekognition trained on its standard dataset, meaning the model was only capable of recognising that the animal it's seeing is a cat, not identifying it as any specific cat.

However, I realised while researching AWS services for this sprint that the AWS offers the Custom Labels version of Rekognition and experimented with training a model using a dataset of 57 images of my dog. The model was trained in less than one hour, which is impressively fast, given that previously, when I trained a YOLO model 'by hand', the training took over six hours to complete. However, in that case, the model achieved a confidence score of almost 60% when performing inference on new pictures of the dog, but the Rekognition model was capable of only 20-25% confidence, which for my purposes is of no use.

```
Jake@DESKTOP-19IKNBI MINGW64 ~
$ aws rekognition detect-custom-labels   --project-version-arn "arn:aws:rekognition:eu-west-1:795967158941:project/raid-test/version/raid-test.2023-02-16
T15.51.40/1676562700256"   --image '{"S3Object": {"Bucket": "test-raid-bucket","Name": "lola_test.jpg"}}'   --region eu-west-1
{
    "CustomLabels": [
        {
            "Name": "lola",
            "Confidence": 25.235998153686523,
            "Geometry": {
                "BoundingBox": {
                    "Width": 0.24461999535560608,
                    "Height": 0.4057300090789795,
                    "Left": 0.20633000135421753,
                    "Top": 0.12160000205039978
                }
            }
        }
    ]
}

Jake@DESKTOP-19IKNBI MINGW64 ~
$ aws rekognition detect-custom-labels   --project-version-arn "arn:aws:rekognition:eu-west-1:795967158941:project/raid-test/version/raid-test.2023-02-16
T15.51.40/1676562700256"   --image '{"S3Object": {"Bucket": "test-raid-bucket","Name": "WIN_20230115_17_27_25_Pro.jpg"}}'   --region eu-west-1
{
    "CustomLabels": [
        {
            "Name": "lola",
            "Confidence": 22.53700065612793,
            "Geometry": {
                "BoundingBox": {
                    "Width": 0.13761000335216522,
                    "Height": 0.2709600031375885,
                    "Left": 0.40511998534202576,
                    "Top": 0.155689999461174
                }
            }
        }
    ]
}

Jake@DESKTOP-19IKNBI MINGW64 ~
$
```

Figure 37 - Using the AWS CLI to run inference using the Rekognition model on two images of my dog stored in S3

Unfortunately, I will need to abandon Rekognition as a possible solution and will now be using AWS SageMaker to train a model myself instead.

5.7 Sprint 4

5.7.1 Goal

My goal for this sprint was to train a custom object detection model on an animal dataset, run live inference on a webcam feed, and stream this feed from the server to the React frontend. I also aimed to capture screenshots of detected movement and send them to Amazon S3 for storage. Finally, I planned to implement an authorisation flow using AWS Lambda, Lambda Authorizer, API Gateway, and Cognito.

5.7.2 Item 1 – Custom Object Detection Model

I began by training a custom object detection model using YOLOv8, the latest single-shot detector from Ultralytics, and the first YOLO implementation to include a Python SDK. A collection of one thousand pre-labelled images was acquired from Roboflow Universe, split into 70% training, 20% validation and 10% testing images.

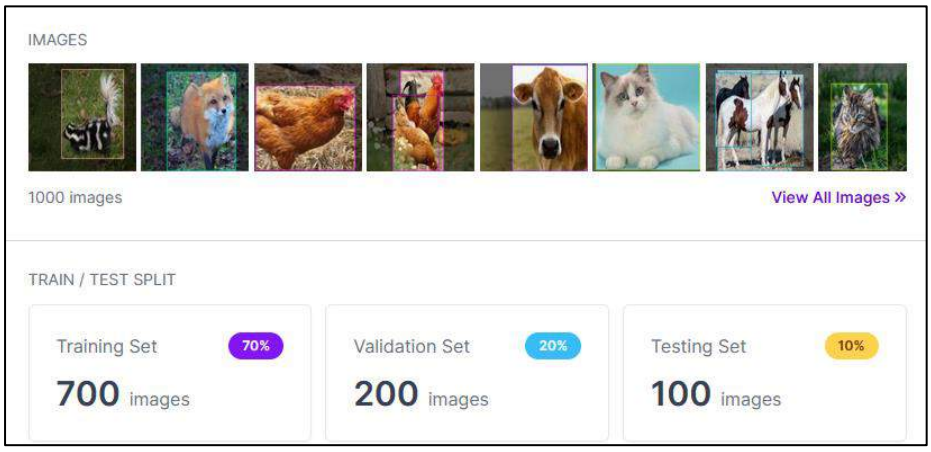


Figure 38 - Animal dataset from Roboflow Universe

The Roboflow Python package was used to import the dataset and select the YOLOv8 format.

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="[redacted]")
project = rf.workspace("iadt").project("animals-detection-ylign")
dataset = project.version(1).download("yolov8")
```

Figure 39 - Using the Roboflow Python package to access the dataset.

For testing purposes, the model was only trained for 25 epochs, equivalent to around 30 minutes, yet performed impressively well, achieving a mean average precision (mAP) of greater than 50% for

all but two out of ten classes (chicken and cow).

```
/content
Ultralytics YOLOv8.0.20 🚀 Python-3.8.10 torch-1.13.1+cu116 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 11129454 parameters, 0 gradients, 28.5 GFLOPs
val: Scanning /content/datasets/animals-detection-1/valid/labels.cache... 200 images, 0 backgrounds, 0 corrupt: 100% 200/200 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 13/13 [00:06<00:00,  1.94it/s]
                   all        200        351      0.814      0.668      0.804      0.543
                   cat        200         18      0.821      0.556      0.797      0.502
               chicken        200         51      0.665      0.427      0.685      0.369
                   cow        200         51      0.628      0.563      0.612      0.351
                   dog        200         26      0.683      0.692      0.767      0.562
                   fox        200         21          1      0.658      0.921      0.729
                  goat        200         32      0.994      0.531      0.782      0.583
                 horse        200         44      0.909       0.75      0.873      0.591
                person        200         52      0.875      0.75       0.809      0.574
                raccon        200         35      0.732      0.829      0.846      0.535
                 skunk        200         21      0.829      0.925      0.965      0.632
Speed: 4.1ms pre-process, 14.3ms inference, 0.0ms loss, 2.1ms post-process per image
```

Figure 40 - Evaluating the performance of the model on each of its classes.

5.7.3 Item 2 – Live Inference on Webcam

5.7.3.1 NodeJS/GStreamer- Livecam & Diffcam-Engine

My first inroad toward implementing a motion-detection enabled livestream used diff-cam-engine, which provides a DiffCamEngine object capable of requesting webcam access via the browser, capturing frames from it, and evaluating motion. This library worked well for detecting motion in the browser, but this is of no use in the context of the RAID project. To solve this, I used Livecam, a NodeJS wrapper library for the GStreamer multimedia framework, which itself is written in C.

GStreamer provides a means to access the webcam, apply various transformations to the feed, and stream it over the local network via the Real-Time Streaming Protocol (RTSP). Options are provided to GStreamer via 'pipelines', with each parameter separated by an exclamation mark, such as in this example, which runs the GStreamer executable, and provide the 'kvideosrc' option, which is the default means of accessing the web camera on Windows. We provide width and height parameters, tell GStreamer to convert frames to jpeg for displaying on-screen:

```
gst-launch-1.0 -v ksvideosrc do-stats=TRUE ! image/jpeg, width=640, height=480 ! jpegdec ! videoconvert ! autovideosink
```

Figure 41 - An example GStreamer pipeline

Livecam provides a convenient wrapper around GStreamer, allowing us to provide these options to the Livecam object rather than running the GStreamer executable directly. For example:

```
// npm install livecam

const LiveCam = require("livecam");
const webcam_server = new LiveCam({
  // address and port of the webcam UI
  ui_addr: "127.0.0.1",
  ui_port: 11000,

  // address and port of the webcam Socket.IO server
  // this server broadcasts GStreamer's video frames
  // for consumption in browser side.
  broadcast_addr: "127.0.0.1",
  broadcast_port: 12000,

  // address and port of GStreamer's tcp sink
  gst_tcp_addr: "127.0.0.1",
  gst_tcp_port: 10000,

  // callback function called when server starts
  start: function () {
    console.log("WebCam server started!");
  },

  // webcam object holds configuration of webcam frames
  webcam: {
    // should width of the frame be resized (default : 0)
    // provide 0 to match webcam input
    width: 800,

    // framerate of the feed (default : 0)
    // provide 0 to match webcam input
    framerate: 25,

    // etc...
  },
});
```

Figure 42 - Livecam's convenience wrapper for GStreamer, providing options via JSON

The Livecam library is no longer maintained, however, and so I created a fork of the library and added several modifications, including updating the library's version of SocketIO, fixing the method for accessing the GStreamer binaries on Linux, and modifying the library's SocketIO server setup to include CORS headers:

```
this.socket.connect( gst_tcp_port, gst_tcp_addr, () => {

    const httpServer = Http.createServer()
                        .listen( broadcast_tcp_port, broadcast_tcp_addr )

    const SocketIO = require( 'socket.io' )(httpServer, {
        cors: {
            origin: "*",
            methods: ["GET", "POST"]
        }
    });


    this.io = SocketIO.listen(
        httpServer
            .listen( broadcast_tcp_port, broadcast_tcp_addr ) );
```

Figure 43 - Providing CORS options to Livecam's SocketIO server implementation.

```
// Look for GStreamer on PATH
var path_dirs = process.env.PATH.split(':');
for (var index = 0; index < path_dirs.length; ++index) {
    try {
        var base = Path.normalize(path_dirs[index]);
        var bin = Path.join(
            base,
            gst_launch_executable);
        FS.accessSync(bin, FS.F_OK);
        detected_path = bin;
    } catch (e) { /* no-op */ }

var bin = '/usr/bin/gst-launch-1.0'

try{
    FS.accessSync(bin, FS.F_OK);
    detected_path = bin;
}
catch(e){
    console.log(e)
}
```

Figure 44 - Correcting Livecam's means of accessing GStreamer on Linux, which did not work on Raspberry Pi OS

```
this.gst_video_src = '-v ksvideosrc do-stats=TRUE';
this.gst_video_src = 'v4l2src ! decodebin';
```

Figure 45 - Using v4l2src, rather than kvideosrc to access the webcam on Linux.

With this done, I also needed to create a fork of DiffCamEngine to prevent the library from requesting access to the webcam via the browser, to receive its frames from a canvas element already present in the DOM instead of the web camera, and to evaluate motion by comparing each frame received from the canvas with the previous frame.

```
function capture() {
        let self = this;

        // Create an ImageData object to store the current frame from the canvas
        let canvasData = canvas.getContext('2d').getImageData(0, 0, canvas.width, canvas.height);

        // If this is the first frame, store it as the background
        if (!this.prevImageData) {
                this.prevImageData = canvasData;
                return;
        }

        // Store the current frame as the previous frame for the next iteration
        this.prevImageData = canvasData;


        // save a full-sized copy of capture
        captureContext.drawImage(video, 0, 0, captureWidth, captureHeight);
        var captureImageData = captureContext.getImageData(0, 0, captureWidth, captureHeight);
        captureContext.drawImage(canvas, 0, 0, canvas.width, canvas.height);
        let captureImageData = captureContext.getImageData(0, 0, canvas.width, canvas.height);
```

Figure 46 - One modification made to DiffCamEngine. Retrieving image data from a canvas, rather than a video element.

I managed to implement all these changes, and successfully streamed a live webcam feed from my NodeJS server running a modified Livecam implementation to a canvas element in the React

frontend, which in turn passed the canvas data to a modified DiffCamEngine for motion detection, which yielded this result:

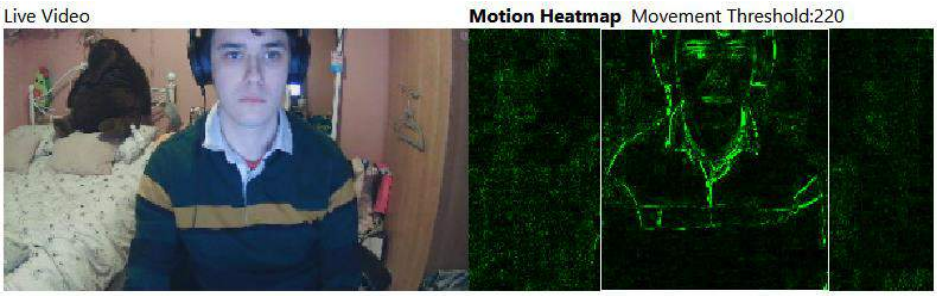


Figure 47 - Streaming video from NodeJS Livecam to the React canvas, which a modified DiffCamEngine evaluates for motion.

When the motion threshold exceeded a predefined limit, a browser alert would appear. Unfortunately, after conferring with my supervisor on the direction the application should take from this point on, we determined that motion detection which occurs in the browser is of no use, considering that the browser would need to remain open constantly for motion detection to occur.

I first attempted to repurpose the existing server and frontend logic by using the opencv4nodejs library to perform object detection on the server-side, but found that opencv4nodejs is no longer maintained, so I again resorted to using a fork. I had difficulty installing this library, despite successfully building OpenCV locally. I opened a GitHub issue, and it appears that the problem has yet to be fixed by the creator. Therefore, I opted to use Python and Flask over NodeJS for handling the camera feed streaming and object detection, as Python more readily supports OpenCV, and even supports the new YOLOv8 Python SDK from Ultralytics.

I had difficulty understanding multithreading and the various async modes available in Python, and believed I needed to make multithreading in order to run OpenCV, YOLOv8 inference, and a SocketIO server at the same time. However, I experienced no issues when running live inference on the webcam and sending the video output to a HTML template generated by Flask. After consulting with my supervisor about it, we determined that a SocketIO setup was not, in fact, necessary, as I had setup the Flask server to stream images over the local network. Instead of using SocketIO, I simply added an image element to the React frontend and provided the address to the Flask server as its ‘src’ attribute.

I have now successfully created a Python/Flask server which runs OpenCV to capture a webcam feed and uses the YOLOv8 SDK to run inference on the webcam frames using my custom-trained animal detection model.

```
results = self.model.predict(frame, conf=0.5)

        # Loop over the list of dictionaries and extract the bounding boxes and class labels for each image in the batch
        for result in results:
            bboxes = result.boxes.boxes
            labels = result.names

            # Draw the bounding boxes on the image
            for bbox, label in zip(bboxes, labels):
                if bbox.numel() > 0:  # check if there are elements in the YOLO frame
                    # x1, y1, x2, y2 = map(int, bbox)

                    x1, y1, x2, y2 = map(int, bbox[:4])
                    cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
                    # Access the last index of the bbox array. The format of bbox is [x1, y1, x2, y2, inference certainty, label index]
                    cv2.putText(frame, labels[int(bbox[-1])], (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
                                2)

        ret, jpeg = cv2.imencode('.jpg', frame)

        # Return the JPEG-encoded frame and the results
        return jpeg.tobytes(), results
```

Figure 48 - Retrieving the bounding box coordinates and object label from YOLO and overlaying them on our video feed.

When an object, such as a person is detected by the YOLO model, we iterate over the bounding box coordinates and label index returned by the model's prediction and use the OpenCV rectangle and putText methods to apply a bounding box and label overlay on the webcam feed. The feed is converted to individual JPEG frames, which are displayed on the frontend. Due to our use of the 'multipart/x-mixed-replace' mimetype on the server, the images are updated constantly, meaning the stream of JPEG frames appears to be a video feed.

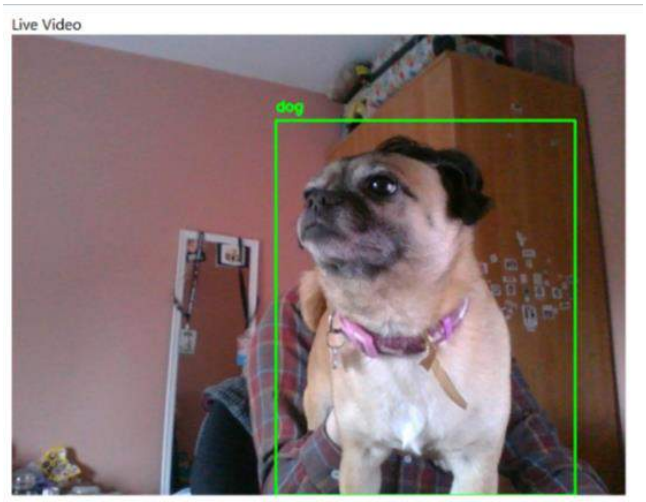


Figure 49 - Live object detection occurring on the server and rendered on the client-side.

5.7.4 Retrospective

Unfortunately, significant time was wasted during this sprint pursuing possible solutions to creating a live webcam feed running object detection on the server side and displaying that feed in a React frontend. Ultimately, however, the main sprint goal was achieved, and the application is now in a good position for continuing development.

5.8 Sprint 5

5.9 Interim Presentation

The interim presentation with my supervisor and second reader consisted of a PowerPoint presentation, a demonstration of the application, an overview of the Figma prototype to illustrate any unimplemented features, and Q/A. The PowerPoint presentation's agenda consisted of an overview of the project background and research findings, an evaluation of existing applications, an iterative overview of the technologies used in the project throughout each of the four versions, and finally an implementation plan detailing how I intend to proceed with the application's development. A brief app demonstration followed, which coincided with the Q/A session.

The technology overview and implementation plan highlighted the difficulty I've had throughout the development process, which led to my supervisor and second reader advising me to simplify the application as much as possible and 'get it over the line', as at this point, fully implementing the originally proposed system seems infeasible.

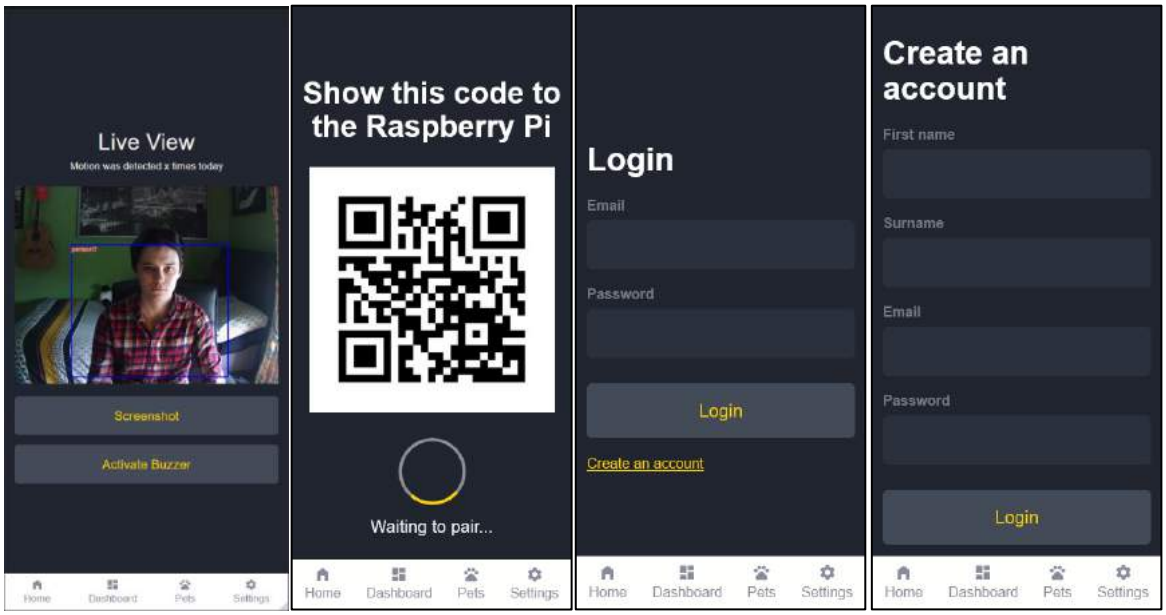
Specifically, based on my supervisor's advice, I now intend to continue the development as if the application is intended as a standalone project for my personal use only, omitting the creation of user accounts, individually trained models per user, and the authorisation flow.

5.10 Goal

Based on the advice received following the interim presentation, I resolved to fully implement the core functionality of the application, if for a single animal only. I aimed to extend the current functionality by training a custom model to detect my dog, Lola meaning the app should be able to detect a dog, take a photo of it, and send it to a hosted model for further inference, returning the result to the server, and passing the results to the frontend via SocketIO.

5.10.1 Item 1 – Switching to NodeJS/Express, Abandoning Python/Flask

Prior to beginning this sprint, the following pages had been added to the application:



Of the four pages, three were purely static. Only the home page featuring the live camera had any real functionality. At the end of the previous sprint, I had encountered difficulty adding SocketIO to the app, and was once again encountering what I believe to have been a race condition, in which SocketIO, Flask, and YOLO were all trying to access webcam frames. I had solved this problem once before using the event-like BaseCamera class from Jian-Kai Wang on GitHub, but found it particularly difficult to solve it again to enable SocketIO events. Fortunately, the NodeJS OpenCV port was updated. Two weeks prior, I had opened a GitHub issue on the opencv4nodejs repository after attempting to install it, as I had already planned to switch to using NodeJS at that point.

After explaining the steps I had taken in attempting to resolve the issue, the author released a new version of the library, which allowed me to abandon my existing Python/Flask code and the multithreading issues it caused, and instead use the more familiar NodeJS and its convenient event loop. I switched back to using the Express server I had created when I first began work on the application and installed opencv4nodejs. I converted a TypeScript example from the repository to

JavaScript and installed a pretrained SSD MobileNet V2 model, which is trained on the COCO dataset. I modified the code to run asynchronously, with the runVideoDetection function waiting for grabFrames function to complete, which in turn waits to receive the finished frame from classifyImg.

```
const grabFrames = async (videoFile, delay, onFrame) => {
  const cap = new cv.VideoCapture(videoFile);
  //let done = false;
  while (true) {
    let frame = cap.read();

    if (frame.empty) {
      cap.reset();
      frame = cap.read();
    }

    // Call the onFrame callback and wait for it to complete
    const result = await onFrame(frame);

    if (result) {
      return result;
    }

    // Delay before reading the next frame
    await new Promise((resolve) => setTimeout(resolve, delay));
  }
};

exports.runVideoDetection = async (src, detect) => {
  let res = await grabFrames(src, 1000, async (frame) => {
    return await detect(frame);
  });
  return res;
};
```

The simplified example below demonstrates how the frame is received within a setInterval in the main server file, which runs every 500 milliseconds. When an image is received from runVideoDetection, the base64arraybuffer method is used to convert the array buffer received into a base64 encoded string, which is then emitted to the frontend via the ‘image’ SocketIO event.

```
setInterval(() => {
  runVideoDetection(0, classifyImg).then(async (res) => {
    let image;

    if(res.img){
      image = encode(res.img)
      io.emit('image', image)
    }
  })
}, 500)
```

I am now also returning the object label received from SSD MobileNet and emitting its value to the frontend via a ‘detection’ SocketIO event.

One issue still to be solved is repeated emitting for the same object detection. While my solution is imperfect, I’ve made some effort at mitigating this. In the simplified example below, we can see that an emitCount variable is initialised to 10. Every time a new object is detected, we emit an event for

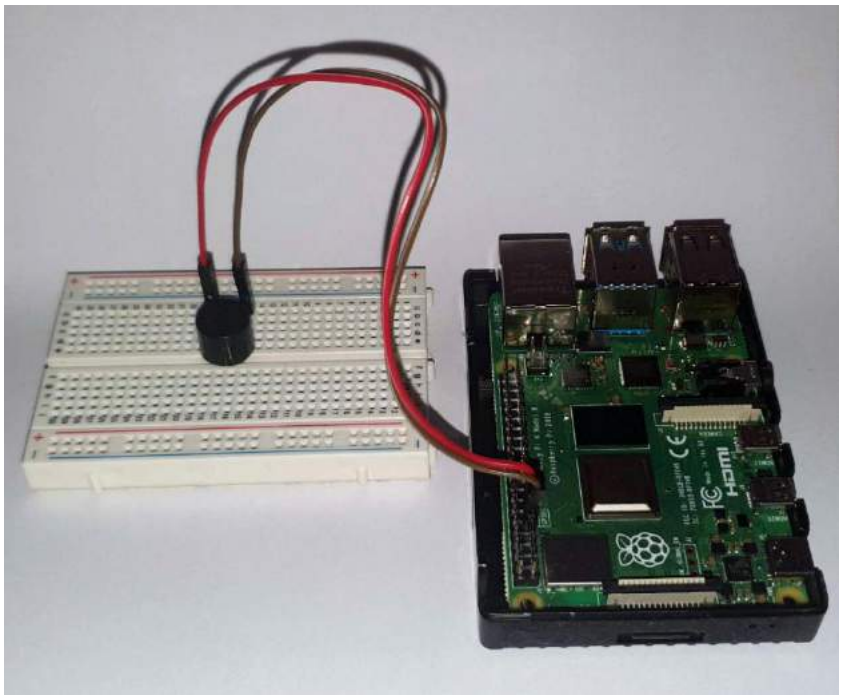
it, and set the prevLabel equal to the name of the object we've just detected. If the same object is detected again immediately, we won't emit an event for it, and will instead decrement the emitCount until it reaches 0, at which point we will emit a new event for the same object. However, if a *different* object from the one stored in prevLabel is detected, even before the timer reaches 0, we will emit a new event anyway.

```
let prevLabel;
let emitCount = 10;

setInterval(() => {

    runVideoDetection(0, classifyImg)
      .then(async (res) => {
        let image;

        if (res.img) {
          // was doing it this way: btoa(String.fromCharCode(...new Uint8Array(res.img)));
          // but caused stack overflow
          image = encode(res.img); // this is an array buffer until converted to base64

          io.emit("image", image);
        }

        if (res.text) {
          // if e.g. we just detected a person, then detect a person again immediately afterward, don't notify for that
          // wait for 10 iterations before emitting for the same object detection again

          // it will STILL emit if the object it's detecting CHANGES
          if (res.text.split(" ")[0] !== prevLabel || emitCount === 0) {
            io.emit("detection", res.text);


            // if it's the case that we've emitted an event for this object already, but emitCount is 0, let it emit again,
            //but reset counter
            emitCount = 10;
          } else {
            // every time we don't emit, decrease the counter,
            // so on the 10th iteration, if we're still detecting the same object as the first time, let the user know
            emitCount--;
          }

          // exclude the confidence score
          prevLabel = res.text.split(" ")[0];
        }
      })
      .catch((e) => {
        console.log(e);
      });

}, 500);
```

This solution works well when only one object is visible. An event is emitted, then the system waits for 10 iterations of the setInterval before emitting a new event for that same object. If a new object is seen, the system also works as intended. However, events are emitted erratically if two different objects appear on-screen at once. It is possible this problem will be less pronounced once I've retrained the SSD MobileNet model on a custom animal dataset, replacing the current COCO-trained model. Once this is done, only animals will be detected, rather than other common objects.

5.10.2 Item 3 – Raspberry Pi Piezo Buzzer

For the physical aspect of the application, I've used the Raspberry Pi 4B, two male to female jumper wires, an active piezo buzzer, and a breadboard. The wiring setup is demonstrated below, demonstrating the connection between the Raspberry Pi's General Purpose Input Output (GPIO) pins, and the active buzzer. I also experimented using a passive buzzer, which is capable of playing at various pitches, but found that one continuous tone seems more than suitable for the purposes of this project.



I've written the following NodeJS script to access the GPIO pins and turn the buzzer on and off. We first import the Gpio module from the onoff library and initialise a new Gpio object at pin 18. Every 150 milliseconds, the buzz method is called, which alternates the buzzer between an on and of state. Finally, once five seconds have elapsed, the stopBuzzing method clears the interval and turns off the buzzer.

```
const Gpio = require('onoff').Gpio; //
 onoff Libary used to interact with the GPIO
const Buzzer = new Gpio(18, 'out'); //
use GPIO pin 18, and specify that it is output
const buzzInterval = setInterval(buzz, 150);

const buzz = () => {
  if (Buzzer.readSync() === 0) { // check if the buzzer is off
    Buzzer.writeSync(1); // turn the buzzer on
  } else {
    Buzzer.writeSync(0); // turn the buzzer off
  }
}

const stopBuzzing = () => { //function to stop blinking
  clearInterval(buzzInterval); // Stop blink intervals
  Buzzer.writeSync(0); // Turn LED off
  Buzzer.unexport(); // Unexport GPIO to free resources
}

setTimeout(stopBuzzing, 5000); // 5 second total duration
```

The buzzer is quite loud and seems ideal for acting as a pest deterrent. Unfortunately, when trying to run my application on the Raspberry Pi (running Raspberry Pi OS), I encountered yet another issue, which seems to be specific to Linux, having also tested the code on Ubuntu. Having attempted to solve the issue in many ways, I've again resorted to opening an issue on the opencv4nodejs GitHub repository, shown below.

jakewarrenblack commented yesterday • edited

Hello, I've managed to install this package without issue on the Raspberry Pi 4B after reading the instructions on this issue:

#67

The example code provided runs fine:

```
backend > JS test.mjs > ...
  import cv from '@u4/opencv4nodejs';
  import {resolve} from 'path';

  export async function applyColorMap() {
      const file = resolve('Lenna.png');
      console.log('loading ', file);
      const image = cv.imread(file);
      console.log('Lenna.png loaded');
      const processedImage = cv.applyColorMap(image, cv.COLORMAP_AUTUMN);
      cv.imwrite(resolve('Lenna2.png'), processedImage);
      console.log('done');
  }
  applyColorMap();
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

❯ node test.mjs
info config no opencv4nodejs section found in /home/jake/College/y4-project-jakewarrenblack/backend/package.json
loading  /home/jake/College/y4-project-jakewarrenblack/backend/Lenna.png
Lenna.png loaded
done
```

But when trying to run my own code, opencv can't open the webcam (device 0)

This runs fine on Windows. For the Raspberry Pi, I've read that only the legacy camera driver supports opencv, so I switched to the legacy driver, but that didn't help. I've read through several similar issues on this repository, the opencv forums, and the original opencv4nodejs repository, but nothing has helped. I understand it's possible this problem has nothing to do with opencv4nodejs itself, and may relate purely to opencv, but any help you can offer is much appreciated.

I've tried running this on both the Raspberry Pi and my laptop running Ubuntu, neither worked.

The error I'm getting looks like this:

```
❯ node server.js
info config no opencv4nodejs section found in /home/jake/College/y4-project-jakewarrenblack/backend/package.json
HTTP Server listening on port 3000
[ WARN:0@1.692] global /home/jake/opencv/opencv-4.6.0-dldsa/opencv/modules/videoio/src/cap_v4l.cpp (902) open VIDEOIO(V4L2:/dev/video0): can't open camera by index
VideoCapture::isOpened() - failed to open capture
[ WARN:0@2.159] global /home/jake/opencv/opencv-4.6.0-dldsa/opencv/modules/videoio/src/cap_v4l.cpp (902) open VIDEOIO(V4L2:/dev/video0): can't open camera by index
VideoCapture::isOpened() - failed to open capture
[ WARN:0@2.712] global /home/jake/opencv/opencv-4.6.0-dldsa/opencv/modules/videoio/src/cap_v4l.cpp (902) open VIDEOIO(V4L2:/dev/video0): can't open camera by index
VideoCapture::isOpened() - failed to open capture
^C
```

It is just a warning rather than an error, but no webcam feed is captured.

Using Python, I have no problem capturing the webcam feed from the same device.

Result of running this script to find valid opencv/ffmpeg devices

```
----------------------------

List of cams with pixel formats supported by openCV :
 [0, 2]

List cams with pixel formats supported by ffmpeg :
 [0, 2]
```

Ubuntu/Raspberry Pi OS - Can't open camera by index #89

jakewarrenblack opened this issue yesterday · 0 comments

```
❯ v4l2-ctl --list-devices
Chicony USB 2.0 Camera: Chicony (usb-0000:00:14.0-2):
        /dev/video0
        /dev/video1

USB Camera: USB Camera (usb-0000:00:14.0-4.2):
        /dev/video2
        /dev/video3
```

I've tried using both those device indexes (0, 1), and -1. I've tried setting the capture method as cv.V4L and cv.V4L2.

I've tried passing a gstreamer pipeline straight into VideoCapture, as suggested here

I read here that I should use the path to the video source e.g. '/dev/video0' instead of just an index, like on Windows, which also didn't work.

The script should have permission to access both /dev/video0' and /dev/video1', so it's not a permissions issue.

I already had ffmpeg installed when installing opencv4nodejs, so I'm not sure if it could be related to this issue.

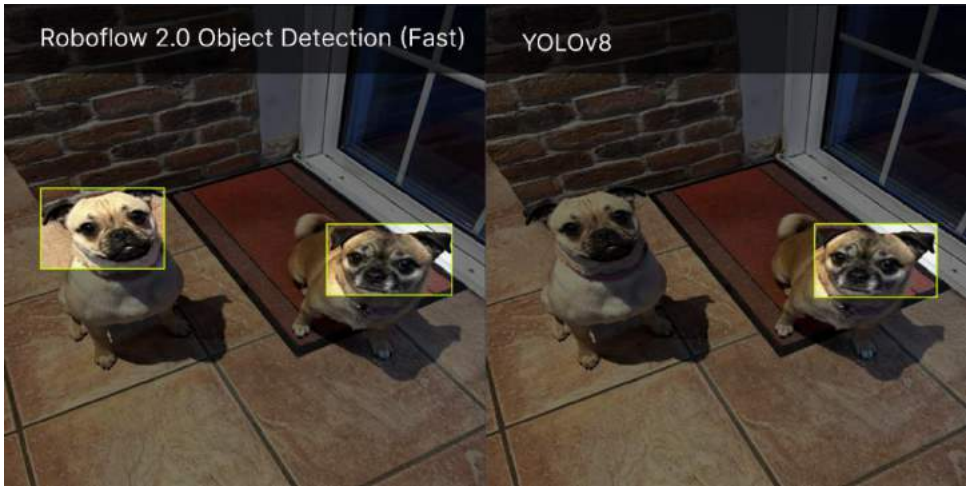
I've also tried the solution mentioned at the bottom here

Could you recommend anything I can do? I'm wondering if maybe I need to use my own opencv build, with gstreamer enabled.

As described, an error is thrown by OpenCV itself, saying that the V4L2 dev/video0 capture device can't be opened by its index. My attempted solutions have included modifying permissions for /dev/video0 and /dev/video1 using the sudo chmod 777 command, providing a path directly to the cv.VideoCapture constructor instead of an index, e.g. '/dev/video0', directly providing a GStreamer pipeline for OpenCV to use, and installing ffmpeg. At present, I've generated a custom OpenCV build with GStreamer enabled, as I suspect the issue may lie with some plugins being disabled by the author of opencv4nodejs, who supplies a prebuilt OpenCV with their library.

5.10.3 Item 4 – Custom Model Training; RoboFlow Train & YOLOv8

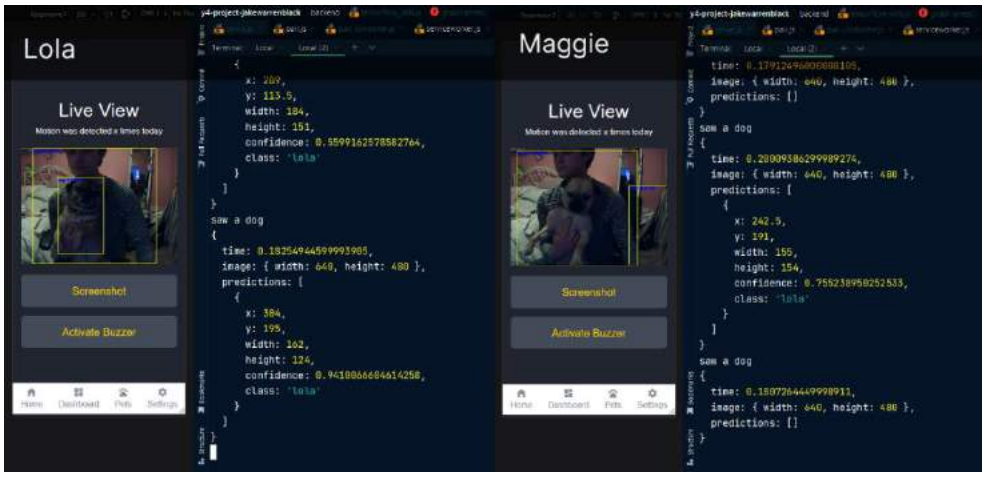
As described earlier in this document, I had previously trained a YOLOv7 model using a small number of images featuring my dog, Lola. While the model was capable of recognising her, its accuracy was not ideal, and so I decided to gather a much larger number of images, label them, and train them first using Roboflow's automated training system, and again by hand using YOLOv8. I gathered and labelled 235 images of Lola and used Roboflow to divide them into a training (70%), testing (20%), and validation set (11%).



Both models perform impressively well at recognising Lola. In the above comparison, however, we can see that in an image containing Lola and Maggie (both pug mixes), Roboflow fails to distinguish the two dogs, and labels them both as Lola, while YOLOv8 correctly recognises only the dog on the right as Lola. Both tests were performed at 70% confidence. When the confidence level is lowered to only 50%, both Roboflow and YOLOv8 incorrectly predict that both dogs are Lola.

Having tested this with several images, YOLOv8 seems to outperform Roboflow. However, Roboflow's performance is impressive considering that this training is a 'one-click' system, which took only around two hours two complete. By comparison, training YOLOv8 using Google Colab required purchasing premium compute hours for access to a GPU, and 1500 epochs of training took roughly six hours.

Roboflow conveniently provides a hosted endpoint for trained computer vision models, and so a connection to the live YOLOv8 model is now integrated into the application. When SSD MobileNet detects a dog, a webcam frame is captured and sent to Roboflow for YOLOv8 to run inference. The result is then returned, which we can emit to the frontend using SocketIO.



The above image shows the system running inference on Lola (left) and Maggie (right), along with the corresponding console output occurring while the system is running. The system initially returned a confidence of 59% for its detection of Lola while her face was obscured, and ran a second time when she faced the camera, returning a confidence of 94%. On the other hand, the system initially provided a confidence of 75% for Maggie, followed by no result, or 0%. It can be difficult to interpret these results at times, considering that there is delay of roughly two seconds on the

camera feed, due to the strain placed on hardware resources when running constant object detection. It is possible that the initial result of 75% for Maggie was actually run on an image of Lola. While the system is imperfect and will sometimes fail to distinguish between these two similar dogs, I believe the problem could be mitigated if a minimum threshold of, say, 75% were set for its detections, or possibly if a series of frames were captured and detections run against them, and the average of a set of results was used as the final verdict.

```
if (res.text.split(" ")[0] === "dog") {
  console.log("saw a dog");

  await axios({
    method: "POST",
    url: "https://detect.roboflow.com/lola/4",
    params: {
      api_key: "s1ghwb35772bcbxmvj64",
    },
    data: image,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  })
    .then((res) => {
      console.log(res.data);
    })
    .catch((e) => {
      console.log(e.message);
    });
}
```

The code above shows how this detection occurs. Within the previously discussed main loop (setInterval), runVideoDetection returns frames and labels from the object detection performed by SSD MobileNet. If a label is returned, we split the label in half, separated by its spaces. A label may look something like ‘dog 0.99251’, so we use only the first index, the name of the detected object, omitting the confidence score. We then make a POST request to our hosted Roboflow endpoint, passing our image, which has been encoded to base64 format. The API then responds with the detection confidence and bounding box coordinates seen in the console output above.

5.10.4 Item 5 – Beginning QR Pairing Implementation

While for now my focus is to see the basic functionality of the application through to completion as a standalone project without user accounts or individually trained custom models, I am hopeful I will still be able to make this app available for others to use, and so I was eager to begin implementing the QR pairing process, the first step required for a new user once they’ve registered.

Beginning on the frontend, when the user visits the ‘pairing’ page, a request is made to the /pair endpoint immediately, which sets a ‘code’ state variable to contain a QR code fetched from the server. The frontend also emits a ‘pair’ event to the server via SocketIO. Once the QR code has been received, it is displayed on-screen. On the server-side, the server generates a Universally Unique Identifier (UUID) and converts it into a QR code.

To handle the problem of stopping the object detection from occurring when the device is in pairing mode, the following middleware is used in conjunction with a conditional statement within our main loop:

```
app.use("/", (req, res, next) => {
  // set isPairing to false for all other endpoints
  app.locals.isPairing = false;
  next();
});

io.on("connection", (socket) => {
  console.log("user connected");

  socket.on("disconnect", function () {
    console.log("user disconnected");
  });

  // use this to trigger pairing mode
  socket.on("pair", () => {
    console.log("frontend wants to pair");
    app.locals.isPairing = true;
  });

  app.locals.isPairing = false;
});
```

This middleware sets the app.locals.isPairing variable to false when the user visits any endpoint. However, if the SocketIO connection is established and the server receives a 'pair' event from the client, this variable is set to true. Note that the variable is also false by default when a connection is established, so if the user went to the login page, for example, pairing mode would not be enabled.

In our main loop, this simplified example below demonstrates that if the app.locals.isPairing variable is set to true, the object detection is skipped over, and we instead continuously read QR codes via the webcam:

```
setInterval(() => {
  if (!app.locals.isPairing) {
    // object detection and sending frames, results to client
  } else {
    // detect qr codes
    readQRCode().then((res) => {
      console.log(res);
    });
  }
}, 500);
```

The readQRCode method contains its own readFrame function, which receives frames from the webcam using OpenCV as 'mat' objects, and converts them to base64 strings. We then use the Jimp library to read an image from the base64 buffer. A qrCode object can then be created from this image and its data is decoded.

```
exports.readQRCode = async () => {
  let result;

  const readFrame = async () => {
    if (frameReady) {
      frameReady = false;

      const frame = cap.read();

      // jimp expects a buffer, but opencv returns a 'mat' object. convert accordingly
      const buffer = cv.imencode(".jpg", frame).toString("base64");
      const image = await Jimp.read(Buffer.from(buffer, "base64"));

      const qrcode = new qrCode();
      qrcode.callback = function (err, value) {
        if (err) {
          result = err;
        }

        // Printing the decrypted value
        if (value) {
          result = value.result;
        }

        frameReady = true;
      };
      if (image.bitmap) {
        // Decoding the QR code
        qrcode.decode(image.bitmap);
      }
    }
  };

  return await readFrame().then(() => {
    return result;
  });
};
```

5.10.5 Item 6 – PWA & Web-Push

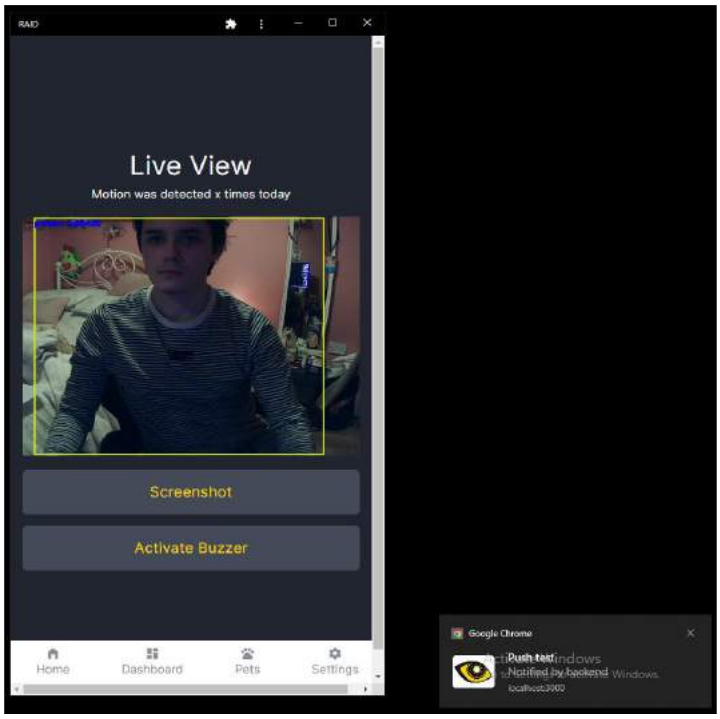
The final thing achieved during this sprint was the beginnings of the app's implementation as a progressive web application. A simple 'offline.html' page is now cached and rendered when the user has no Internet connection, and the frontend can be downloaded to the desktop. The web-push library is now being used in combination with the app's service worker, which now listens for 'push' events from the server and displays desktop notifications.

```
self.addEventListener("push", (event) => {
  // Retrieve the payload
  const data = event.data.json();

  console.log("push has been received");

  // title received from payload (from the server)
  self.registration.showNotification(data.title, {
    body: "Notified by backend",
    icon: "./images/logo-01.png",
  });
});
```

```
// Generate these with .\node_modules\.bin\web-push generate-vapid-keys
// These identify who is sending the push notification
const publicVapidKey = "BM6G-d8QYWAUkjhdwr5umb107dJv9DF2sn9_mTTsz0KHyYArGYkaw4Z7X0fbdKWAKk";
const privateVapidKey = "oZ7g_dLq1-25ujv368-8K3Ec";

webPush.setVapidDetails(
  "mailto:jakewarrenblack01@gmail.com",
  publicVapidKey,
  privateVapidKey
);

// Responsible for sending notifications to the service worker
app.post("/subscribe", (req, res) => {
  // Get pushSubscription object
  const subscription = req.body;

  res.status(201).json({});

  // Payload is optional
  const payload = JSON.stringify({ title: "Push test" });

  // Pass that object into sendNotification
  webPush
    .sendNotification(subscription, payload)
    .catch((e) => console.error(e));
});
```

The initial code in the service worker file itself registers an event listener and uses the showNotification method to display the data received from the server on the screen. The second image shows our server-side code, which creates our /subscribe endpoint and passes a payload to the client via the webPush.sendNotification method. Finally, in our client's index.html file, we the same publicVapidKey defined in the server to register a subscription using the 'pushManager' web API, having registered our service worker file. A fetch POST request is then made to the /subscribe endpoint to test our notifications.

Our PWA is shown below alongside the push notification received when the app opened:



5.11 Retrospective

Overall, this sprint has been quite successful, despite having started the application again from scratch using NodeJS, meaning most existing code has been scrapped once again. Several features have been implemented, though they remain imperfect and their development will continue to be refined during the next sprint.

5.12 Sprint 6

The immediate goal of this sprint was to train a custom object detection model for performing the initial inference deciding whether an object of interest had been detected, prior to running the fine-grained detection deciding if a specific animal had been spotted or not. Since restarting development in a NodeJS environment, the application has been using an SSD MobileNet V2 model which has been pre-trained on the COCO dataset. Significant experimentation was needed to determine the most efficient method of detecting objects from within the application. Note that this type of object detection is distinct from the YOLOv8 model discussed in the previous sprint chapter, as that model is running on a Roboflow hosted endpoint, while this sprint pertains to a model which will run within the application itself, though Roboflow hosted endpoints were also explored as possible means of running the initial inference.

5.12.1 Creating a Dataset

An existing animal dataset from Sprsiter, D. 2022 was used to train a Roboflow 2.0 Object Detection model via the Roboflow Train service, providing a hosted endpoint for the model. The model appeared to perform somewhat well, achieving a mean average precision (mAP) of 86.6%.

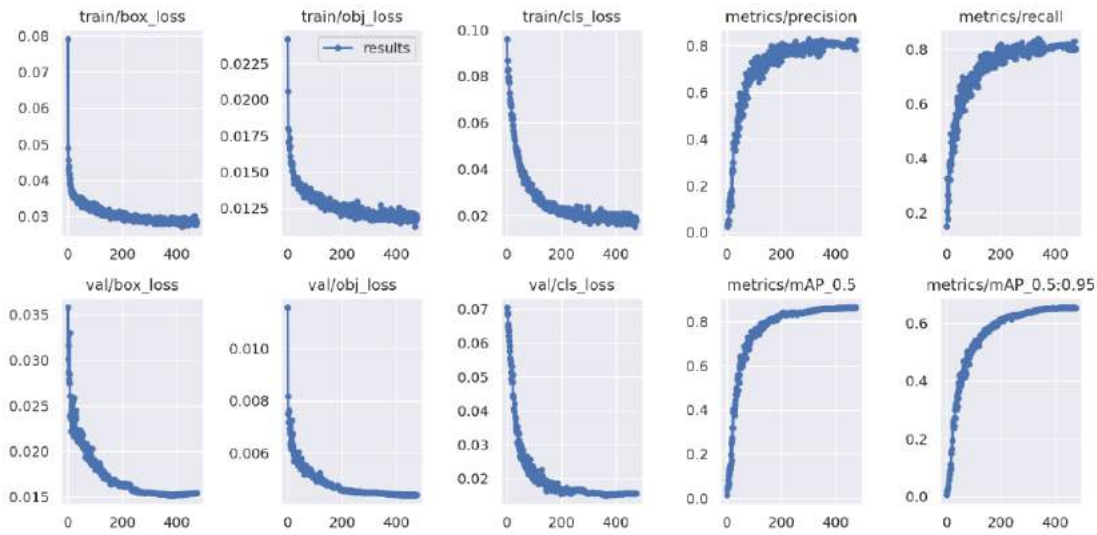


Figure 50 - Training graph for the Roboflow 2.0 Object Detection Model

However, examining the model's performance on the validation set, the accuracy is less than ideal.

In the first and second validation sets above, we notice the model's performance seems satisfactory on wild animals such as the skunk, fox, and chicken, but appears to falter when detecting dogs. In the first validation batch, 7 dogs are identified correctly, but three are mistaken for cows, foxes, and horses. One of the dogs was also detected both as a dog (70%) and as a chicken (90%) simultaneously.

In the second validation batch, a dog and a person are both identified with 90% confidence as 'person', a goat is identified as a dog with 30% confidence, a chicken is misidentified as a person, and in two other cases where both a dog and a person are present, only the person is detected. For our purposes, the accurate detection of dogs, cats, and wild animals is of paramount importance.

The dataset used to train this model was scrapped, and a new aggregate dataset was gathered from several sources. I experienced significant difficulty in creating this dataset. Roboflow offers a 'clone' tool for combining images from publicly available datasets into datasets on your personal account, which worked well when dealing with a small number of images. However, as the size of my dataset increased into the thousands, the Roboflow website began to struggle, and would simply fail to copy

images using the clone tool. Instead, I began downloading images for manual upload, but this would still fail and repeated attempts were needed in order for the browser to allow the upload to begin. An obvious suggestion would be to upload images in small batches, but this would have been exceedingly difficult given that the images were formatted in a specific order to correspond with the text files containing their annotation data. As such, it took repeated attempts to upload a total of almost ten thousand images, which took a total of roughly 20 hours to upload. When this upload had finished, however, each image had somehow been triplicated, bizarrely resulting in a dataset size of 30 thousand images, despite Roboflow's upload limit of 10 thousand. I attempted to train a model using this dataset, but the results were poor, presumably because the model was severely overfitted to its training data due each image being present three times.

5.12.2 The Roboflow CLI

Giving up on using the browser to upload such a large number of images, I resorted to using the Roboflow command line interface, assuming a CLI would be more robust than a website, and capable of processing a larger number of files.

I originally installed the Roboflow CLI via NPM, as recommended in their documentation, but the CLI simply failed silently when attempting to import a dataset. Consulting the source code on the CLI's GitHub repository, I noticed some of the import commands had been commented out. I then forked the repository and debugged it locally, which revealed that the Glob JavaScript library was failing to read paths provided in Windows format and expected them written in POSIX format instead. To correct this, I replaced all forward slashes with back slashes in the file path Glob attempts to read, which fixed the issue of being unable to import files. Further debugging revealed that on Windows, the CLI converts every dataset to the PASCAL VOC data annotation format using its 'toVOC' function, regardless of the data type specified by the user. Rather than attempting to fix this myself, I simply converted my existing dataset to PASCAL VOC format, and uploaded successfully. I informed Roboflow of this problem via their forum and opened a pull request on their repository explaining the cause of the issue and providing my fix for the Windows imports.

Figure 51 - My pull request on the Roboflow CLI GitHub repository

Now that I could upload my data successfully, the Oxford Pets dataset was added first, which includes of 3680 images of 37 breeds of cats and dogs, consisting of roughly 100 images of each.

Expecting that a model trained on this dataset would perform well in differentiating cats and dogs, regardless of whether it could correctly identify the animal's breed, some of the least common cat and dog breeds were removed using Roboflow's data augmentation tools based on popularity data from the American Kennel Club and The International Cat Association, specifically the following breeds:

Cat	**Dog**
Abyssinian	American Bulldog
Birman	Basset Hound
Bombay	Great Pyrenees
Egyptian Mau	Japanese Chin
Russian Blue	Keeshond
Sphynx	Scottish terrier
	Staffordshire Bull Terrier
	Wheaten Terrier
	Leonberger

Wild animals were then added to include classes mentioned in the survey results, from the following sources:

- Bear, chicken, fox, raccoon, rabbit
    - https://universe.roboflow.com/team5/final-9xhp6/browse?queryText=class%3ARaccoon&pageSize=50&startingIndex=0&browseQuery=true
- Deer, bear, coyote
    - https://universe.roboflow.com/animal-detection-rhvda/backyard-animal-detection/browse?queryText=class%3ACoyotes&pageSize=50&startingIndex=0&browseQuery=true
- Wild boar/pig
    - https://universe.roboflow.com/parshant/3try-74igy/browse?queryText=class%3AWild-boar&pageSize=200&startingIndex=0&browseQuery=true#
- Chicken
    - https://universe.roboflow.com/nattamon111-22-hotmail-com/detection-h5oox/browse?queryText=class%3AChicken&pageSize=200&startingIndex=0&browseQuery=true#
- Fox
    - https://universe.roboflow.com/college-qnvl8/animal-detection-4hlxr/browse?queryText=class%3AFox&pageSize=200&startingIndex=400&browseQuery=true
- Porcupine
    - https://universe.roboflow.com/college-qnvl8/animal-detection-4hlxr/browse?queryText=class%3APorcupine&pageSize=200&startingIndex=200&browseQuery=true

100 images of each were initially added to match the number of cat and dog classes. However, after testing the dataset by using it to train a Roboflow 2.0 Object Detection model, it was clear that the detection accuracy was skewed by the comparably much greater number of cat and dog classes when compared with the individual wild animal classes. In other words, the model was biased toward cats and dogs, considering that there were 22 cat and dog classes, or over 1100 images per species, and only 100 for each wild animal species.

The wild animal data was then augmented further, adding as many images of each class as was reasonably possible, from the following sources:

- Raccoon (x358), rabbit (x600), squirrel (x600), deer (x504)
    - https://universe.roboflow.com/new-workspace-uyku6/test-wimda/browse?queryText=class%3ARaccoon&pageSize=200&startingIndex=600&browseQuery=true
- Bear
    - https://universe.roboflow.com/school-uzwrt/home-animals/browse?queryText=class%3Abear&pageSize=50&startingIndex=0&browseQuery=true
- Fox
    - https://universe.roboflow.com/hackvengers/wild-animal-detection-zdn1e/browse?queryText=class%3Afox&pageSize=50&startingIndex=200&browseQuery=true
- Boar, deer, rabbit
    - https://universe.roboflow.com/samuel-e5hya/wildtiere_ba/browse?queryText=class%3Aboar&pageSize=50&startingIndex=50&browseQuery=true
- Chicken
    - https://universe.roboflow.com/object-detection-kefbo/animals-nhm7s/browse?queryText=class%3Achicken&pageSize=200&startingIndex=200&browseQuery=true
- Fox
    - https://universe.roboflow.com/foxes/foxes-fldf1/browse?queryText=&pageSize=50&startingIndex=50&browseQuery=true

In one sense, this corrected the previous issue of the data being overfitted towards cats and dogs but was of no use as now the model was overfitting toward wild animals. This was pre-empted by Roboflow's suggestion in its 'Health check' which indicated that models trained on this data would be overfitted due to the data's overrepresentation of wild animal classes. The existing wild animal data was scrapped and a new one created from scratch.

At this point, the first attempt at creating a new wild animal dataset consisted of 100 images of bears, deer, raccoons, squirrels, rabbits, crows, and sparrows, and 95 images of hedgehogs. This was too little data, and an additional 50 deer, rabbits, and hedgehogs were added, along with an additional 100 bears, squirrels, and crows. After performing another health check, the sparrow class was removed, as there were far too many duplicate images in the dataset. The model was then trained via Roboflow Train again, with detection accuracy improving noticeably. However, a clear issue was the model's inability to recognise when no object of interest was present in a scene. For example, 'person' is not a class in the dataset, so no detections would be expected when a person is in front of the camera. However, the model would always attempt to perform a detection, no matter

the type of object. To correct this, the dataset was further augmented by adding 700 images from the Stanford Background dataset, albeit with any images containing animals manually removed.

Before attempting to train the model again, 100 additional raccoons and 50 additional hedgehogs were added to the dataset, and images of bears were manually modified, as they were overrepresented. Polar bears, giant pandas, and red pandas were all removed, as this is outside the application's use case considering their reclusiveness and the geographical isolation of regions inhabited by these species. Also removed were any animals mislabelled as bears, images of teddy bears, and animated or painted images of bears, leaving black, brown, grizzly, and sun bears as the remaining examples. Finally, data augmentation was applied using Roboflow's class modification tool to merge the Polish-language labels for species of hedgehog 'pigmejski' and 'europjski' were merged into the hedgehog class, and merge uppercase and lowercase class labels for raccoons and crows from various datasets.

The custom wild animal dataset was then merged with the Oxford Pets dataset with all of its 37 original classes included, resulting in a collection of around 100 labelled images of cat and dog breeds, around 200 labelled images each of bears, crows, deer, hedgehogs, rabbits, and raccoons, and squirrels, and roughly 700 null-labelled images of backgrounds, outdoor scenes, and people. A slight over-representation of wild animals has been intentionally included, with the hope that cats and dogs will always be detected accurately, even if the breed prediction is wrong, but a wild animal will generally be detected accurately. This is important for categorising screenshotted images correctly in the application.

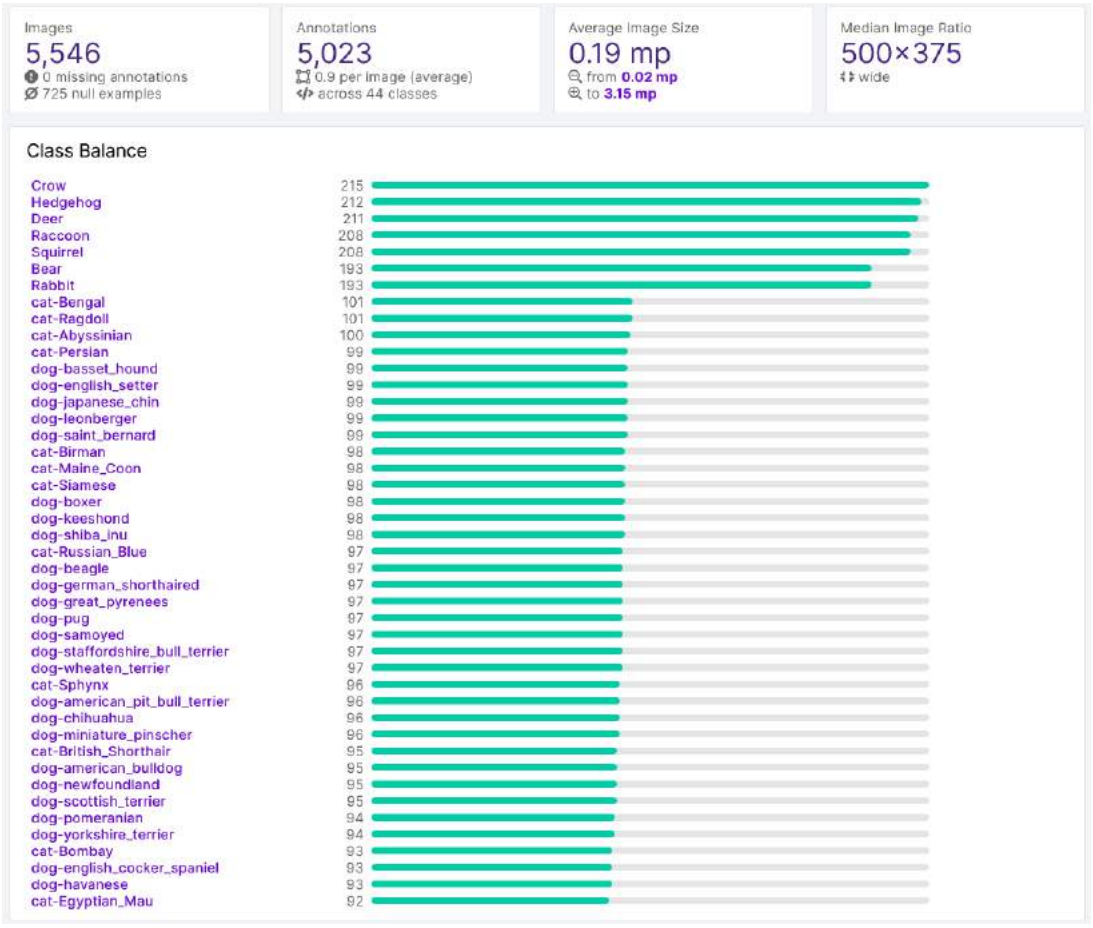
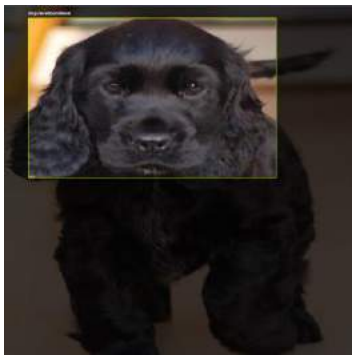


Figure 52 - The final wild animal, cat, dog, and null example dataset

5.12.3 Experimenting with Roboflow Hosted API

A new model was generated using Roboflow Train, which performed reasonably well at predicting dog and cat breeds, identifying wild animals, and returning no prediction for background images.



For models trained on their platform, Roboflow offer an inference server both as a hosted API, and as a Docker container designed for edge devices such as the Raspberry Pi and Nvidia Jetson, which can be run locally. Curious as to how the hosted API would perform in place of the current local SSD MobileNet model, I first experimented with sending webcam frames to Roboflow's server for inference. Where previously webcam images would have been passed directly to the local SSD MobileNet model, they were now sent to the Roboflow API as follows:

```
let axiosimage = cv.imencode('.jpg', frame) // From OpenCV mat object to jpg
axiosimage = encode(axiosimage) // From array buffer to base64 string

await axios({
  method: "POST",
  url: "http://localhost:9001/cats_dogs_and_wild_animals/1",
  params: {
    api_key: "NotZ49lvMpo1QwEINQgR",
  },
  data: axiosimage,
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
}).then(function (response) {

  console.log(response.data);

  let pred = response.data.predictions[0];

  // convert decimal coordinates to pixel coordinates
  const imageWidth = response.data.image.width;
  const imageHeight = response.data.image.height;

  let x1 = Math.round(pred.x * imageWidth);
  let y1 = Math.round(pred.y * imageHeight);
  let x2 = Math.round(pred.width * imageWidth);
  let y2 = Math.round(pred.height * imageHeight);

  let predClass = pred["class"];
  let conf = pred.confidence;

});
```

Where the API acts as a drop-in replacement for the local inference with SSD MobileNet, and a confidence value, class name, and coordinate set is returned from the API. While the server returned reasonably accurate predictions, it was immediately clear that it was significantly slower than the original local model, and so I installed the Roboflow Docker image containing a version of their inference API optimised for use on edge devices. Roboflow's documentation states that the Docker image requires a Raspberry Pi 4 running a 64bit version of Ubuntu, but I found that the image also runs without issue on 64bit Raspberry Pi OS, which makes sense given that Raspberry Pi OS, like

Ubuntu, is built on Debian. After installing the Docker container, the inference server is run, passing the connection through the Raspberry Pi's network card:

```
sudo docker run --net=host roboflow/inference-server:cpu
```

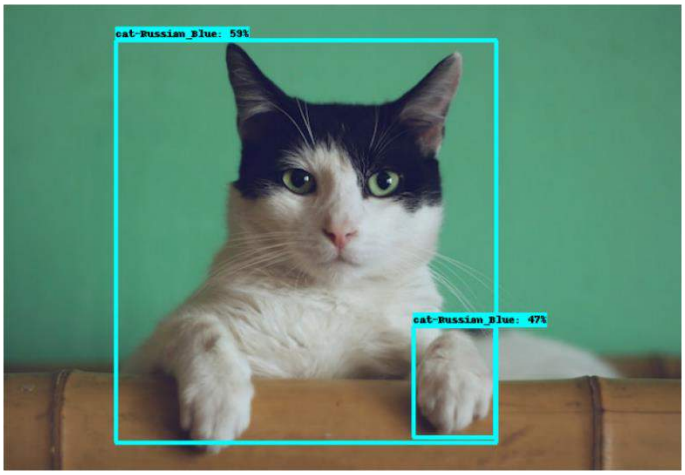
With the inference server now running locally on port 9001, the local inference server can now be used in place of the hosted API:

```
await axios({
    method: "POST",
    url: "http://localhost:9001/cats_dogs_and_wild_animals/1",
    params: {
        api_key: "NotZ49lvMpo1QwEINQgR"
    },
    data: axiosimage,
    headers: {
        "Content-Type": "application/x-www-form-urlencoded"
    }
})
```

However, while the local server was faster than the hosted API, it was still clearly slower than the original SSD MobileNet V2 model, which ran locally within the application itself, and so I began training a custom model using my dataset.

5.12.4 TensorFlow 1 & 2

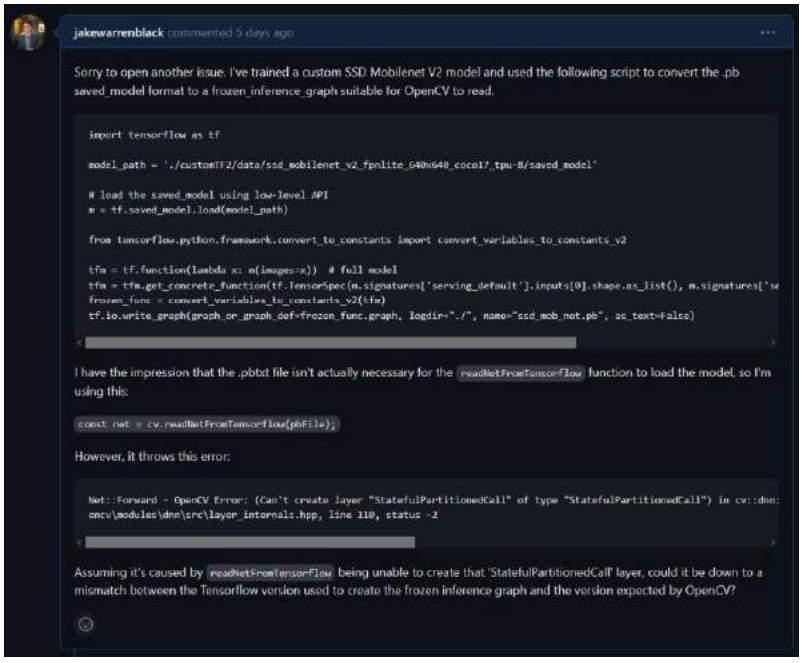
I initially trained a custom SSD Mobilenet V2 model using TensorFlow 2 on Google Colab, by following this notebook. Training for TensorFlow 2 was straightforward, and was further simplified by Roboflow's ability to export a dataset in any format, and so I exported by dataset as TensorFlow TFRecords, and manually added them to the Google Drive folder linked to the Colab Notebook. I trained the model for only 50,000 first to ensure this method of training would be compatible with OpenCV, which resulted in a loss of 0.58, far higher than the ideal loss of less than 0.1, but this was to be expected with only 50,000 epochs of training, given that the SSD Mobilenet V2 pipeline configuration was set at 200,000 epochs by default. Despite this very short training time, the resulting model still seemed capable of making predictions:



However, issues began to arise when I tried to use the exported model weights in my application. OpenCV threw an error when reading the .pb weights file saying “Cannot create layer ‘StatefulPartitionedCall’ of type ‘StatefulPartitionedCall’”. Errors of this type persisted, and I discovered that the frozen inference graph exported by Tensorflow 2 is not suitable for use with OpenCV, as described in this GitHub issue comment from a user describing the same problem:

I tried several approaches to solve this issue, including generating a .pbtxt file using the tf_text_graph_ssd generation script included in the OpenCV repository, converting the model weights to TensorFlow light (.tflite) format, converting the weights to ONNX format, using tfjs-node to read the weights, and reading the files using both opencv4nodejs and opencv-python.

I now believe the issue to have been caused by a version mismatch between the OpenCV version embedded in opencv4nodejs and the version of TensorFlow (version 2) used to create the frozen inference graph. I opened an issue asking this question on the opencv4nodejs GitHub repository but have not received a response from the author.



Failing any other solution, I retrained the model using TensorFlow version 1. Unfortunately, Google Colab no longer supports Python version 3.7, which is required by TensorFlow version 1. Instead, I trained a TensorFlow 1 model locally. I faced some difficulty installing an old version of TensorFlow locally. To do this, I created an Anaconda virtual environment using Python 3.7, installed numpy version 1.19.5, keras 2.1.6, and protobuf version 3.2.1, older versions of these packages are all

required in order to run the original TensorFlow. This Colab notebook was used for training TensorFlow 1. Certain aspects of the notebook were skipped over, considering that it was trained locally rather than on Google Colab as intended. Steps involving creating TFRecord files and populating the train and valid directories specifically were ignored, as I already had this data from Roboflow's export feature. Again, the model was only trained for 50,000 epochs as a test to ensure OpenCV supported it. The resulting model performed extremely poorly, but this was not an issue, as the training only sought to determine whether TensorFlow 1 was capable of producing an SSD MobileNet V2 model which was readable by OpenCV. The final model resulted in a loss value of 11.6, which is extremely high, considering that an ideal value is less than 1.

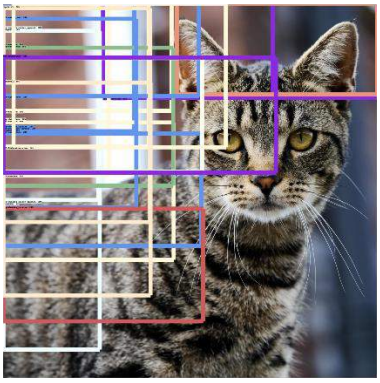


Figure 53 - The result of testing the model demonstrating the (expected) poor results

Once again, I attempted to export a frozen inference graph using the generation script included in the notebook, which did not work. I again tried using the generated .pb and .pbtxt files in both opencv4nodejs and opencv-python; neither could read the files. OpenCV threw the following error in attempting to read the model:

"Net::Forward - OpenCV Error: (Can't create layer "TFLite_Detection_PostProcess" of type "TFLite_Detection_PostProcess") in cv::dnn::dnn4_v20220524::detail::LayerData::getLayerInstance, in file c:\build\master_winpack-build-win64-vc14\opencv\modules\dnn\src\layer_internals.hpp, line 110, status -2"

Once again I tried the previously described steps of converting to TensorFlow Lite format, and attempted to use the TensorFlow export_tflite_ssd_graph.py script, and the tf_text_graph_ssd script to generate the .pb and .pbtxt files. Considering that neither TensorFlow 1 nor TensorFlow 2 seemed capable of producing a model readable by OpenCV, I decided to switch to YOLOv4-Tiny.

5.12.5 YOLOv4-Tiny

To begin with, I took an example from the original opencv4nodejs repository rather than the fork by GitHub user UrielCH, as the original repository includes JavaScript, rather than TypeScript examples. Downloading the .cfg and .weights files for a YOLOv4-Tiny model trained on the COCO dataset, I confirmed that opencv4nodejs could run YOLOv4-Tiny without issue, and began training my own custom model. Considering how long a YOLOv4-Tiny model seemed to need for training, this Colab Notebook was used locally, to avoid paying Google Colab's compute resource fees. The following code was taken from a YOLOv4-Tiny example notebook produced by Roboflow:

```
#Set up training file directories for custom dataset
dataset = '/content/darknet/data'

%cd /content/darknet/
%cp {dataset}/train/_darknet.labels data/obj.names
%mkdir data/obj
#copy image and labels
%cp {dataset}/train/*.jpg data/obj/
%cp {dataset}/valid/*.jpg data/obj/

%cp {dataset}/train/*.txt data/obj/
%cp {dataset}/valid/*.txt data/obj/


with open('data/obj.data', 'w') as out:
  out.write('classes = 3\n')
  out.write('train = data/train.txt\n')
  out.write('valid = data/valid.txt\n')
  out.write('names = data/obj.names\n')
  out.write('backup = backup/')

#write train file (just the image list)
import os

with open('data/train.txt', 'w') as out:
  for img in [f for f in os.listdir(dataset + '/train') if f.endswith('jpg')]:
    out.write('data/obj/' + img + '\n')

#write the valid file (just the image list)
import os

with open('data/valid.txt', 'w') as out:
  for img in [f for f in os.listdir(dataset + '/valid') if f.endswith('jpg')]:
    out.write('data/obj/' + img + '\n')
```

The number of classes used was, of course, changed from 3 to 44 for our dataset. This code facilitated the use of my existing dataset exported from Roboflow in the YOLOv4 PyTorch format. Considering that I trained this model locally, it was necessary to install CUDA and cuDNN to avail of my machine's GPU, a CUDA-capable NVIDIA GTX 1060 with 6GB of VRAM. I had previously attempted to install CUDA and cuDNN on Windows, which proved difficult. For ease of installation, I instead went through the installation process on Pop_OS, a Linux distribution based on Ubuntu.

To allow YOLOv4 to take advantage of the GPU hardware, this Gist was followed, with some changes necessary. When the line 'wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.2.1.32/11.3_06072021/cudnn-11.3-linux-x64-v8.2.1.32.tgz' was reached, it was necessary to open the link and download the NVIDIA cuDNN tarfile by hand, as the wget utility download the file as HTML, rather than a tar file, which could be unzipped. I then had to modify the permissions for this file using the command 'sudo chmod 777 long_tarfile_name.tgz', and finally ran 'tar -xzvf ${CUDNN_TAR_FILE}'. After this, a restart was necessary. It was then necessary to 'cd' into the ' ~/cuda/include' directory, and copy the following files to the '/usr/local/cuda-11.3/include' directory:

- cudnn_version.h
- cudnn_ops_infer.h
- cudnn_ops_train.h
- cudnn_adv_infer.h
- cudnn_adv_train.h
- cudnn_cnn_infer.h
- cudnn_cnn_train.h
- cudnn_backend.h

I then added two environment variables using the following commands:

'export PATH=/usr/local/cuda-9.0/bin${PATH:+:${PATH}}'

'export LD_LIBRARY_PATH=/usr/local/cuda-9.0/lib${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}'

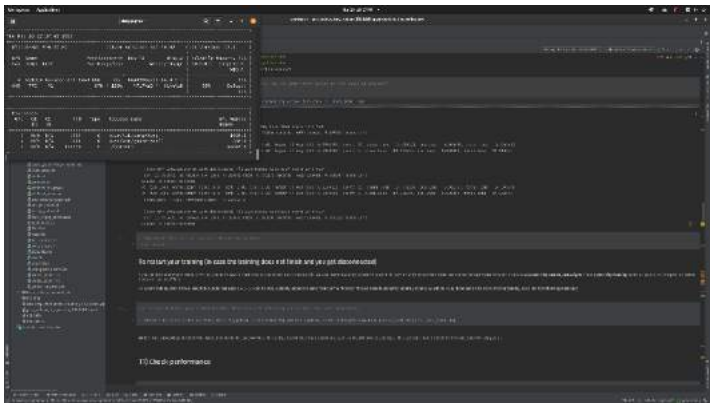
I then ran the nvcc -V command to determine the version of CUDA which had been installed. The command printed CUDA 11.3. However, when running the 'nvidia-smi' command, the OS

determined that CUDA 12.1 was installed. Based on the Darknet makefile used by Darknet, I determined that CUDA 11.3 was the expected version.

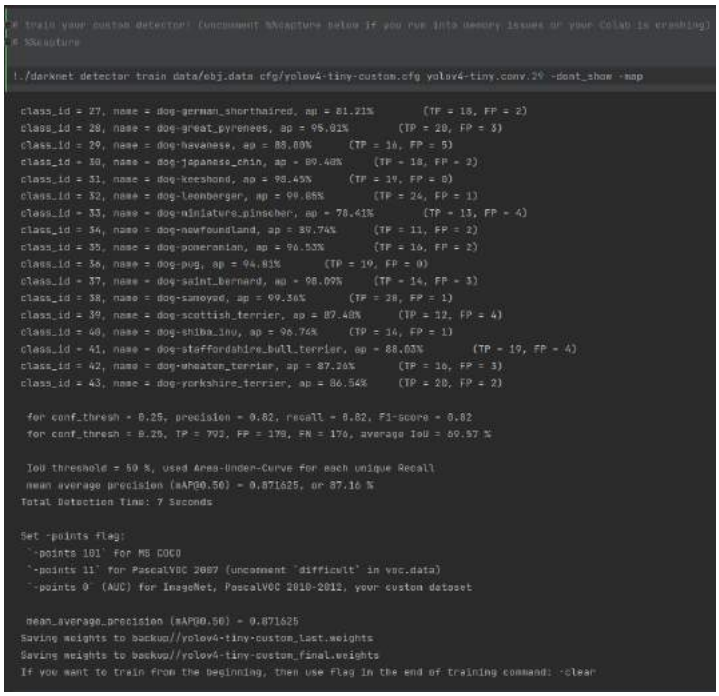
Darknet threw errors relating to the installed version of GCC, a C/C++ compiler being incompatible with the installed version of CUDA, which made sense, given that the default version used was GCC 12, so I added the 'MAX_GCC_VERSION=10' environment variable to my .bashrc to fix this compatibility issue, based on the compatibility chart referenced in this stack overflow answer. Finally, to line 77 of the Darknet makefile, I added the following line, in order to ensure Darknet was finding the versions of CUDA 11.3 and GCC 10 it expected:

'NVCC=/usr/local/cuda-11.3/bin/nvcc -ccbin /usr/bin/g++-10'

Once I had begun training the model, I opened the Pop_OS resource monitor to ensure that the system's VRAM was being used as could be expected, which showed that over 4GB of VRAM had been reserved by the Darknet training process:



After 9 hours of training, the model achieved a mean average precision of 87.16%:



Fortunately, this model was compatible with OpenCV, and after some small modifications to the existing code, it is now running and capable of making detections.

5.12.6 Application Changes – Decoupling the RPi Server from the Middleman, Hosting

Following a meeting with my supervisor, we determined that my next course of action following the completion of my custom-trained YOLOv4-Tiny model, I should decouple the Express server running on my Raspberry Pi from the React client and create a 'middleman' server which will be hosted on a cloud server to act as a go-between for the client and the Raspberry Pi. In this way, the middleman server could act as a means of conveying data from the Pi to the client, meaning the app could run over the Internet, rather than purely over LAN.

Since then, I've removed all functionality from the Raspberry Pi server not relating to object detection and video processing and added a second Express server to act as the middleman between the Pi and the React client. Images are now sent from the Raspberry Pi to the middleman server using Socket.IO, and the middleman server, in turn, sends these frames to the client.

Having achieved this, I needed to host the middleman server on a cloud service. I first experimented with Cyclic.sh but found that Cyclic do not support integration with GitHub organisations, which means I cannot use their service with my final year project repository, as this is managed by IADT via a GitHub organisation. I then hosted the server on Digital Ocean via their NGINX reverse proxy service. Following Digital Ocean's instructions, I used SSH to access the template server hosted on their service and cloned my middleman server. After installing the necessary node modules, I successfully hosted the middleman server on a Digital Ocean 'droplet'.

However, I felt that Heroku would be better suited for my use-case, given that I've used Heroku in the past and am already familiar with it. I have now hosted the middleman server on Heroku instead. With this completed, I hosted the React client on Vercel, and have now established a connection between the Raspberry Pi, the middleman server on Heroku, and the React client on Vercel, meaning the video stream is now accessible over the Internet.

Finally, I have now integrated the Doppler secrets management service with my local application, the React client hosted on Vercel, and the middleman server on Heroku. Doppler allows for simple management of secrets such as API keys, and their integration with various hosting services means that Vercel and Heroku have now been populated with secrets stored in Doppler. With this established, any time secrets are updated on Doppler, they will be recreated anywhere else that Doppler has populated. Having installed the Doppler CLI, I've now changed the NPM run script for the application to the 'doppler run -- npm run dev' command. This ensures that all client secrets the application attempts to access via 'process.env['CLIENT_SECRET'] will be automatically populated by Doppler, even though there is no .env file present in the application locally.

5.12.7 Retrospective

The training of a custom model capable of integrating with opencv4nodejs has proven difficult, and a significant amount of time spent attempting to train an SSD MobileNet V2 model using TensorFlow versions 1 and 2 during this sprint, culminating in switching the model used entirely over to the Darknet YOLOv4-Tiny model. However, significant progress has now been made in that a custom model has now been successfully trained which is capable of making predictions as to the breed of cat or dog breeds presented to it. The model also seems more than capable of detecting the various wild animal species included in the dataset's classes, and seems to have moved past the issue of detecting objects where no object of interest is present. Finally, a significant hurdle has been overcome in that the model is now hosted live over the Internet, which should allow for significant progress to be made in the coming weeks with regard to authentication, establishing REST endpoints, and allowing users to train custom models based on their use-case.

5.13 Sprint 7

5.13.1 PassportJS

As discussed in the concluding section of the previous sprint chapter, the application now consists of three parts; the React client on Vercel, the 'middleman' Express server on Heroku, and the Raspberry Pi which connects to the middleman server, but is not itself hosted on the Internet. An issue this presented immediately was the need for a robust authentication solution to prevent unauthorised users from accessing protected parts of the application, particularly the video feed, given that it provides a live stream of my own web camera. PassportJS was chosen as the application's authentication handler. Passport is described as an "Express-compatible authentication middleware for Node.js". The sole purpose of Passport is to provide authentication for requests and does so modularly using a set of plugins known as 'strategies'. In our application, the strategies implemented are Google OAuth 2.0, passport local, and passport JWT.

5.13.1.1 MongoDB/User Schema

To facilitate authentication, and later to identify users uniquely for creating their respective Roboflow projects and Cloudinary image folders, it was necessary to create a MongoDB database and coressponding User schema. The first version of the user schema used bcrypt's compareSync to compare the user's password with the hashed password present in the database, and required only the name, email, and password fields.

Google OAuth login was then added, and it was realised the data provided by a Google google login would not match the same data provided by a manual one. If a user logs in with Google, the email address was surprisingly not included in the response, instead the response contains a googelD. To allow for this, a googleID field was added to the schema, and the following check was run before saving the user to the database.

```
userSchema.pre('save', async function(next) {
    if (!this.googleId) {
        if (!this.password) {
            return next(new Error('Password is required.'));
        }
        if (!this.email) {
            return next(new Error('Email is required.'));
        }
})
```

The bcrypt password hash comparison function was also removed, as this is handled by passport-local-mongoose in the background, simply by providing it as a plugin to the user schema.

An issue arose in dealing with passport-local-mongoose's default requirement of a username field's presence in the schema. the documentation explains it is possible to override this value in the following way:

```
// this uses the passport local strategy, it's just abstracted by our use of passport-local-mongoose
.post("/login", passport.authenticate("local", { session: false }), login)
```

However, this did not seem to work. Several issues are currently open in the passport-local-mongoose github repository discussing this problem, which seems not to have been solved yet. As a workaround, it is now a requirement for the user to provide both a username and an email. The username field is not present in the user schema but is still required by passport-local-mongoose. When a user logs in with google, on the other hand, the googleID check has been updated as follows:

```
// Before saving a user, I want to find out if they've provided a googleID
// If so, they don't need to give a username and password
userSchema.pre("save", async function (next) {
  if (!this.googleId) {
    // passport-local-mongoose will pick up on the absence of a username, I don't need to check for that manually
    if (!this.email) {
      return next(
        new Error(
          "Email is required for normal login. Provide an email or login with Google."
        )
      );
    }
    next();
  } else {
    this.username = this.googleId;
  }

  if (!this.latest_version) {
    this.latest_version = 0;
    next();
  }
});
```

In this function, prior to saving a user, we check if the user has not provided a googleId. If this is the case, the user is logging in with email and password, so we confirm that they've provided an email. If not, we throw an error. In the case that the user *has* provided a googleID, we won't receive a username from the client side, so instead we manually set the username field equal to the user's google ID. The final conditional checks if the user does not yet have a 'latest_version' attribute and sets it to zero if not. This field refers to the dataset version generated for Roboflow after the user has uploaded their annotated images. This is explained in more detail in the Roboflow implementation chapter.

The final change necessary to facilitate both Google OAuth 2.0 and passport-local login and registration was to change the 'required' field for email and googleID to instead use 'sparse: true'. This means that a value does not need to be provided, but if it is, it must be unique. This is necessary since an email will not be provided if a user logs in with Google, and a googleId will not be provided if a user logs in with email and password.

5.13.1.2 Google OAuth

Of the three Passport strategies used, Google OAuth 2.0 required the most involved setup process. I first registered a cloud application on the Google Developer Console, which enables google to issue the application with a client ID and client secret, which are required by Passport. For security purposes, Google require OAuth-enabled applications to use authorised domains. To satisfy this requirement, the domain ‘www.raid-app.live’ was purchased and verified. A redirect was then setup from the Vercel-hosted frontend to the newly registered domain. Google also requires registered OAuth applications to provide a privacy policy. A generic privacy policy was modified to include contact details and data-collection information relevant to the RAID application, written in markdown format, and loaded into a newly created settings page within the frontend.

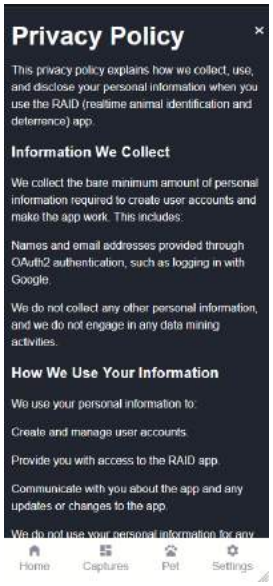


Figure 54 - RAID privacy policy

To load this markdown file, the markdown-to-jsx React library was used to parse the markdown into JSX dynamically. A series of CSS overrides were provided to style the JSX, which resulted in the above screen.

A list of authorised JavaScript origins and valid redirect URIs is also required.

Figure 55 – Authorised redirect URIs & JavaScript origins

5.13.1.3 What is OAuth?

OAuth is short for Open Authorisation, which is merely an approach to authorising users to login to our application. Common examples of services utilising OAuth are sites featuring 'Login with Facebook' or 'Login with Google' buttons. An advantage this provides to both the developer and the user is eliminating the need to fill in a login or registration form for our application. For example, as in RAID, our application provides a 'Login with Google' button which redirects the user to a Google OAuth consent screen describing the profile information the application is requesting. If the user allows the login, the user's Google credentials are sent back to our application, which we can then store in our database. The entire interaction described here is handled via PassportJS.

5.13.1.4 The general flow of OAuth authorisation, as implemented in RAID.

When the user presses the 'login with Google' button, we redirect them to a route on our application. In our case, the route is '/api/auth/google'. The job of this route is to direct the user to the Google OAuth consent screen. Once the user consents to providing their details to our application, they are sent to another custom endpoint within our application. In our case, this endpoint is '/api/auth/google/callback'.

Once the user reaches this endpoint, we determine whether the user has logged into our application before using a 'findOrCreate' method. If a user matching the provided login information is found, that user is retrieved from the database. Otherwise, the provided credentials are saved to our database. In either case, on successful login, a JSON web token is created, which encodes the user's credentials for storage on the client-side. In the case of Google OAuth login, specifically, the JWT is sent to the client and stored as a cookie. Finally, the user is redirected to the '/home' page on the client-side.

The original intention was to send a JWT to the client-side to persist in local storage, but the refresh caused by a successful Google OAuth authentication made it impossible to both redirect the user *and* send the token. Instead, the Google authentication strategy is unique in that it directly sets a cookie from the server, while the standard email and password login receives a cookie from the server and stores it through the client. In the email and password (or passport local) strategy, it would also be possible to store the JWT in local storage, but for the sake of consistency, I've opted to store the token as a cookie in either case.

The endpoint featured below demonstrates the aforementioned '/api/auth/google' route. When the user clicks 'Login with Google' on the client side, a new window is opened, pointing to this URL. This allows PassportJS to send the user to the Google OAuth consent screen.

```
.get(
  "/google",
  passport.authenticate("google", { scope: ["profile"] }),
  () => {
    console.log("received request for google auth");
  }
)
```

Figure 56 - The /api/auth/google endpoint

When Passport uses the Google authentication method to redirect the user to the consent screen, providing the user agrees, the Google strategy triggers the 'findOrCreate' method:

```
passport.use(
  new GoogleStrategy(
    {
      clientID: GOOGLE_CLIENT_ID,
      clientSecret: GOOGLE_CLIENT_SECRET,
      callbackURL: `/api/auth/google/callback`,
    },
    function (accessToken, refreshToken, profile, cb) {
      User.findOrCreate(
        {
          googleId: profile.id,
          name: profile.name.givenName,
          photo: profile.photos[0].value,
        },
        function (err, user) {
          return cb(err, user);
        }
      );
    }
  )
);
```

Figure 57 - The PassportJS Google Strategy

This method is received from the 'mongoose-findorcreate' NPM package, which has been added to the User Mongoose schema as a plugin. This plugin searches for a user in the MongoDB database based on the provided credentials. If one is found, the user's details are returned. Otherwise, a new user is created and stored in the database using the user's Google credentials. Regardless, the

callback function is triggered, which then encodes the user details as a JSON web token, sets this token as a cookie, and redirects the user to the client's home page.

5.13.1.5 Local

Most examples provided by PassportJS include 'serializing' and 'deserializing' users. This means that user data is kept in session storage on the server-side. However, I wanted RAID to feature both Google OAuth and Passport Local login functionality. Since RAID is a single-page React application, a user navigates between pages without the need to refresh the page, and so if user authentication data was stored on the server-side, the page would not refresh when the user navigates to a new route, meaning it would be difficult to confirm whether a user is authorised to access a resource or not, given that a session cookie is not sent with a request to view a new page.

Instead, RAID uses JSON web token for both Google OAuth and local authentication. However, as described previously, confirming a login with Google requires redirecting the user to the Google OAuth consent page, and so it was only possible for data to be sent back to the RAID client following this redirect by first setting the encoded JWT as a cookie, and then redirecting the user. When a user logs in with email and password, on the other hand, their data is still encoded as a JWT, but in this case it would also be possible to send the token to the client and store it in local storage. I've simply opted to store the JWT as a cookie in either case for consistency and simplicity.

```
// this uses the passport local strategy, it's just abstracted by our use of passport-local-mongoose
.post("/login", passport.authenticate("local", { session: false }), login)
```

Figure 58 - The passport local strategy route

When a user logs in using email/username and password, they send a request to the API's 'login' route, which in turn triggers the passport local authentication strategy. It would be possible to validate the user credentials by manually writing an authentication method, but I've instead opted to use the 'passport-local-mongoose' plugin to abstract this functionality away from my authentication implementation. Instead, once the user passes the passport local authentication middleware, they are directed to the 'login' method in the user controller, which encodes their data as a JWT, just as the Google OAuth strategy does. The local strategy, however, simply sends the JWT to the client-side, instead of storing it as a cookie. On the client-side, we use the 'js-cookie' package to set the JWT received from the server in the session cookies under the key 'x-auth-cookie', just as the Google OAuth strategy would on the server-side. Finally, the user is redirected to the home page on successful authentication.

Whether the user originally logged in using Google OAuth or email and password, their user data can then be retrieved from the client-side using the 'loadMe' method, which attaches the JWT token to the request headers to request the user's profile information from the server.

```
const attachTokenToHeaders = (token) => {
  const config = {
    headers: {
      "Content-type": "application/json",
    },
  };

  if (token) {
    config.headers["x-auth-token"] = token;
  }

  return config;
};

const loadMe = async (token) => {
  setLoading(true);
  try {
    const options = attachTokenToHeaders(token);
    const response = await axios.get(`${serverURL}/api/auth/me`, options);

    console.log("getMe success:", {
      user: response,
    });

    setUser(response.data.me);
    setLoading(false);
  } catch (err) {
    setError(err.response.data.message);

    console.log("getMe error:", err);

    setLoading(false);
  }
};
```

Figure 59 - The attachTokenToHeaders and loadMe methods

User credentials are then stored in the 'AuthContextProvider' context provider in the client, which wraps around the entire React application, enabling the retrieval of authentication details in any of the application's pages via the React 'useContext' hook.

5.13.2 Annotation Tool - Development

In order to upload annotated images to Roboflow, a custom solution was created by combining a custom fork of React BBox Annotator with React Dropzone Uploader. After first converting React BBox Annotator from its original TypeScript implementation to JavaScript, I combined an NPM installation of React Dropzone Uploader with a custom local fork of React BBox Annotator and customised React Dropzone Uploader's preview component styling to render uploaded images in a gallery format, with previous and next buttons for updating the selected image. The image preview is then wrapped by the BBox Annotator component to allow the user to draw bounding boxes on top of the selected image. Some structural improvements were made to the BBox Annotator code, by extracting the JSX for annotation entry items into a separate component which receives entry coordinates via its props, replacing verbose CSS styles with Tailwind CSS classes, changing the entry component to no longer update entry values on mouseOver and mouseLeave events, and to always show the option to close an annotation bounding box.

State was initially stored in the image preview component but was later lifted up to the top of the Upload page. A state variable for storing annotation data was initialised and its setter was passed into the BBox Annotator component in order to persist the state of each annotated image. When annotation data was received from the annotator, the uploaded files were mapped over and combined with their annotation data.

Every time a new image was uploaded, the handleChangeStatus method updated a 'files' state value, triggering a useEffect hook which would iterate over files checking for annotation data with a matching file name, combining annotations and files in preparation for upload to Roboflow.

Although annotation data was persisted in state, significant difficulty was encountered in continuing to display bounding boxes on-screen after they had been set.

Since BBox Annotator had been changed to set and receive annotation data from its props, the useEffect within the annotator which watched the annotation data in order to update it would cause an infinite loop. A file would be uploaded and passed into the annotator, the annotation data would be set and passed to the parent, which would in turn cause new annotation props to be passed to the annotator.

To solve this, I removed the onChange listener from BBox Annotator, and opted to handle all updates by passing state variables and their setters as props into the annotator, which would handle the updates itself, and pass them to the Upload parent component. A 'finalEntries' state value was added to store the final state of the annotated data when ready for upload, which was set using a 'Finished' button added to the annotator component. With annotation data received by the parent, files and their corresponding annotations could then be paired according to their matching names, ready for upload to Roboflow.

One issue overlooked when customising the BBox Annotator was having removed the 'multiplier' state value. This value proved crucial for transforming coordinate values according to the width of the image DOM element relative to the original width of the image. Using the forwardRef hook, the annotator would find the offsetWidth value for the DOM element containing the image and divide this value by the original width of the image, returning a multiplier value. The x, y, width, and height values of the bounding box were then multiplied by this value to transform them into an accurate annotation suitable for upload to Roboflow.

Another issue with which I experienced significant difficulty debugging was annotation coordinates becoming completely innacurate when a portrait orientation image was uploaded to Roboflow. To provide context to the issue, when testing the Roboflow CLI locally, I noticed that only images annotated in PASCAL VOC format seemed to be accepted by the upload API. As a result, it is necessary for coordinates received from React BBox Annotator to first have their object keys changed from left, top, width, and height to xmax, ymax, ymin, ymax, image width, and image height, where xmin and ymin correspond with left and top, and xmax and ymax are calculated by adding the left value to the width, and the top value to the height. Annotations are then ready to be written to an XML file in PASCAL VOC format.

An example of an annotation created on a portrait image within RAID is shown below. The bounding box coordinates appear accurate. However, after upload to Roboflow, the coordinates have shifted and are completely inaccurate.

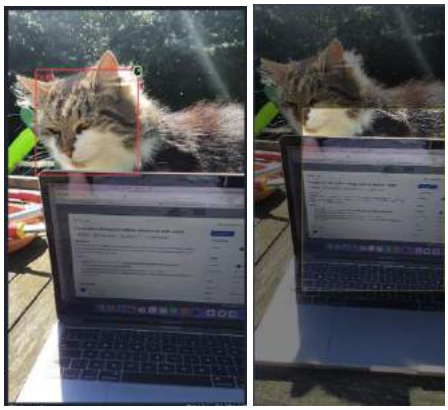


Figure 60 - An annotated portrait image before upload to Roboflow (L) and after upload (R)

For comparison, the issue did not occur when uploading a landscape image.

Figure 61 - A an annotated landscape image before (Top) and after (Bottom) upload to Roboflow

The issue was discovered after inspecting the value of the multiplier for both portrait and landscape images and noticing that it changed depending on the orientation of the first image to be uploaded. A landscape image would have a multiplier of 9.73, and portrait would use 4.60. If all the uploaded images were provided in the same format, either portrait or landscape, the issue would not occur. The annotations were inaccurate, however, when a mix of both portrait and landscape were uploaded. The solution was to *recalculate* the multiplier value after every image upload, so each image would have its values transformed by 9.73 or 4.60, depending on its orientation.

```
useEffect(() => {
  if (entries.some((entry) => entry !== undefined)) {
    setSubmissionEntries(
      entries.map((entry) => {
        const multiplier = entry.imgWidth / maxWidth;
        return {
          width: Math.round(entry.width * multiplier),
          height: Math.round(entry.height * multiplier),
          top: Math.round(entry.top * multiplier),
          left: Math.round(entry.left * multiplier),
          label: entry.label,
          fileName: entry.fileName,
          imgWidth: entry.imgWidth,
          imgHeight: entry.imgHeight,
        };
      })
    );
  }
}, [entries, multiplier]);
```

Figure 62 - Recalculating the multiplier value for every entry.

Having offered to open a pull request in the react-bbox-annotator repository, the author decided the issue could remain closed, as the problem is unique to my custom fork of the annotator, given that the original is only capable of receiving a single image at a time, unlike mine, which receives an entire gallery.

5.13.3 How the Annotator Works – A High-Level Overview

The Upload page begins with the Dropzone component, imported from react-dropzone-uploader. This component receives three child components through its props, InputComponent, LayoutComponent, and PreviewComponent. File uploads are handled using the handleChangeStatus function, passed to Dropzone's onChangeStatus listener.

```
<div style={{ height: "calc(100vh - 85px)" }}>
  <Dropzone
    onChangeStatus={handleChangeStatus}
    InputComponent={myInput}
    LayoutComponent={myLayout}
    PreviewComponent={MyPreview}
    accept="image/*"
  />
</div>;
```

Figure 63 - The React Dropzone Uploader receiving custom component props.

In the documentation for react-dropzone-uploader, examples will also include getUploadParams and onSubmit props. In our case it was also necessary to provide custom Preview and Layout components in order to display images as a gallery, instead of in a simple column of uploaded filenames. We also choose not to handle file uploads through React-Dropzone-Uploader, opting to omit an upload handler. The html5-file-selector library handles receiving files from the client's filesystem, resolving with the uploaded files, and passing the result to handleChangeStatus. This function will run every time we add a file, and files are then stored in state along with their metadata. The uploaded files, the index of the selected file for displaying a preview, and a state array of annotation objects, are all passed to the BBox Annotator through its props.

```
const MyPreview = ({ preview }) => {
  // We've changed bbox annotator to allow entries to be passed in
  // So we should try to find a relevant annotation entry for the image we're looking at right now
  return files[selected] ? (
    <div className={"m-auto h-full"}>
      <BBoxAnnotator
        url={files[selected].meta.previewUrl}
        entries={annotations}
        selected={selected}
        files={files}
        setEntries={setAnnotations}
        inputMethod="text"
      />
    </div>
  ) : (
    ""
  );
};
```

Figure 64 - The BBoxAnnotator component, receiving a preview url, files, selected index, and 'entries' for annotation coordinates.

In the BBox Annotator index file, there are several listeners for mousedown, mousemove, and mouseup events. When the mouseUp event fires, we call the updateRectangle function, passing in the mouse's x and y coordinates. This functon, in turn, calls the setPointer fucntion, using the crop function to adjust the x and y values first.

The crop function receives x and y values and returns a new object with x and y values representing the coordinates of the newly drawn bounding box. PageX and PageY are the mouse position, so we calculate these x and y values based on the position of the mouse on the image, and the position of the image DOM element relative to the page, hence subtracting the image element's offsetLeft and offsetTop values from pageX and pageY.

When the mouse first goes down, the pointer and offset values are set to the X and Y position of the mouse on-screen. When the mouse goes up, the offset and pointer values are set to the result of the crop function, giving us the top-left and bottom-right positions of the bounding box, the user has drawn.

The entire bounding box can then be drawn based on these values using the rectangle function:

```
const rectangle = () => {
  const x1 = offset && pointer ? Math.min(offset.x, pointer.x) : 0;
  const x2 = offset && pointer ? Math.max(offset.x, pointer.x) : 0;
  const y1 = offset && pointer ? Math.min(offset.y, pointer.y) : 0;
  const y2 = offset && pointer ? Math.max(offset.y, pointer.y) : 0;
  return {
    left: x1,
    top: y1,
    width: x2 - x1 + 1,
    height: y2 - y1 + 1,
  };
};
```

Figure 65 - The rectangle function used to return an object containing our bounding box coordinates.

Then, when the user has finished their annotation, the addEntry function runs, creating another annotation entry in the BBox Annotator's entries array:

```
const addEntry = (label) => {
  const newEntry = {
    ...rect,
    label,
    id: uuid(),
    showCloseButton: false,
  };

  setEntries((prevAnnotations) => [
    ...prevAnnotations,
    {
      ...newEntry,
      fileName: files[selected].file.name,
      imgWidth: files[selected].meta.width,
      imgHeight: files[selected].meta.height,
    },
  ]);

  setStatus("free");
  setPointer(null);
  setOffset(null);
};
```

Figure 66 – Adding an annotation object based on the rectangle's coordinate values, and file metadata.

The status, pointer, and offset values are then reset to their defaults for the next annotation to be drawn. Near the top of the BBox Annotator index, a useEffect hook watches the value of the entries state variable in its dependency array. If entries are found which are not undefined, the entries are mapped over, and a multiplier value is calculated by dividing the image's original width by the maxWidth, which is the width of the DOM element containing the image. With this value calculated, the submissionEntries array is set to the original entry coordinates with its coordinate values multiplied by the multiplier. This serves to ensure that the coordinate values are responsive. No matter the width of the viewport, the coordinate values drawn on the image are translated to be accurate to the image's original width, meaning the annotation coordinates are in the same place when uploaded to Roboflow as they are on-screen in the RAID upload page.

```
useEffect(() => {
  if (entries.some((entry) => entry !== undefined)) {
    setSubmissionEntries(
      entries.map((entry) => {
        const multiplier = entry.imgWidth / maxWidth;
        return {
          width: Math.round(entry.width * multiplier),
          height: Math.round(entry.height * multiplier),
          top: Math.round(entry.top * multiplier),
          left: Math.round(entry.left * multiplier),
          label: entry.label,
          fileName: entry.fileName,
          imgWidth: entry.imgWidth,
          imgHeight: entry.imgHeight,
        };
      })
    );
  }
}, [entries, multiplier]);
```

Figure 67 - Transforming annotation coordinates using the multiplier.

We notice that the submissionEntries do not contain any file data. One further step is required before we have a set of file and annotation data pairings suitable for upload to Roboflow. To facilitate this, we use two more useEffect hooks:

```
useEffect(async () => {
  if (files.length && finalEntries.length) {
    // return array of promises, wait for it to finish before running submitEntries()
    const promises = finalEntries.map(async (entry) => {
      // entry being the value returned from BBoxAnnotator with the appropriately transformed values...
      // need to pair them where file.name matches annotation.fileName
      // if the annotation's fileName attribute matches the name of one of our unnannotated files
      // when looking at an entry, i have access to the filename

      const relevantFile = files.find(
        (file) => file.file.name === entry.fileName
      );

      console.log("Relevant file: ", relevantFile);

      if (relevantFile) {
        const fileToBeAdded = {
          file: relevantFile.file, // exclude all the metadata
          annotation: entry,
        };

        setAnnotatedFiles((prevAnnotatedFiles) => [
          ...prevAnnotatedFiles,
          fileToBeAdded,
        ]);
      }
    });

    await Promise.all(promises);
  }
}, [finalEntries]);

useEffect(async () => {
  if (annotatedFiles.length) {
    await submitEntries()
      .then((res) => {
        console.log("Submit success: ", res);
        alert("Success");
      })
      .catch((e) => {
        console.log("Submit error: ", e);
        alert("Something went wrong");
      });
  }
}, [annotatedFiles]);
```

Figure 68 - A set of useEffect hooks used for finalising our annotation data.

The first useEffect is triggered based on a change to the finalEntries array. This array is empty until the user presses the 'Finished' button at the bottom of the screen, at which point, finalEntries is set to the same value as submissionEntries. It may seem redundant to use two different arrays to contain the same values but is done purposefully to avoid causing an infinite loop. Our useEffect hook checks that we have received file objects through the BBox Annotator's props, and that there are finalEntry values ready to be paired with these files. An array of promises is created by mapping over the finalEntry values. We then use the JavaScript find() function to search through the file objects received via the props for a file with a name matching the name of one of our finalEntries. When one is found, we create a new object with two keys; a 'file' key, containing the file object received from the Upload page, with its metadata removed, and an 'annotation' key containing the coordinate values received from the user's input, and transformed according to the multiplier value. This annotation data also contains some of the original metadata now removed from the file object itself, which will be necessary for naming and reading the files on the server-side. We add our new object to yet another state variable, annotatedFiles, and repeat this action for every file/annotation pair with matching names. We wait for this to finish for all our file/annotation pairs.

The second useEffect shown above then runs when annotatedFiles has been set and triggers the submitEntries function. This function simply creates a new FormData object and iterates over the annotatedFiles, appending the file object itself, and a stringified version of the annotation data. A POST request is then sent to the middleman server's 'uploadWithAnnotation' endpoint, passing in the user's token for authorisation.

5.13.4 Roboflow Functionality

The most arduous task of this sprint was implementing the logic for the entire flow of Roboflow functionality. Within the Roboflow UI, the steps necessary to produce a trained model are:

1. Create a unique project for the user.
2. Upload images, divided into train, test, and validate classes.
3. Annotated these images.
4. Generate a dataset version for these images, with any necessary augmentations or pre-processing applied, such as resizing or removing null values.
5. Export the dataset version to YOLOv5 PyTorch format.
6. Begin the training process.

Each of these steps and more needed to be replicated programmatically within the RAID application. Although Roboflow does have a REST API, its documentation is not extensive, and a complete list of the available endpoints and how to interact with them is not available. An SDK is available, but only in Python. A NodeJS SDK is currently under active development and its functionality is incomplete. To implement all the functionality needed by this application, many methods were translated from the Python SDK to JavaScript, others were modified from the existing Node CLI functions, and some written by monitoring network requests to find API endpoints when performing actions on the Roboflow website.

The first steps taken were to review the Python SDK Github repository and identify functions relevant to RAID. The generate_version, uploadImage, uploadAnnotation, and train methods were translated from the Roboflow-Python Project class to JavaScript and added to the middleman server. The methods used in Python are quite different, and mainly served as a reference for the REST endpoints available from the Roboflow API.

5.13.4.1 Generate Project & Version – Users

As explained previously in the PassportJS section, the creation of a MongoDB database was first necessary only for storing user data for authentication. However, it then became apparent that the unique IDs generated by MongoDB when a new user record is saved to the database provides the ideal unique identifier for individual user Roboflow projects. The user registration process was updated for both email and password registration, and Google OAuth login. Regardless of which authentication method the user chooses, the user is first registered, then logged in. On success, the getWorkSpaceDetails function runs, receiving the newly created user's unique MongoDB ID as its sole argument. This function makes a GET request to the Roboflow API for information about a workspace matching the user's unique ID. If one is found, the function resolves, and rejects otherwise. This function should never resolve, since the user has just been created, and so a Roboflow workspace matching their unique ID should not exist, but this step is included as a precaution.

If a Roboflow project is found which matches the user's ID, the login and registration process is finished. Otherwise, the createWorkSpace function is used to send a POST request to the same Roboflow endpoint used previously, this time passing in a project name and annotation name. The annotation name is used to identify the type of object the user is trying to train their model to recognise, but this information is just metadata and has no effect on model training, so 'raid' is hard coded as the annotation value in the request. Like most of the other Roboflow functions used by RAID, the endpoint for creating a new project was found within the Roboflow Python SDK and translated to JavaScript.

```
const createWorkSpace = async (name, annotation) => {
  return new Promise(async (resolve, reject) => {
    // annotation, what are you annotating, e.g. 'fido', 'dog', 'cat', whatever the user wants
    await axios
      .post(
        `https://api.roboflow.com/iadt/projects/?api_key=${process.env.ROBOFLOW_API_KEY}`,
        {
          name: name,
          type: "object-detection",
          license: "CC BY 4.0",
          annotation: annotation,
        }
      )
      .then((res) => {
        resolve(`Created new workspace: ${res.data.id}`);
      })
      .catch((e) => {
        reject(`Error creating workspace: ${e}`);
      });
  });
};
```

```
def create_project(self, project_name, project_type, project_license, annotation):
    data = {
        "name": project_name,
        "type": project_type,
        "license": project_license,
        "annotation": annotation,
    }

    r = requests.post(
        API_URL + "/" + self.url + "/projects?api_key=" + self.__api_key, json=data
    )

    r.raise_for_status()

    if "error" in r.json().keys():
        raise RuntimeError(r.json()["error"])

    return self.project(r.json()["id"].split("/")[-1])
```

Figure 69 - The JavaScript-translated version (L) of the Roboflow Python SDK's create_project method (R). Note 'workspace' in RAID's context is used differently that it is by Roboflow. Workspace refers to individual user projects, where there is only a single Roboflow project in our case, named IADT.

Finally, the User object is also used for saving the most recently generated dataset version after annotated images have been uploaded and converted to a dataset, ready for model training. In this way, the user's project name and latest dataset version are both stored in MongoDB, simplifying all other Roboflow-related operations. Once a user has been authenticated, we have access to the information required to do whatever is needed via the Roboflow API.

```
await generate_version(req.user._id).then(async (generated_version) => {
  console.log("Successfully generated version: ", generated_version);

  // now save the latest version as the user's 'latest_version' attribute
  await User.findByIdAndUpdate(
    req.user._id,
    {
      // roboflow returns a string, but schema expects a number
      latest_version: parseInt(generated_version.version),
    },
    {
      new: true,
    }
  );
});
```

Figure 70 - Saving the most recently generated version to the User object in MongoDB

5.13.4.2 Uploading Images & Annotations – the REST endpoint

The uploadImage and uploadAnnotation methods were combined to form saveImageAndAnnotation, the function called by the middleman server's 'uploadWithAnnotation' endpoint. The annotation functionality described previously calls this endpoint when all uploaded images have been annotated, and the user is ready to send them to Roboflow for model training. As mentioned earlier, Roboflow would be unable to receive the raw annotation coordinates from the Upload page on the client-side. Instead, annotation data needs to be converted to an appropriate object detection annotation format. There are many available annotation formats including COCO JSOn, Tensorflow TFRecord, and YOLOv8 PyTorch TXT. However, when experimenting with the functionality of the Roboflow Node CLI, I noticed that it only seems capable of receiving annotations in PASCAL VOC format, an issue which was highlighted to the Roboflow team. As a result, I chose to convert the annotation data to this format.

To determine how to convert the raw coordinates to PASCAL VOC, some data was manually exported to this format via the Roboflow UI. By comparing the PASCAL VOC file with the bounding box represented on-screen in Roboflow, it was clear that the xmax and ymax values could be calculated by adding the 'left' and 'width' values, and the 'top' and 'height' values, respectively. PASCAL VOC also allows for providing additional metadata about the annotation, such as an object's pose, whether the bounding box is truncated or occluded, or whether a prediction will be difficult to make. These are represented using the pose, truncated, difficult, and occluded keys. Object detection annotations can also be drawn in a 'segmented' format, whereby an 'outline' is drawn

around an object, instead of a simple rectangular bounding box. In our case, none of these values are relevant, so either be set to 0, or excluded from the XML file. For RAID's annotations, the necessary data includes the filename, which matches the name of the image being uploaded, the path to the relevant image, the width and height of the image, a depth value, which corresponds to the colour channels, in our case RGB, so 3, and finally the bounding box coordinates themselves, represented as xmin, xmax, ymin, and ymax.

Based on this information and the Roboflow endpoints identified via the Python SDK and Node CLI, it is now possible to receive images and annotations from the client-side and upload them to Roboflow.

When a user has finished their annotations on the client-side, each file is appended to a FormData object, and each corresponding annotation is stringified using JSON.stringify(). A POST request is made to the middleman server's 'uploadWithAnnotation' endpoint, using Multer, a middleware for parsing form data, to receive the files and annotations. As with the other Roboflow-related endpoints, the request passes through a second middleware, authenticateJWT, which uses passportJS to authenticate the user's request using the 'x-auth-token' included in the request headers.

Once the request is authenticated and the files and annotations are received, the server iterates through every file, and runs 'saveImageAndAnnotation' on each file and its corresponding annotation. In the loop, each file index will have the same index as its annotation.

```
const saveImageAndAnnotation = async (file, annotation) => {
  const imageBuffer = file.buffer;

  // Create a new file with a unique name in a directory called "uploads"
  const fileName = `${Date.now()}-${file.originalname}`;
  const imagePath = `../raid-middleman-server/uploads/${fileName}`;
  const annotationPath = imagePath.replace(/\.(jpeg|jpg|png)$/i, ".xml");

  try {
    // Write the image buffer to the new file
    await writeFile(imagePath, imageBuffer);

    const json = JSON.parse(annotation);
    // Write the annotation to a separate JSON file with the same name as the image file
    const annotationXML = JSONtoXML(
      fileName,
      imagePath,
      json.width, // xmin + width
      json.height, // ymin + height
      json.top, // ymin
      json.left, // = xmin
      json.label,
      json.imgWidth,
      json.imgHeight
    );

    await writeFile(annotationPath, annotationXML);

    // Send a response indicating that the file was successfully uploaded and saved
    console.Log("Image and annotation saved");
  } catch (err) {
    console.error(err);
  }
};
```

Figure 71 - The saveImageAndAnnotation function

In this function, the buffer data is retrieved from the file, and a unique filename is generated based on a combination of the current date and time, and the file's original name. '../raid-middleman-server/uploads' will be the directory where the file is saved. An 'annotationPath' is then generated by using a Regex pattern to replace the image's file extension with a '.xml' extension. A promisified version of the fs.writeFile method is used to save the image buffer to the uploads directory. Then, the annotation data is parsed back into JSON, and the fileName, imagePath, and the entire annotation data object (width, height, top, left, label, imgWidth, imgHeight) is passed into the JSONtoXML function. As described, this function transforms the coordinate data into a valid PASCAL

VOC annotation, where xmin corresponds with 'left', ymin corresponds with 'top', and xmax and ymax are the sum of the left and width, and top and height values, respectively. This data is passed into a string in the format of a PASCAL VOC annotation, and the the xmlbuilder2 library is then used to convert the data to a valid XML file.

```
const JSONtoXML = (fileName,filePath,width,height,top,left,label,imageWidth,imageHeight) => {

  let xmin = left;
  let ymin = top;
  let xmax = left + width;
  let ymax = top + height;

  const xmlStr = `<annotation>
            <folder></folder>
            <filename>${fileName}</filename>
            <path>${filePath}</path>
            <size>
                <width>${imageWidth}</width>
                <height>${imageHeight}</height>
                <depth>3</depth>
            </size>
            <segmented>0</segmented>
            <object>
                <name>${label}</name>
                <bndbox>
                    <xmin>${xmin}</xmin>
                    <xmax>${xmax}</xmax>
                    <ymin>${ymin}</ymin>
                    <ymax>${ymax}</ymax>
                </bndbox>
            </object>
    </annotation>`;

  const doc = create(xmlStr);

  const xml = doc.end({ prettyPrint: true });

  return xml;
};
```

Figure 72 - The JSONtoXML function

With the raw coordinate values successfully converted to PASCAL VOC, the XML file returned from JSONtoXML is saved. Each time saveImageAndAnnotation completes, the return value is added to a 'promises' array, and the entire array is awaited using Promise.all(). When all promises have completed successfully, fs is used to read the entire directory containing the images and annotation XML files. The files are sorted in ascending order, so the images come first. Next, the images are split into train, test, and validation tests, using the Roboflow default values of 70%, 20%, and 10%. The files are iterated over two at a time to process both an image and its corresponding annotation file.

A ‘numUploaded’ counter is incremented after every successful upload, and this value is divided by the half the number of files and multiplied by one hundred. In this way, we can determine the percentage of files whose upload has been completed. While the percentage value is less than 70%, images are assigned to the ‘train’ set. When the percentage is between 70% and 90%, images are assigned to the ‘validate’ set, and the final 10% are assigned to the test set. The set each image is assigned to is passed into the image upload request made to Roboflow.

```
await Promise.all(promises).then(() => {
  // now upload all the image and annotation files to roboflow
  const directoryPath = "../raid-middleman-server/uploads";

  fs.readdir(directoryPath, async function (err, files) {
    if (err) {
      return console.log("Unable to scan directory: " + err);
    }

    // Sort files in ascending order so that image files come first
    files = files.sort();

    const trainSplit = 0.7;
    const validSplit = 0.2;

    let numUploaded = 0;

    let split;

    // find out how far through we are
    let percentUploaded;

    // Loop through the files two at a time, image file and its corresponding annotation
    for (let i = 0; i < files.length; i += 2) {
      percentUploaded = (numUploaded / (files.length / 2)) * 100;

      const imageFilePath = path.join(directoryPath, files[i]);
      const annotationFilePath = path.join(directoryPath, files[i + 1]);

      if (percentUploaded < trainSplit * 100) {
        split = "train";
      } else if (percentUploaded < (trainSplit + validSplit) * 100) {
        split = "valid";
      } else {
        split = "test";
      }

      await uploadWithAnnotation(
        imageFilePath,
        annotationFilePath,
        req.user._id, // user's unique _id is also used as their project name (users only need one)
        process.env.ROBOFLOW_API_KEY,
        {
          split,
        }
      );
    }
  });
});
```

Figure 73 - A shortened version of the uploadWithAnnotation function

Functionality for splitting uploaded images into separate sets by percentage values does not exist in the Python SDK or the Roboflow Node CLI. In discussion of an issue with a Roboflow team member, they were impressed by this code, and requested that a pull request be opened to their API examples repository showcasing this function.

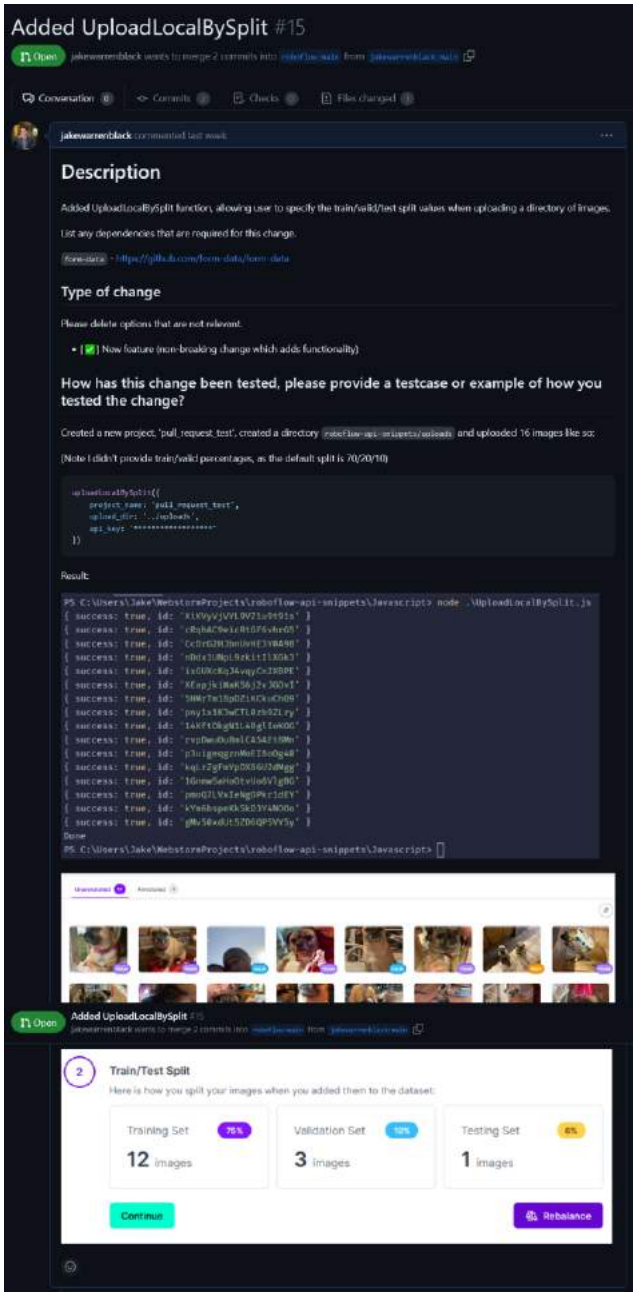


Figure 74 - The pull request opened to the Roboflow API examples repository.

The examples in the pull request above demonstrate the result of using this function with 16 images. Again, the default split of 70/20/10 is used, and the resulting split is the closest possible approximation of these values, at 75%, 19%, and 6%.

The next step of the upload process moves to the Roboflow utility functions, where uploadWithAnnotation is called on each image and its corresponding annotation. The function receives the image file name, the annotation file name, a project URL, api key, and an 'extraOptions' object, to which we pass the aforementioned split value.

The function first calls another Roboflow utility function, uploadImage, which simply creates a new FormData object, reads the file using fs.createReadStream(), and sends a POST request to the Roboflow upload endpoint, passing the FormData object containing the file name, the file itself, and the split values.

```
const uploadWithAnnotation = async (
  fileName,
  annotationFilename,
  projectUrl,
  apiKey,
  extraOptions
) => {
  const uploadPromise = uploadImage(fileName, projectUrl, apiKey, extraOptions);

  // uploadAnnotation requires imageId from uploadImage, so I have to wait for that to complete first
  const annotationPromise = uploadPromise.then(async (uploadResult) => {
    const imageId = uploadResult.id;

    if (annotationFilename.includes("[filename]")) {
      annotationFilename = annotationFilename.replace(
        "[filename]",
        path.parse(fileName).name
      );
    }

    if (fs.existsSync(annotationFilename)) {
      return await uploadAnnotation(
        imageId,
        annotationFilename,
        projectUrl,
        apiKey
      )
        .then((annotationResult) => {
          return { uploadResult, annotationResult };
        })
        .catch((e) => {
          console.log(e);
        });
    } else {
      return { uploadResult };
    }
  });

  // resolve if both uploadImage and uploadAnnotation succeed, but reject if either fail
  return Promise.all([uploadPromise, annotationPromise])
    .then(([uploadResult, { annotationResult } = {}]) => {
      return { uploadResult, annotationResult };
    })
    .catch((error) => {
      console.error(`  Error uploading ${fileName}: `, error);
      throw error;
    });
};
```

Figure 75 - uploadWithAnnotation - A Roboflow utility function

The uploadWithAnnotation function waits for uploadImage to complete, and then passes the result of the image upload to 'annotationPromise'. The image ID generated by Roboflow when the upload completed is passed through to the annotation upload request, in order to match the local annotation file with the correct image file on Roboflow. Similarly to the image upload function, the uploadAnnotation function is then called, which uses fs.readFileSync to read the XML file, and passes this data into a POST request to the Roboflow annotate endpoint. The uploadWithAnnotation function uses Promise.all() to await success from both the image upload and the annotation upload, and returns the result of the promise. This process runs for consecutively for every image and annotation pair saved to the uploads directory. When the upload function resolves, fs is used once again to delete all uploaded file from the uploads directory.

5.13.4.3 Generate Version & Export

The next step necessary in the process of training a Roboflow model is generating a version. When images have been uploaded and annotated to Roboflow, a version must be generated in order to apply pre-processing and data augmentation steps, if necessary. Possible pre-processing and augmentation steps include blurring, adjusting brightness, cropping images, simulating occlusion by

blacking out random portions of an image, and many other possibilities. Pre-processing can correct aspects of the image, such as auto-orienting, adjusting contrast, resizing, or remapping class names to new values if necessary. In RAID's case, default values have been provided for consistency across all generated models. Images are auto oriented, 100% of unannotated images are removed from the dataset, and the images are resized to 640x640 pixels. The resize is included to allow the model to train faster, as larger images naturally take more time to process. The version generation function simply receives a project name as its sole argument and makes a POST request to the /generate Roboflow API endpoint, generating a new dataset version. This endpoint is called every time a dataset is annotated and uploaded, to allow a model to be trained.

```
const generate_version = async (
  project_name,
  settings = {
    augmentation: {},
    preprocessing: {
      "auto-orient": true,
      "filter-null": {
        percent: 100,
      },
      resize: {
        width: 640,
        height: 640,
        format: "Stretch to",
      },
    },
  }
) => {
  return new Promise(async (resolve, reject) => {
    if (
      !Object.keys(settings).includes("augmentation") ||
      !Object.keys(settings).includes("preprocessing")
    ) {
      reject(
        new Error("augmentation and preprocessing keys are required to generate. If none are desired specify empty object associated with that key.")
      );
    }

    await axios
      .post(
        `https://api.roboflow.com/iadt/${project_name}/generate?api_key=${process.env.ROBOFLOW_API_KEY}`,
        {
          ...settings,
        }
      )
      .then((res) => {
        if (res.status === 200) {
          resolve(res.data);
        }
      })
      .catch((e) => {
        reject(
          new Error(
            `Error when requesting to generate a new version for project: ${e}`
          )
        );
      });
  });
};
```

When the version generation completes, the User object is updated in the database with the result of the /generate API call, changing the user's 'latest_version' attribute to the result. Given that the user's unique MongoDB ID is also used as the name of their Roboflow project, a model can then be trained simply by making a request to the /train endpoint, passing the user's ID and latest_version attribute.

Internally, though it is not documented, the Roboflow train endpoint seems to generate a YOLOv5 model, and therefore requires a dataset to be exported in YOLOv5 PyTorch format for training. To facilitate this, the final function called before model training begins is the exportForTraining function.

```
const exportForTraining = async (project, version) => {
  return new Promise(async (resolve, reject) => {
    // it looks like roboflow train expects data in yolov5pytorch format internally, so convert for it.
    try {
      const res = await axios.get(
        `https://api.roboflow.com/iadt/${project}/${version}/yolov5pytorch?api_key=${process.env.ROBOFLOW_API_KEY}`
      );

      if (res.status === 202) {
        // we get 202 when generation has begun. it'll take a few seconds, depending on the dataset size.

        await sleep(2000); // wait 2 secs before trying again. we're waiting for 200 OK.

        const result = await exportForTraining(project, version);
        resolve(result);
      } else if (res.status === 200) {
        resolve(res.data.version.exports[0]);
      }
    } catch (e) {
      reject(e);
    }
  });
};
```

Figure 76 - The exportForTraining function, which calls itself recursively until success is reached.

To export the dataset for training, we simply make a GET request to workspace/project/version with the desired model type included as a query parameter. This endpoint will return a status code of 202 Accepted when the export begins, but will not return any data, and is not complete until it returns a status of 200 OK. To check for success, the exportForTraining makes its request and checks the response code. If the code is 202, it will wait for two seconds, and then call itself recursively. This will repeat until a status code of 200 is returned from the API. At this point, RAID has successfully generated a unique user project, uploaded and split images into train, test, and validation sets, and applied each annotation to its corresponding image. When the image upload and annotation process is complete, a dataset version is generated with the appropriate pre-processing steps applied, and the dataset is exported in YOLOv5 PyTorch format, and the user's data is ready for model generation to recognise their specific pet.

5.13.4.4 Troubleshooting with the Roboflow team

Upon reaching the final stage of the Roboflow functionality, the training function, a very unusual and difficult to debug issue was encountered. When images were uploaded and annotated via the RAID app, and the model generation begun, the model produced on Roboflow referred to non-existent class labels. In this case, a new model was trained as a test, based on a dataset of 26 images of a cat, using a single class label, 'Holly.' However, on Roboflow's side, the model's class labels were 'dime, nickel, penny, quarter.' Assuming the issue was occurring on the Roboflow servers, an issue was raised on their forum. The code below was used for triggering the Roboflow training process, and while it is incomplete, it worked for beginning training without issue.

```
const train = (project, version) => {
  return new Promise(async (resolve, reject) => {
    // first check if already generating a model.
    // if so, do not let the user generate another model - will cause havoc in terms of model accuracy,
    // since it trains the same model twice.
    await getTrainingStatus(project, version)
      .then(async (trainingDetails) => {
        if (Object.keys(trainingDetails.version.model).length) {
          // this dataset version already has a model. there can be only 1 per version.
          reject("Please generate a new version to train a model");
        }

        if (!trainingDetails?.version?.generating) {
          // this call starts the training process
          await axios
            .post(
              `https://api.roboflow.com/iadt/${project}/${version}/train?api_key=${process.env.ROBOFLOW_API_KEY}`
            )
            .then(async (res) => {
              let status = "training";

              setInterval(async () => {
                if (status === "training") {
                  // this call checks on the status of the training process
                  await getTrainingStatus(project, version)
                    .then(async (trainingDetails) => {
                      // if res.version.generating = false, training has stopped (I think)
                      if (!trainingDetails?.version?.generating) {
                        // change training status to break out of the interval and resolve
                        status = "done";

                        // maybe use websocket here? keep updating the user on the training status
                      }
                    })
                    .catch((e) => {
                      reject("Failed to check training status: ", e);
                    });
                } else {
                  resolve("Training has finished");
                }
              }, 5000);
            })
            .catch((e) => {
              reject("Failed to begin training process: ", e);
            });
        } else {
          reject(
            "A version is already generating. Please wait for it to finish before generating another version."
          );
        }
      })
      .catch((e) => {
        reject("Error while determining training status: ", e);
      });
  });
};
```

Figure 77 The function used to trigger the Roboflow model training process.

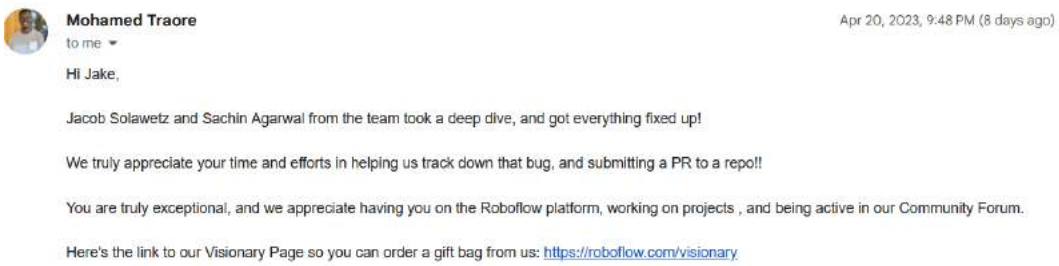
This code will first make an API call using getTraining Status to determine whether a model for this specific project and dataset version has already begun training. If a model is not already training, the training process is initialised using a POST request to the 'train' endpoint on Roboflow. A setInterval loop then runs every five seconds, running the getTraining Status function on every iteration. The interval continues until the model training has completed. A response from the founder of Roboflow first suggested sharing the project with the Roboflow support team and generating a new ‘health check’ on the dataset, to determine that the class values used were as expected.

At this point in the project’s development, uploaded images were not split into train, test, and validate tests, and merely all uploaded to the train set. I suggested to Roboflow support that the issue could be caused by this, and Mohamed, an engineer at Roboflow, responded with an explanation of how the split functionality is implemented in the Roboflow Python SDK. After writing the split functionality for the RAID upload method, which is detailed above, this proved to be unrelated to the issue I was experiencing. However, Mohamed was impressed with the code I had provided, and then requested that a pull request be opened to Roboflow’s API snippets GitHub repository, demonstrating how to perform data splitting in JavaScript.

Another suggestion was made after discovering that when a request is made via the API for information on a project generated through the Roboflow website, the ‘project.classes’ object in the response lists each class included in a dataset, and the number of images corresponding to that class. When the same request is made for a dataset generated programmatically, however, the response does not list the number of classes. Unfortunately, this again proved not to be the cause of the issue.

Mohamed felt the tests I had run in attempting to determine the cause of the issue were extremely helpful and added the results of these tests to an internal bug report. I then emailed the Roboflow support team with all the images and annotations used for the Roboflow project on which this issue occurred and provided two GitHub gists containing the RAID API methods relating to Roboflow, and all the Roboflow utility functions used by these API endpoints, respectively. The Roboflow team work remotely but are based mainly in the United States. The time difference meant that my troubleshooting session with Mohamed continued for around 8 hours, until 7:30AM. The following morning, the Roboflow engineering team isolated the issue, and prepared a fix.

After the issue had been fixed, I received an email from Mohamed, explaining that Jacob Solawetz and Sachin Agarwal from the Roboflow engineering team and taken a ‘deep dive’ into the API codebase, and had solved the issue. Jacob Solawetz commented that it was ‘pretty insane’ that I had found the ‘train’ API endpoint to begin with and noted that Roboflow were not yet ready for it to be used outside their Python SDK. As thanks for assisting the team in resolving the issue, Roboflow sent a merchandise gift bag.



Mohamed Traore Apr 20, 2023, 9:48 PM (8 days ago)

to me

Hi Jake,

Jacob Solawetz and Sachin Agarwal from the team took a deep dive, and got everything fixed up!

We truly appreciate your time and efforts in helping us track down that bug, and submitting a PR to a repo!!

You are truly exceptional, and we appreciate having you on the Roboflow platform, working on projects , and being active in our Community Forum.

Here's the link to our Visionary Page so you can order a gift bag from us: https://roboflow.com/visionary

Figure 78 - Email from Mohamed, an engineer at Roboflow, thanking me for my help.

After running the training function again via the RAID UI, the following model was trained based on the same dataset which caused the non-existent class name issue. The model can then be represented in the application using its dataset split values and accuracy statistics by making a request to the server's 'getVersionInfo' endpoint.

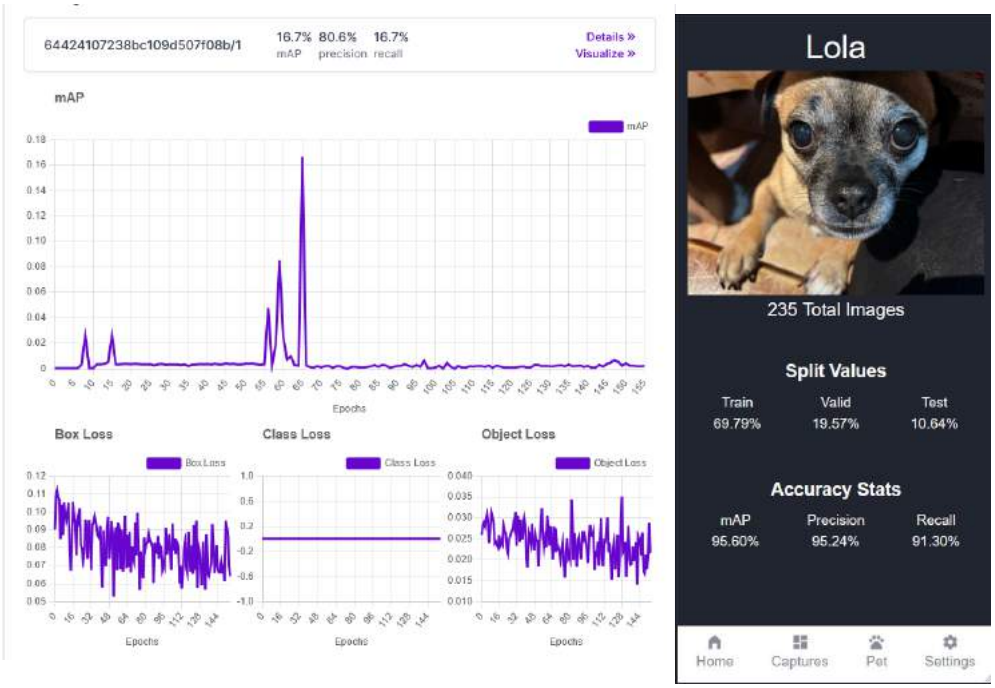


Figure 79 - A model trained on Roboflow via the RAID UI (L) and represented in RAID (R)

The Roboflow team expressed particular interest in this project and have requested that I provide them with a video demonstrating what the project does, as well as a blog post to be featured on the Roboflow newsletter.

5.13.4.5 Auto-Annotate

In discussion with the Roboflow support team via the website's live chat, I received a feature suggestion from Joseph Nelson, the CEO at Roboflow. Joseph suggested the architecture of the app be changed to reduce the burden on the end-user by automatically annotating provided images.

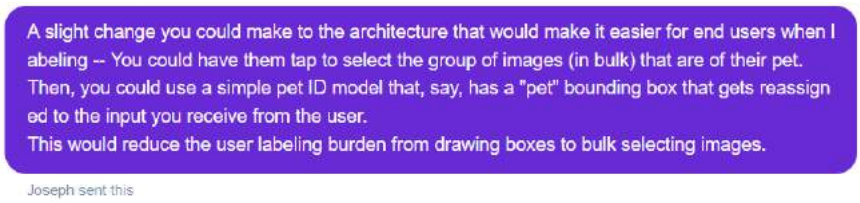


Figure 80 - Message from Joseph Nelson making a feature suggestion.

Considering that the Raspberry Pi used by RAID already has its own custom-trained YOLOv4-Tiny model capable of recognising animals, I felt it would be possible to implement this feature and made an attempt at doing so. My interpretation of Joseph's suggestion was to allow the user to upload images of their pet, just as they do now, but pass these images into the YOLOv4-Tiny model for automatic annotation, removing from the user the burden of needing to annotate each image manually.

The implementation decided on was as follows. The handleChangeStatus function in the Upload page was used to receive uploaded files, then send them to the middleman server. The YOLOv4-Tiny model was copied to the middleman server, and opencv4nodejs was installed. Every image received from the cleitn was passed to through the object detection model, and the prediction coordinates returned to the client for display in the annotator component. The concept was for the user to review the automatic annotations, and be provided with the opportunity to change them, or simply submit them if they are satisfied with their accuracy.

An 'annotate_controller' file was added to the middleman server, which would work similarly to the 'detect' file from the Raspberry Pi. The 'auto-annotate' endpoint would call the 'annotate' function in this controller, and receive the files passed from a POST request on the client-side. Iterating over the files, every file would be saved locally, and the promise returned from the save image operation is returned to a 'promise' array, just as when images are uploaded for manual annotation.

```
const annotate = async (req, res) => {
  const files = req.files["files[]"];

  const promises = [];

  let annotations = [];

  // first we save all the files to an /annotations directory
  for (let i = 0; i < files.length; i++) {
    const file = files[i];
    const promise = saveImage(file);
    promises.push(promise);
  }

  // wait for the files to finish saving
  await Promise.all(promises).then(() => {
    const directoryPath = "../raid-middleman-server/annotations";

    //
    fs.readdir(directoryPath, async function (err, files) {
      if (err) {
        return console.log("Unable to scan directory: " + err);
      }

      for (let i = 0; i < files.length; i++) {
        // run files through the animal detection model
        // do I need to resize and convert to base64? Not sure
        await classifyImg(files[i]).then((res) => {
          annotations.push({
            fileName: files[i].name,
          });
        });
      }

      console.log(
        "Annotations finished. Now emptying the annotations directory."
      );

      // when finished, empty the /uploads directory
      fs.readdirSync(directoryPath).forEach((file) => {
        console.log(`Removing file ${file}`);
        fs.unlinkSync(`${directoryPath}/${file}`);
      });
    });
  });
};
```

Figure 81 - The annotate function, which is called in response to a request to the auto-annotate endpoint.

When all images are saved, they are read and passed through the object detector via the classifyImg function. The returned coordinates are then passed to an annotations array, and finally sent back to the client. This version of the method, when compared with the one used in the Raspberry Pi, has been promisified in order to allow the REST endpoint to await a response from it. It is also no longer

necessary for OpenCV to draw an annotation and label on the image, and instead this function's only purpose is to return the prediction's coordinates.

```
const classifyImg = async (img) => {
  return new Promise(async (resolve, reject) => {
    // object detection model works with 416 x 416 images
    const size = new cv.Size(416, 416);
    const vec3 = new cv.Vec(0, 0, 0);
    const [imgHeight, imgWidth] = img.sizes;

    // network accepts blobs as input
    const inputBlob = cv.blobFromImage(img, 1 / 255.0, size, vec3, true, false);
    net.setInput(inputBlob);

    // forward pass input through entire network
    const layerOutputs = net.forward(layerNames);

    // object with x, y, width, height
    let box;

    layerOutputs.forEach((mat) => {
      const output = mat.getDataAsArray();
      output.forEach((detection) => {
        const scores = detection.slice(5);
        const classId = scores.indexOf(Math.max(...scores));
        const confidence = scores[classId];

        if (confidence > minConfidence) {
          const box = detection.slice(0, 4);

          const centerX = parseInt(box[0] * imgWidth);
          const centerY = parseInt(box[1] * imgHeight);
          const width = parseInt(box[2] * imgWidth);
          const height = parseInt(box[3] * imgHeight);

          const x = parseInt(centerX - width / 2);
          const y = parseInt(centerY - height / 2);

          // here's what i need to return for my coordinates
          // no need to return an image from this like the pi does
          //boxes.push(new cv.Rect(x, y, width, height));
          box = { x, y, width, height };
        }
      });
    });

    if (box) {
      resolve(box);
    } else {
      reject("No coordinates generated for image");
    }
  });
};
```

Figure 82 - The classifyImg function, used for generating coordinates from an image.

An issue arose after returning annotations generated by OpenCV and rendering them in React BBox Annotator. Annotations would appear slightly innacurate. Furthermore, the YOLOv4-Tiny model often seemed incapable of making a detection on an image obviously containing an animal.

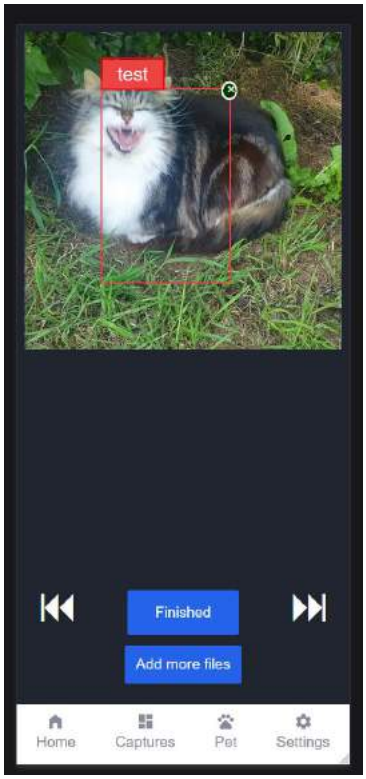


Figure 83 - An example of an innacurate annotation produced by OpenCV/BBox Annotator

The solution to recalculating the values to appear correctly when displayed by React BBox Annotator was to translate the width and height values as follows:

width: (annotatedFile.width - annotatedFile.x)

height: (annotatedFile.height - annotatedFile.y)

This was determined after manually drawing the annotation coordinates generated by OpenCV in Adobe Illustrator, and noticing they were accurate. By comparing the OpenCV annotation with the bounding box displayed by BBox Annotator, it was apparent that an accurate annotation could be generated using this translation. Furthermore, it was necessary to recalculate the multiplier value used by BBox Annotator depending on whether an image was received via a manual annotation, or from OpenCV. In the latter case, the multiplier needed to be calculated as maxWidth divided by width, whereas a manual annotation would require width divided by maxWidth. This was controlled by making the following modification to the multiplier calculation in BBox Annotator, where each automatically annotated image would be passed a key/value pair of 'auto: true'.

The issue of the model being incapable of making predictions was solved by resizing the provided images to 416x416 pixels. This was necessary because the YOLOv4-Tiny model was trained on 416x416 images, and so was only capable of making accurate predictions on images of this size.

When dealing with automatic annotations, it was also no longer possible for BBox Annotator to receive metadata about the files in order to display a preview URL for each image, as the original image was resized on the server-side to 416x416, rendering any annotation drawn on that image completely innacurate. To solve this, each resized image would be transformed to a base64 string, and sent back to the client-side for display. Finally, for saving the image again on the client-side, in preparation for uploading an annotation to Roboflow, it was necessary to convert this base64 string back int a blob. This was achieved using the b64toBlob function:

```
const b64toBlob = (b64Data, contentType = "", sliceSize = 512) => {
  const byteCharacters = atob(b64Data);
  const byteArrays = [];

  for (let offset = 0; offset < byteCharacters.Length; offset += sliceSize) {
    const slice = byteCharacters.slice(offset, offset + sliceSize);
    const byteNumbers = new Array(slice.Length);

    for (let i = 0; i < slice.Length; i++) {
      byteNumbers[i] = slice.charCodeAt(i);
    }

    const byteArray = new Uint8Array(byteNumbers);
    byteArrays.push(byteArray);
  }

  return new Blob(byteArrays, { type: contentType });
};
```

The result of this was a successful implementation of the feature suggestion made by Joseph Nelson. However, not all annotations generated automatically would be accurate. When testing this feature, 40 images were passed into the auto-annotator, of which only 18 produced automatic annotations. Among these, not all bounding boxes would be accurate, and some others would be wildly out of place. While many of these annotations would be perfectly accurate, I felt it would be a hindrance to the user to need to manually check every automatically annotated image for accuracy.

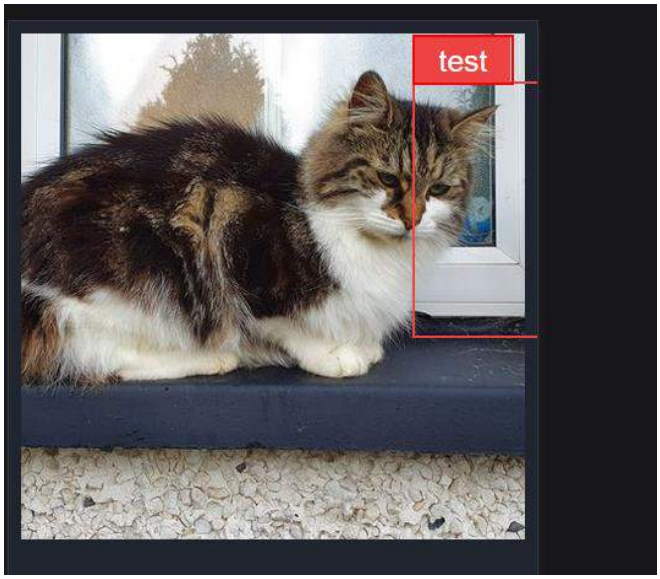


Figure 84 - An example of an innacurate automatic annotation

As a result, this feature was removed from the main application and into its own feature branch. If a more accurate model than the existing YOLOv4-Tiny animal detector were created, a feature such as this could be extremely helpful to the end-user, but in this case, the time required for training such a model is not available.

5.13.5 Cloudinary Image Capture

As described previously, one of the application's requirements is to automatically take screenshots of detection events, and send the image to the middleman server to be uploaded to a cloud storage

service, allowing the user to view the detected objects categorised by their class name on the client-side. A user should also be able to use the 'screenshot' button on the client-side to capture a screenshot manually. On the middleman server, images are already received in a constant stream for display on the client as a video feed, so regarding the manual screenshot functionality, it was possible to allow manual screenshots simply by maintaining a reference to the most recently received image at all times.

A 'latestImage' variable was added to the middleman server, which is updated after every SocketIO event receiving an image from the Raspberry Pi. The image received from the Raspberry Pi is already encoded to a base64 string, so it was trivial to add an endpoint to the middleman server to allow the client to send a request asking the server to save the most recently received image to Cloudinary, a cloud image storage and content delivery service.

The Cloudinary NodeJS SDK was installed on the middleman server, and two new endpoints were added to the server, 'api/screenshot' and 'api/getimages'. Since it is possible for a user to take a screenshot manually, or for a screenshot to be triggered by the Raspberry Pi based on a detection event, a request to the screenshot endpoint can optionally provide a 'folderName' query parameter. If no folderName is provided, the folderName is set to 'manual', implying the user has taken the screenshot manually. When an image is saved to Cloudinary, it uses a path in format of 'userID/folderName'.

This means that every folder will be unique to each user but can also contain sub-folders named based on the type of detection event which occurred. For example, the Raspberry Pi server might detect a dog, and save it to 'userID/dog', or the user might take a manual screenshot, which will be saved to 'userID/manual'. The 'api/getimages' endpoint then receives a folder name as a query parameter, but also requires authentication from the user.

This means that the user's unique ID will be available to the request, even without being explicitly provided. Images may then be retrieved from Cloudinary according to a user's ID, and the type of object they are requesting. In practice, these endpoints are implemented in the following way on the middleman server. To prevent the same object detection event from causing multiple screenshot uploads, it was also necessary to add an 'uploadInProgress' flag. The uploadInProgress variable is only set to false when an upload has completed, and a detection event cannot trigger a screenshot upload until this value is false.

```
let latestImage;

app.use("/api/pair", require("./routes/pair"));
app.use("/api/buzz", require("./routes/buzz"));
app.use("/api/auth", require("./routes/auth"));
app.use("/api/roboflow", require("./routes/roboflow"));

let uploadInProgress;

const takeScreenshot = async (
  image,
  folderName,
  userID,
  type = "automatic"
) => {
  // Don't allow taking more than 1 screenshot in the space of 30 seconds (to prevent spamming by automatic captures)

  // Users can manually take as many as they want to
  if (type === "automatic") {
    if (
      takeScreenshot.lastTime &&
      Date.now() - takeScreenshot.lastTime < 30000
    ) {
      return Promise.reject(new Error("Cannot take another screenshot yet"));
    }
  }

  return new Promise(async (resolve, reject) => {
    if (uploadInProgress) {
      reject("An upload is already in progress");
    }

    uploadInProgress = true;

    // an automatic screenshot will provide a folder name based on the type of animal detected, .e.g /dog
    // but if user just presses 'screenshot' it could be of anything, so save it to a generic 'manual' screenshots folder
    const folder = folderName || "manual";

    try {
      const result = await cloudinary.uploader.upload(image, {
        folder: `${userID}/${folder}`,
      }); // creates the folder if it doesn't already exist. so e.g. user1/dogs, user1/manual
      resolve(result);
    } catch (e) {
      reject(e);
    } finally {
      uploadInProgress = false;
    }

    takeScreenshot.lastTime = Date.now();
  });
};
```

Figure 85 - The takeScreenshot function used on the middleman server for controlling both manual and automatic screenshot uploads.

A manual screenshot request from the client-side simply makes an authenticated request with no additional parameters required. The user does not need to provide a folder name. As described, in the case of a manual screenshot, the user simply makes an authenticated request, and their unique ID is used as their folder name. When no folderName query parameter is provided, 'manual' is used as the sub-folder to which the screenshot is saved.

For automatic screenshots, the logic is slightly more involved. When a user makes a request from the client-side, they are authenticated by providing an x-auth-token. However, the Raspberry Pi does not have a user token, yet still needs to make requests to the server to save screenshots. This was further complicated by the fact that only a single Raspberry Pi is available during development. In reality, if RAID were a real product, each user would have their own Raspberry Pi, which would make it possible to assign a unique ID to each device on a per-user basis, saving screenshots to a user's unique Cloudinary folder. To work around this issue, it was necessary to add a check to the authenticate-socketio middleware to allow authentication to be skipped when a request is received from the Raspberry Pi.

```
const authenticateSocketIO = (socket, next) => {
    // Check for the X-Is-Pi header
    if (socket.handshake.headers["x-is-pi"] === "true") {
      // Skip authentication for the Raspberry Pi
      socket.isPi = true;
      return next();
    }

    const token = socket.handshake.auth.token;
    if (!token) {
      return next(new Error("Authentication error: missing token"));
    }

    jwt.verify(token, process.env.APP_KEY, {}, (err, decoded) => {
      if (err) {
        return next(new Error("Authentication error: invalid token"));
      }

      console.log("Valid user token");

      // Set the user ID on the socket object
      socket.userId = decoded._id;
      socket.latest_version = decoded.latest_version;

      // Call the next middleware in the chain
      return next();
    });
  };
```

Figure 86 - Adding user data to the socket request object if the request came form the client, and skipping authentication otherwise.

In the above middleware, when an 'x-is-pi' header is received, we know a request has come from the Raspberry Pi, and so authentication can be skipped by running 'next()'. Otherwise, a request has come from the client, and so a token must be provided and decoded to attach to the user object for SocketIO to use. Although unrealistic, the closest approximation RAID can make to a production app with a Raspberry Pi per user is using the 'lastUserToConnect' variable. When a user authenticates through SocketIO to make a request for a screenshot, they become the 'lastUserToConnect' and this is set on the server side. In this way, every user to request a screenshot is set as the last user, allowing the Raspberry Pi an awareness of which user folder to save an automatic screenshot to. This is necessary because the Raspberry Pi itself does has no means of knowing which user is currently logged in on the client-side, so simply saves automatic screenshots to the folder of the last logged in user the middleman server is aware of. When the takeScreenshot function is used, a base64 encoded latest image is passed, as well as the 'data.toLowerCase()' which is the class name of the object which was detected, as well as the userID of the last user to connect.

To prevent the same object detection event from causing multiple screenshot uploads, it was also necessary to add an 'uploadInProgress' flag. The uploadInProgress variable is only set to false when an upload has completed, and a detection event cannot trigger a screenshot upload until this value is false. It was also necessary to prevent 'spamming' of screenshots by implementing a timeout. Every time a screenshot is taken, a takeScreenshot.lastTime variable is initialised to the current time. When a new screenshot request is made, we first check if the screenshot type is set as 'automatic' which is a flag provided by an automatically captured screenshot.

```
const takeScreenshot = async (
    // additional parameters here...
    type = "automatic"
  ) => {
    // Don't allow taking more than 1 screenshot in the space of 30 seconds (to prevent spamming by automatic captures)
    // Users can manually take as many as they want to
    if (type === "automatic") {
      if (
        takeScreenshot.lastTime &&
        Date.now() - takeScreenshot.lastTime < 30000
      ) {
        return Promise.reject(new Error("Cannot take another screenshot yet"));
      }
    }

    return new Promise(async (resolve, reject) => {
      if (uploadInProgress) {
        reject("An upload is already in progress");
      }

      // additional code here involving the API request to Cloudinary...


      takeScreenshot.lastTime = Date.now();
    });
  };
```

Figure 87 - The takeScreenshot function with a 30 second timeout added to prevent spamming.

This check is in place to ensure that the user can capture as many manual screenshots as they like, as often as they like. However, if the automatic flag is set to true, we compare the current time with the time of the most recently captured screenshot, and if the time since the last screenshot has been less than thirty seconds, we reject, not allowing a screenshot to be captured until more than thirty seconds has elapsed since the previous screenshot was captured.

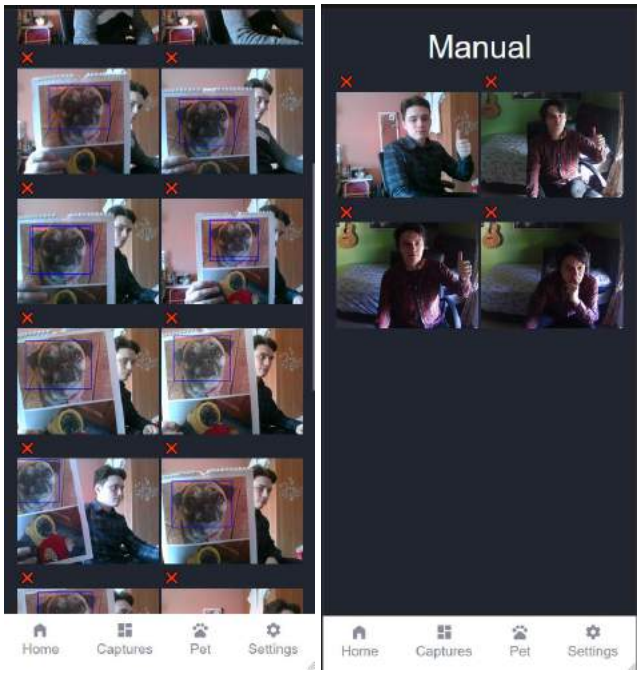


Figure 88 - Automatic (L) and manual (R) screenshots

Finally, screenshots can be deleted using the '/api/delete-screenshot' endpoint, which requires only an image ID as its sole query parameter. This function then uses the user authentication data to find the unique user ID from which to delete the image. The Cloudinary SDK then provides the 'destroy' method, to which we pass the ID passes in the query parameters. In this way, a specific image is removed from a user's unique Cloudinary folder.

```
app.use("/api/delete-screenshot", authenticateJWT, (req, res) => {
    if (req.query?.id) {
      cloudinary.uploader
        .destroy(req.query.id)
        .then((response) => {
          if (response.result === "ok") {
            res.status(200).json({
              msg: `Resource deleted ${response}`,
            });
          } else {
            if (response.result === "not found") {
              res.status(404).json({
                msg: response.result,
              });
            }
          }
        })
        .catch((e) =>
          res.status(500).json({
            msg: `Error while deleting resource: ${e}`,
          })
        );
    } else {
      res.status(400).json({
        msg: "Please provide an image ID",
      });
    }
  });
```

Figure 89 - The endpoint for deleting an image from Cloudinary.

5.13.6 Calculating Average Detection Confidence

Although the Raspberry Pi's YOLOv4-Tiny animal recognition model seems accurate, false positives are still an issue. In particular, the model often erroneously predicted 'dog-american_pit_bull_terrier', so this class is now ignored. To further mitigate against false positives and prevent 'spamming' of detections, causing several screenshots of the same object, a cooldown timer and minimum average calculation is now used when the middleman server is notified of a detection event. The image, class label, and confidence value are now all sent from the Raspberry Pi to the middleman, and the confidence is first parsed to an integer. If the confidence value is less than a minimum confidence value, no screenshot is captured and no notification is emitted to the client-side, the image is simply sent as normal.

If the confidence value is high enough, the current datetime is captured, and a 'timeElapsedSinceLastDetection' value is set to the current time minus the last time a positive detection occured, represented in seconds. Outside the main setInterval, a 'detections' array has been initialised, which is used for storing key/value pairs for each unique detection, storing the number of times an object was detected, the time it was first detected, and the confidence value. When an object is detected which is not already in the array of detection objects, it is added, with its count, confidence, and detection time values initialised to 1, the confidence value received from the Raspberry Pi, and the current time, respectively.

When a class label *is* already in the detections array, its 'count' value is incremented, and its most recent confidence value is added to its existing confidence value.

```
const confidence = parseInt(res.confidence * 100);

// check if the confidence is above the minimum
if (confidence < minConfidence) {
  console.log(`Skipping ${label} detection, confidence below minimum`);
  return;
}

const now = Date.now();
const timeElapsedSinceLastDetection = (now - lastDetectionTime) / 1000;

// add detection to the detections object
if (!detections[label]) {
  detections[label] = {
    count: 1,
    confidenceSum: confidence,
    firstDetected: now,
  };
} else {
  detections[label].count++;
  detections[label].confidenceSum += confidence;
}
```

Figure 90 - Adding objects to the detection array when a detection is received from the Raspberry Pi

Then, whenever a detection occurs, we read the detection object from our detections array for that class name, and check if the 'count' value exceeds a pre-defined 'detectionWindow'. This means we need to detect an object a minimum number of times. We also then check if the 'timeElapsedSinceLastDetection' value is less than the 'cooldown' time, which is set to thirty seconds. For example, if a 'pug' were detected, we would check if the time elapsed between the first time we saw a pug and the current detection event has been less than thirty seconds, and we've seen a pug more than five times in that same space of time, we likely have a positive detection.

```
if (detections[label].count >= detectionWindow && timeElapsedSinceLastDetection <= cooldown) {
    // if we get to here, it means we've detected the same thing 5 or more times within the timeframe of the cooldown period

    // calculate the average confidence and make sure the average confidence of our 5 detections is over 50%
    const avgConfidence = detections[label].confidenceSum / detections[label].count;

    // check if the average confidence is above the minimum
    if (avgConfidence >= minConfidence) {
      console.log(`Emitting detection event for ${label} with average confidence ${avgConfidence}`);

      socketClient.emit("detection", label);
    }

    // reset the detections for this label
    detections[label] = undefined;
}

// update the time of the last detection
lastDetectionTime = now;
```

Figure 91 – Checking the number of detections, and the time elapsed since the previous detection

We finally calculate an average confidence value by dividing the detection object's 'confidenceSum' by the number of detection events for that label. If the average confidence is above 50%, we emit a detection to the client, and a screenshot may be captured. We then reset the detection object matching the class name of the current item. Outside our conditional, regardless of whether we've confirmed a positive detection or not, we update the last detected time for our class name to the current time.

5.13.7 HiveMQ / MQTT

A conceptually difficult issue to solve was allowing the user to trigger the Raspberry Pi's buzzer over the Internet. The difficulty of this is due to the fact that neither the React client nor the middleman server have any knowledge about the Raspberry Pi. They don't know its IP address, and the Raspberry Pi's Express and Socket IO servers only run locally. One solution considered was acquiring a static IP address and performing port forwarding on my local network to allow the Raspberry Pi to be accessed over the Internet, so a REST endpoint setup on the Pi could receive requests from the middleman server directly. However, this solution seemed inflexible, and would mean the Pi would need to be running on my local network only. It couldn't be connected to a different network. (AWS, n.d)

After some research, MQ Telemetry Transport, or MQTT, was found to be an ideal candidate for handling this functionality. In traditional network communication, our client and server communicate directly with each other network addresses and ports. The MQTT protocol operates on a publish/subscribe model, where the message sender and message receiver have no direct knowledge of one another. Instead, a third component, known as the message broker, is used. The message broker listens for incoming messages received from 'publishers', and distributes them to its clients, known as 'subscribers'. In RAID's case, the middleman server is the publisher, and the Raspberry Pi is the subscriber. However, in MQTT, both are known as 'clients'. The clients never need to connect directly to each other, and only communicate via the broker. The MQTT clients connect to their broker, and once connected, can send, or receive messages to and from the broker. When the broker receives a message, it is forwarded to any subscribed clients.

HiveMQ provides a free MQTT cloud broker service, which acts as the broker for the middleman server (the publisher), and the Raspberry Pi (the subscriber).

Figure 92 - HiveMQ Broker dashboard

The HiveMQ broker dashboard shows three active MQTT client sessions. One is the middleman server hosted on Heroku, and two others are the middleman server and Raspberry Pi server running locally. With the MQTT broker active, the React client can now send a request to the middleman server's '/publish-mqtt' endpoint when the user presses the 'buzz' button. On the server, the MQTT.js library has been installed, to allow the server to connect to the broker.

The server establishes its connection to the broker using the HiveMQ URL and login credentials and publishes a simple 'true' message to the broker's 'rpi/buzz' topic. The Raspberry Pi establishes the same connection, but instead of publishing messages to the broker, it subscribes to the same ‘rpi/buzz’ topic the middleman is publishing to. When it receives a message from the broker, it triggers the buzzer.

```
var mqtt = require("mqtt");

var options = {
  host: process.env.HIVE_MQ_URL,
  port: 8883,
  protocol: "mqtts",
  username: process.env.HIVE_MQ_USERNAME,
  password: process.env.HIVE_MQ_PASSWORD,
};

// initialize the MQTT client
var client = mqtt.connect(options);

// setup the callbacks
client.on("connect", function () {
  console.log("MQTT Connected");
});

client.on("error", function (error) {
  console.log("MQTT error: ", error);
});

app.use("/api/publish-mqtt", (req, res) => {

  client.publish("rpi/buzz", "true");

  res.status(200).json({});
});
```

```
// subscribe to topic 'my/test/topic'
client.subscribe("rpi/buzz");

client.on("message", function (topic, message) {
  // called each time a message is received
  console.log("Received message:", topic, message.toString());

  buzz();
});
```

Figure 93 - The Raspberry Pi subscribes to messages (Bottom) and the middleman server publishes messages (Top)

6 Testing

6.1 Introduction

This chapter describes the testing that has been undertaken for the application. This chapter will be presented in two distinct but related parts, functional testing and user testing. functional testing will seek to confirm that the software's functionality corresponds with the functional requirements.

This type of testing is carried out by testing small aspects of the functionality individually, to determine whether the actual output corresponds with the expected output. The user testing aspect of the chapter will then seek to determine whether the app is user-friendly and intuitive.

6.2 Functional Testing

The application's expected functionality has changed in several ways since the functional and non-functional requirements were written based on the survey responses. Namely, the QR code pairing system has been removed, and the animal whitelisting settings have been scrapped. These changes aside, the functional tests conducted here are written in line with the functional requirements.

These functional tests can be categorised across the following domains:

- Authorisation
- Image Annotation and Upload
- Animal Recognition
- Detection Response (Screenshot, Buzz)
- Manual functionality (buzz, screenshot)

In carrying out these tests, we are uninterested in the internal logic of the system, and instead seek only to confirm the actual output agrees with the expected output. This is known as a Black Box Testing technique.

6.2.1 Authorisation

Test No	Description of test case	Input	Expected Output	Actual Output	Result
1	Only authenticated users can access the app	Attempt to access every page of the app while logged out	App only allows access to the login/register screen	App only allows access to the login/register screen	PASSED
2	Unauthenticated users redirected to the login page	Attempt to access every page of the app while logged out	Redirected to the login page in every case	Redirected to the login page in every case	PASSED

3	User can refresh the page and is still logged in	Log in as user 'joe_bloggs', soft and hard refresh browser	User is still authenticated after refresh	User is still authenticated after refresh	PASSED
4	User is redirected to the upload page if they have not yet generated a dataset version	Register a new user, 'joanne_blogs', and login	Immediately redirected to the 'upload' page	Immediately redirected to the 'upload' page	PASSED
5	User can logout	While logged in as 'joanne_blogs', go to settings page, and press 'logout'	User is logged out and redirected to the login page, user token removed from cookies	User is logged out and redirected to the login page, user token removed from cookies	PASSED
6	Registering a user also creates a unique Roboflow project	Registered user 'joanne_blogs' and visited Roboflow.com to check projects	Project created matching user ID '6452b672a684e3f9d6605bee'	Project created matching user ID '6452b672a684e3f9d6605bee'	PASSED

6.2.2 Image Annotation & Upload

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
7	User can annotate and upload images to Roboflow	Uploaded and annotated 12 images of a cat, pressed 'finished'	Upload loading screen displayed, followed by success message	Upload began, but the server threw an error. Loader ran infinitely.	FAILED
8	Retry Above	Uploaded and annotated 12 images of a cat	Upload loading screen displayed, followed by success message	Upload loading screen displayed, followed by success message	PASSED Upload directory was missing from server. This did take much longer than

					expected but succeeded.
9	After reaching Roboflow, annotations are assigned to the correct image and appear as expected	Manually inspect annotated images after Roboflow upload	Images and their corresponding annotations uploaded in the right order, with accurate coordinates	Images and their corresponding annotations uploaded in the right order, with accurate coordinates	PASSED

Figure 94 - Images annotated and uploaded successfully.

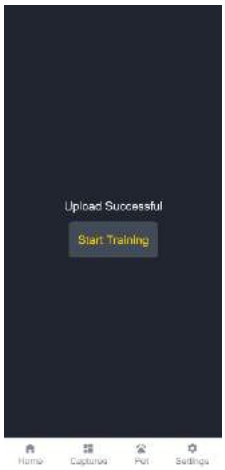


Figure 95 - Prompting the user to begin training after successful image upload.

6.2.3 Custom Model Training

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
10	Users can train a custom object detection model from within the app	Logged in as user 'paul_doyle', who has already created a dataset	Client displays 'training has begun' and Roboflow site displays	Client displays 'training has begun' and Roboflow site displays	PASSED

			training in progress	training in progress	
11	Custom model performs as expected on test data	Having trained a custom model, provided an image of the same dog the model trained on, and another dog	The user's pet is detected accurately, and the unknown pet is not detected	The user's pet is detected accurately, and the unknown pet is not detected	PASSED
12	User can view statistics on their model from within the app	Logged in as user 'joe_bloggs' who has already trained a model	Can view statistics relating to number of images and model accuracy on 'pet' page	Can view statistics relating to number of images and model accuracy on 'pet' page	PASSED

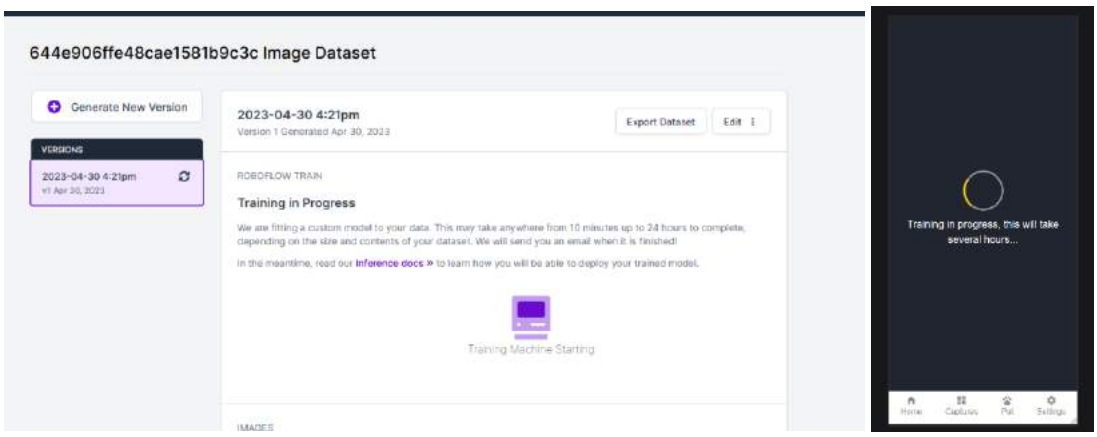


Figure 96 - Roboflow dashboard showing model training in progress (L) and RAID showing model training in progress (R)

6.2.4 Animal Recognition

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
13	With a custom model trained, the app correctly determines whether an animal is the user's pet	Logged in as user 'joe_bloggs', monitored console logs to view detections	User's pet 'lola' is detected and printed to the console	User's pet 'lola' is detected and printed to the console	PASSED

14	Animals in general can be recognised	Presented the Raspberry Pi's camera with an image of a pug	Raspberry Pi predicts the dog as being a dog of breed 'pug'	Raspberry Pi predicts the dog as being a dog of breed 'pug'	PASSED

6.2.5 Detection Response (Automatic Screenshot, Buzz)

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
15	Raspberry Pi triggers its piezo buzzer in response to a non-pet animal detection	Logged in as user 'joe_bloggs' and showed an image of an unknown dog to the Raspberry Pi's camera	After a few moments, the Raspberry Pi's piezo buzzer is triggered, sounding ten times	After a few moments, the Raspberry Pi's piezo buzzer is triggered, sounding ten times	PASSED
16	App automatically captures and saves screenshots of detected animals	Presented an image of a pug to the Raspberry Pi's camera, waited for a positive detection	An image is added to the currently logged in user's 'dog' detection folder	An image is added to the currently logged in user's 'dog' detection folder	PASSED

6.2.6 Manual Functionality

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
17	User can manually trigger the piezo buzzer from within the app	Pressed the 'buzz' button on the client-side	The Raspberry Pi's piezo buzzer is triggered, sounding ten times	The Raspberry Pi's piezo buzzer is triggered, sounding ten times	PASSED
18	User can take a manual screenshot from the app	Pressed the 'screenshot' button on	A new screenshot is added to the	A new screenshot is added to the	PASSED

		the client-side	‘manual category in the user’s image directory	‘manual category in the user’s image directory	

6.2.7 Cloudinary Integration

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
19	Users can view and delete images from their Cloudinary folders	Viewed the currently logged in user’s ‘manual’ screenshot directory, then pressed the delete icon	The image is deleted from the Cloudinary directory corresponding to the logged in user’s ‘manual’ screenshot directory, and removed from state on the client-side	The image is deleted from the Cloudinary directory corresponding to the logged in user’s ‘manual’ screenshot directory, and removed from state on the client-side	PASSED

6.2.8 Discussion of Functional Testing Results

All but one of the functional tests passed as expected, and the application's functionality seems to be complete and aligns with the functional requirements set out earlier in this document. The single test which failed has since been corrected and passed after running the test again. The 'uploads' directory had been missing from the middleman server, due to the fact that Git is incapable of committing an empty folder. After a recent commit, the directory had been ignored by Git and so was not present when the project was cloned. To correct this, the 'fs' library was used to instead create the uploads directory every time the server runs, if it does not already exist.

6.3 User Testing

While the application's functional tests completed for the most part without issue, the application's usability in practical terms is also of paramount importance. The client's user interface is minimal, as the most significant part of the application is its business logic, and so users will only need to navigate through a small number of pages and test a handful of features.

Since the application is targeted towards pet owners, friends and family who own cats and dogs were recruited as participants. The user testing was unmoderated, in that users were simply provided with a list of tasks to carry out and given no further instruction on how to complete their tasks. Users were asked to 'think out loud' and relate their thoughts on the user interface and their experience as they carried out the tasks.

The tasks carried out by each user were as follows:

- **Task 1**: Register an account (Not prompted as to which registration method to use)
- **Task 2**: Upload images of your pet
- **Task 3**: Label the images of your pet
- **Task 4**: Begin training a model
- **Task 5**: When task 4 has completed, view your model's accuracy statistics
- **Task 6**: View the live video feed
- **Task 7**: Take a screenshot
- **Task 8**: Trigger the buzzer
- **Task 9**: View your manual screenshots
- **Task 10**: Show the camera a photo of a dog, and view the screenshot automatically captured
- **Task 11**: Logout of the app

After completing all eleven tests, users were provided with a list of questions in order to provide written feedback on their experience. Some questions were given in a Likert scale format, where users were given a phrase, and asked to agree or disagree. For individual comments on each task, users were also provided with open-ended questions corresponding to each task and Likert question. Finally, the users were asked for overall feedback on the application. The questions provided after each test were as follows:

Likert 1:

- The registration process was straightforward and easy to understand.
- The amount of information requested during registration makes sense.

Likert 2:

- It was easy to upload my images.
- The upload process was quick and efficient.

Likert 3: The labelling process was easy to understand.

Likert 4: It was easy to begin training a model.

Likert 5:

- The accuracy statistics were presented in an easy-to-understand manner.
- The information presented was useful to me for assessing the performance of the model.

Likert 6:

- It was easy to find the live video feed.
- The video feed clearly shows objects being accurately detected.

Likert 7:

- It was easy to find the screenshot button.
- The screenshot was captured quickly and accurately.

Likert 8:

- It was easy to find the button to trigger the buzzer.
- The buzzer was loud and clearly audible.

Likert 9: It was easy to locate my manual screenshots.

Likert 10: After showing the camera a dog, I could easily find the captured screenshot.

Likert 11: It was easy to logout of the app.

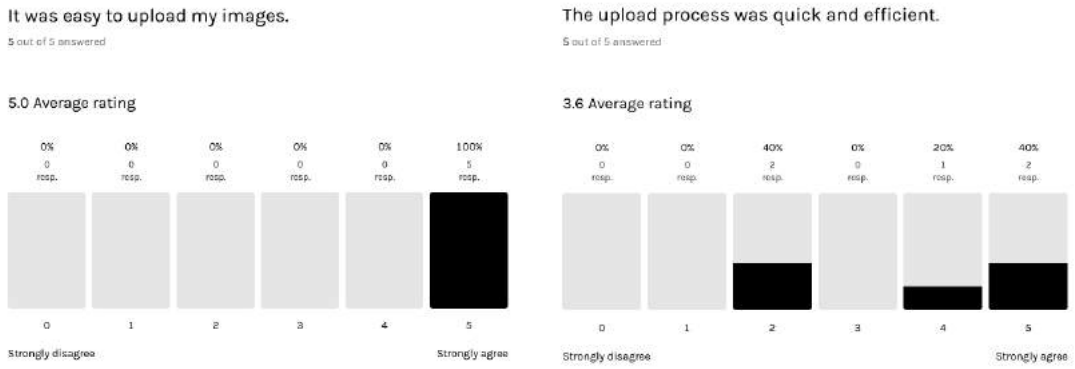
Overall feedback:

- How would you rate your overall experience with the application?
- How was your experience navigating and using the application?
- Did you experience any problems while using the application?
- Do you have any additional comments about the application?

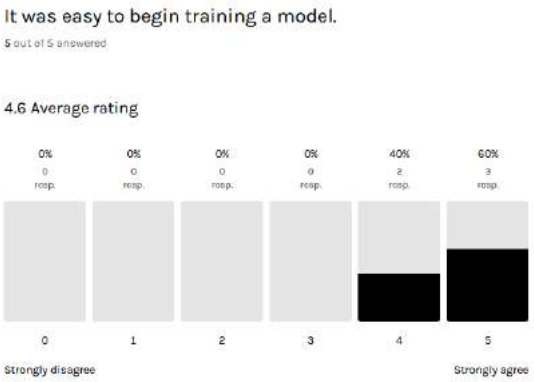
For the testing participants' ease of use, a form was created using Typeform to allow users to answer the Likert and open-ended questions. In total, five users were agreed to participate in the testing. Users were provided with a desktop PC running the application locally. The results are outlined and discussed below.

6.3.1 Discussion of User Testing Results

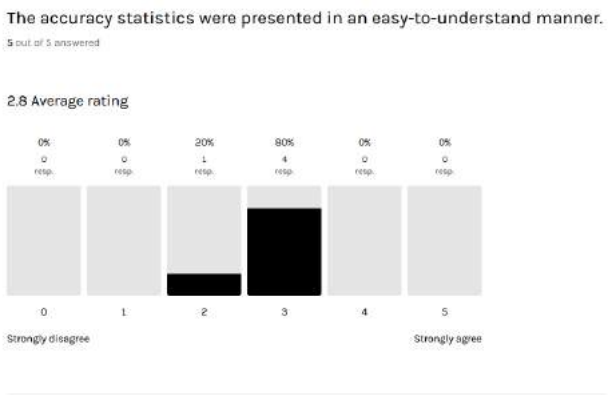
For both questions relating to the registration process, users had no issue, though one user (20%) did feel the requirement to provide both a username and an email address was strange. This feedback is noted, though the reasoning for this is due to a limitation in the customisability of PassportJS, which is detailed in the implementation the implementation chapter.

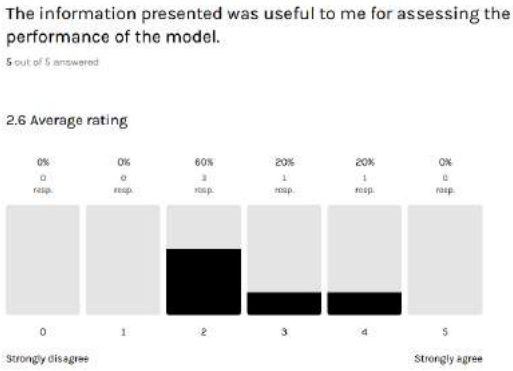


Regarding the upload process, users had no issue with the image upload itself in terms of adding a series of images and viewing them within the app in gallery format. However, responses were mixed when users were asked to rate the speed and efficiency of the upload process. Users commented that the upload takes a long time to complete. The process of uploading an annotated image, as detailed in the implementation chapter, is complex, and involves many steps. Little can be done to improve the upload speed, but a sensible user suggestion is to add visual feedback to the upload step, detailing the processes being carried out in the background.

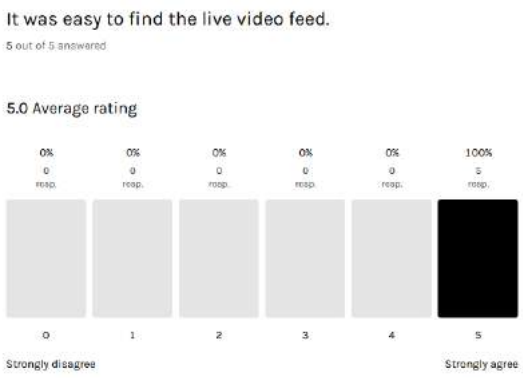


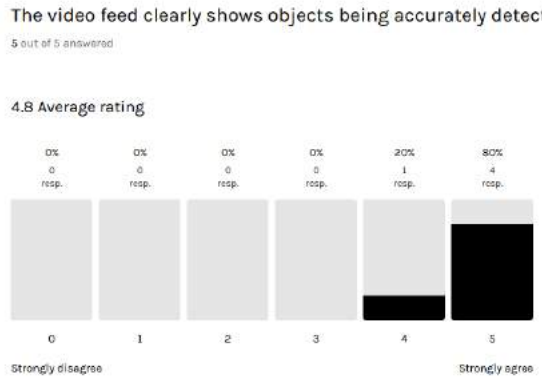
In terms of annotating their images and beginning the training process, users had no issue beginning the training, as expected, but 60% of users had some difficulty understanding the labelling process.



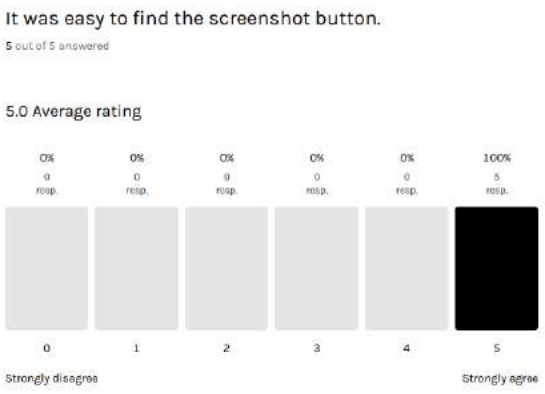


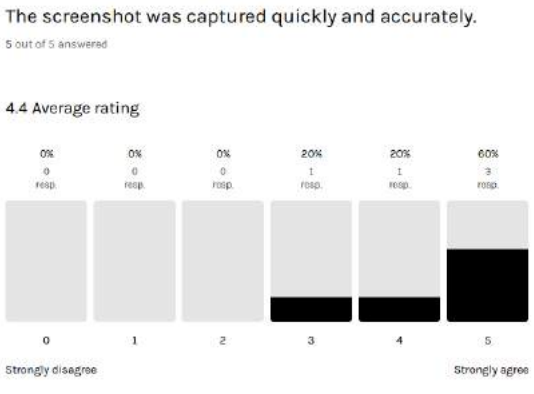
After users had trained a model, feeback relating to the model accuracy statistics was mixed. Users generally disagreed that the information presented after a model completes its training is useful for assessing the model's performance. Similarly, users also felt the statistics are not presented in an easy-to-understand manner. Respondents were unsure what the accuracy statistics represent.



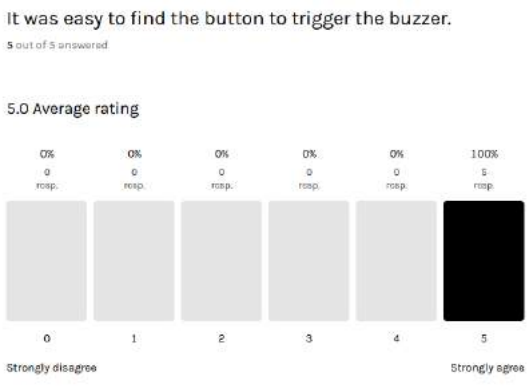


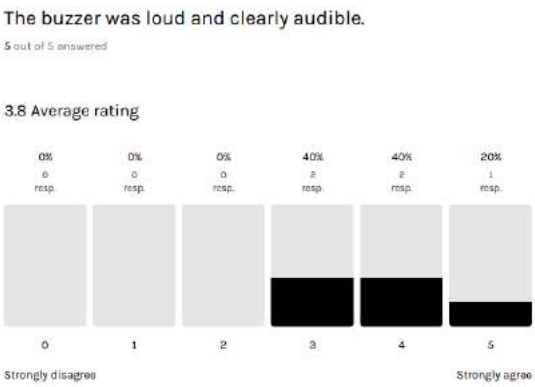
Positive responses were received in relation to the live video feed, suggesting the navigation aspect of the app is appropriate in terms of design, as users successfully navigated to the homepage in order to view the video feed after their model had completed training.



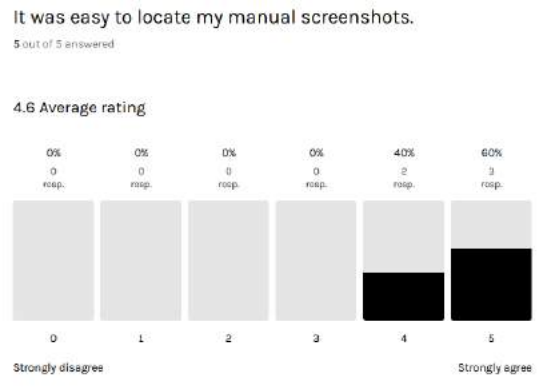


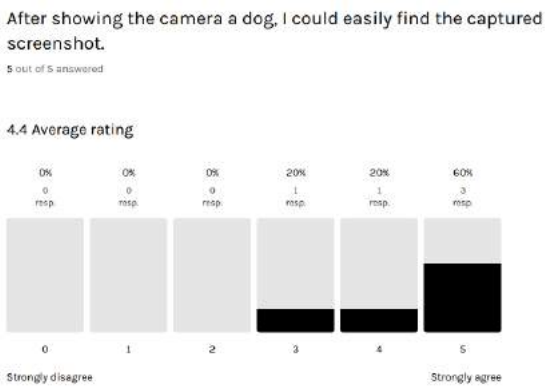
When users were asked to capture a manual screenshot, no issues were encountered in finding and using the screenshot button. Some variation in experience occurred in terms of the speed with which the screenshot was captured, which was possibly simply due to unreliable Internet connections. In any case, the manual screenshots were captured and saved to the user's manual screenshot folder.





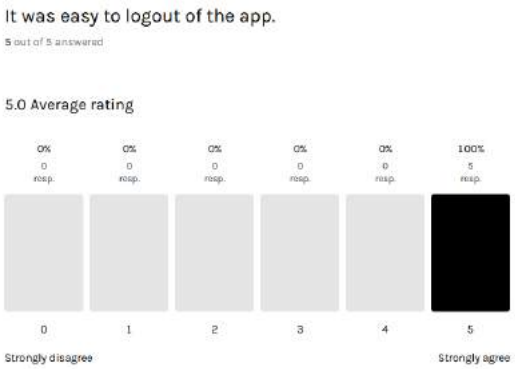
Users were also asked to manually trigger the buzzer using the 'buzz' button. Again users had no issue finding and using this button. In terms of the buzzer's audability, responses were generally positive, but some users clearly felt the system would benefit overall from a louder buzzer. In this case, it cannot be made any louder due to the hardware limitations of the Piezo buzzer itself.





Regarding screenshots, users were asked both to capture a manual screenshot, and to show the system an animal (not necessarily a dog) in order to trigger the automatic screenshot capture. In

either case, the screenshot was successfully captured, and users were able to view the saved screenshot afterward. However, users noted that in some cases, the system would take a moment to recognise the animal and trigger the screenshot capture.



For the final Likert-style question, users had no issue with logging out of the app, and when asked to rate their overall experience, the response could be considered to hover around 55-60% approval of the user experience. When asked to describe their experience navigating around and using the application, the responses were as follows:

> *I didn't have a problem navigating the app, wasn't sure what to do after uploading images, didn't realise straight away that you have to draw labels on them.*
>
> *Moving around the app is easy and it's easy to use. Good experience.*
>
> *The navigation aspect of the app is easy, there aren't that many pages and the navbar at the bottom makes it clear how to get around. Wasn't sure what to do on the labelling page at first, and was confused as to why it was taking so long to upload images.*
>
> *Easy to navigate and everything was straightforward. some of the processes took a very long time to complete, and I was left wondering if something had gone wrong. It would have been helpful to have some sort of visual feedback on what was happening in the background.*
>
> *I found navigating and using the application to be a fairly straightforward experience. There were a few areas where I got a bit stuck, but I was able to figure things out with a bit of trial and error.*

The responses to this question generally suggest that users take no issue with the navigation and had little difficulty understanding how to use the app for the most part. However, users were somewhat confused with the labelling process, and were unsure whether the app was operating normally during image upload and model training. Users were then asked to describe any problems they encountered while using the application:

> *It couldn't recognise my dog. The buzzer went off no matter what animal it saw.*
>
> *It doesn't seem like it's able to recognise my pet. I probably didn't upload enough images, but there's nothing to tell you how many you need to upload.*

Nothing broke, no major issues. Was confused by the time everything took.

Nothing major went wrong. I had to show the camera the dog photo twice before the screenshot was saved.

I wasn't sure what to do on the annotation page at first. Also, when I uploaded my images, it took longer than I expected, and I thought something might be wrong. Overall, and I was able to navigate the application without too much difficulty.

Users did not describe any major errors, but again one user recounted having some difficulty understanding the purpose of the annotation page, suggesting instructions may be necessary so users are aware they can draw bounding boxes on their images. Two users also did not upload a sufficient number of images to train an accurate model for recognising their pet, meaning the buzzer would be activated regardless of whether the system had seen the user's pet, or an unknown animal. Finally, users were asked to include any additional comments they have regarding their experience using the app:

Good idea overall but was a bit confused regarding how to use it.

Same as previous feedback, need to enforce a minimum number of images if one is required.

It would be helpful to provide feedback or a message letting you know what's happening in the background while images are uploading, and the model is training. Obviously, these things take some time, but maybe something other than just a loader. Also, a bit strange it asks for both a username and an email address when registering.

Good idea. I found it easy to use overall, just more visual feedback is needed for when things are happening in the background, since it can take so long.

It's a good idea. Annotation page was a bit confusing at first. The image upload process took a bit longer than I expected, and I wasn't sure if something had gone wrong. Checked again after a few hours and the model was trained, but I found the accuracy statistics a bit confusing, especially the term "mAP" - I wasn't sure what it meant. Other features easy to find and use.

6.4 Conclusion

To conclude the functional testing and user testing sessions, the application's functional testing demonstrated that its functionality aligns with the functional requirements set out earlier in the document. A single functional test failed, which then passed following a bug fix.

The user testing, which involved a list of tasks for pet owners, friends and family to complete, showed that users had no major issues with the registration process or uploading images, but had mixed feedback on the speed and efficiency of the upload process. Users also had some difficulty understanding the labelling process, and feedback on the accuracy statistics of the trained model was mixed. Overall, users found the live video feed and screenshot capture process to be easy to

use, and useful feedback has been received on how the application could be improved, such as adding visual feedback during the upload process.

7 Project Management

7.1 Introduction

In delivering any large project, a robust project management system is imperative. Software development in particular carries its own unique challenges. Technology changes, poor documentation, difficult bugs, and many more. A strong project management strategy can help to keep the development team on track and avoid becoming bogged down in low-level issues. When RAID's design had been finalised, a Kanban board and task backlog was created using Jira Software.

7.2 Project Management Tools

7.2.1 Jira

Jira is a project management and bug tracking software designed specifically for agile software project management. Jira facilitates a wide range of features, including SCRUM management and Kanban boards, and integrates with version control systems such as GitHub or Bitbucket to efficiently track progress.

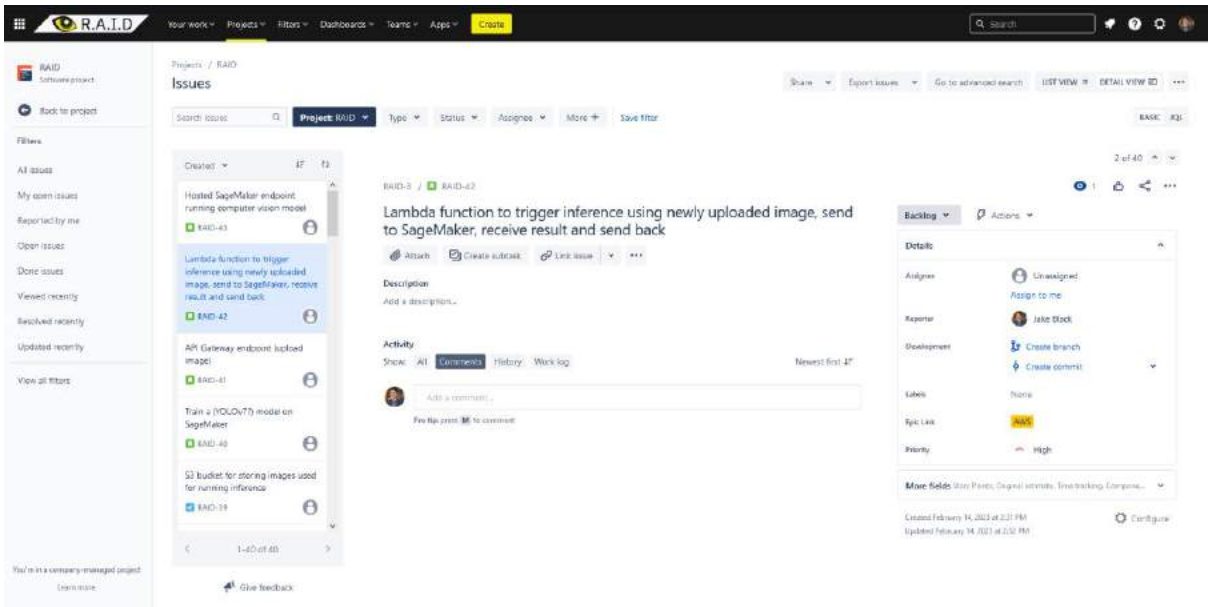


Figure 97 - An example Jira story, assigned a difficulty of 'high' and connected to the 'AWS' epic.

Forty tasks were immediately added to the Jira backlog, categorised by epics, stories, and tasks, with individual stories and tasks linked to an epic, and assigned a priority level and difficulty. Issues were related to one another in a hierarchal manner, whereby every sprint had stories, and every story had tasks. The epics, in turn, were divided into three categories: frontend, backend, and AWS.

Quite a significant amount of time was spent creating this Jira board, dividing the backlog issues into epics, stories, and tasks, and assigning difficulty ratings and associated epics or stories to each issue individually.

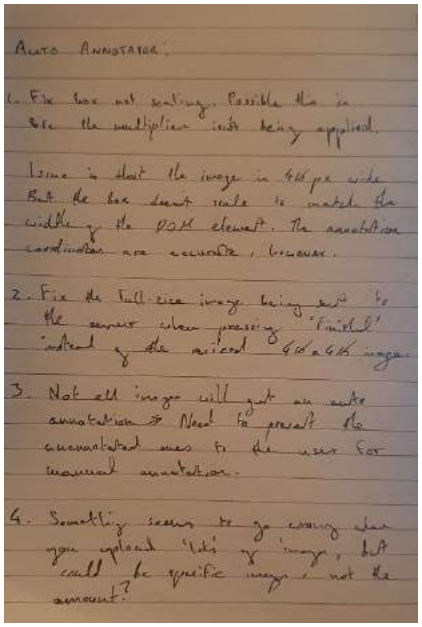


Figure 98 - An example page of notes documenting necessary changes to the proposed auto-annotator feature.

However, as the enormity of the task of developing the project became apparent, and the number of tasks in the development backlog grew exponentially, the Jira board was abandoned. From this point on, a physical notepad was used as the main source of project management. Roughly forty pages of notes were written over the course of the project, detailing the most important backlog items, notes on potential bug solutions, and functionality which was of high importance with regard to testing.

7.2.2 GitHub

To understand what purpose GitHub serves, we first need some background on what version control in general does. When dealing with a large software development project, bugs and abandoned features will eventually become an issue. Version control software provides software developers with a means of tracking the history of their project's changes, separating experimental features, and merging completed features into their base code. In version control, an offshoot of the main source code used for the purpose of creating a new feature is known as a branch, and combining such a branch with the main codebase is known as merging. Other functionality afforded to the developer by version control systems is the ability to add and track changes to their codebase, and revert these changes if need be. There are a range of version control systems available, but the most popular of these is Git, which is an open-source version control system (VCS) created in 2005.

All of the functionality described above, and much more, is available to a software developer by installing Git on their local machine. However, beyond purely local version control, several companies offer hosted platforms which integrate with Git to allow for changes to be tracked remotely, code repositories to be made available to the public, or to facilitate collaboration between team members. One such platform is GitHub, which facilitates the use of all the Git features described here in terms of a live platform, with repositories available on a public or private basis.

Git and GitHub were used as the version control system for this project.

The RAID project consisted of three distinct but interdependent components, the React client, the Express middleman server, and the Raspberry Pi. Every significant change was committed to GitHub, and significant features which were considered experimental and separate from the main codebase

were delegated to separate branches. An example of using a separate branch for a significant but experimental feature is the auto-annotator described in the implementation chapter.

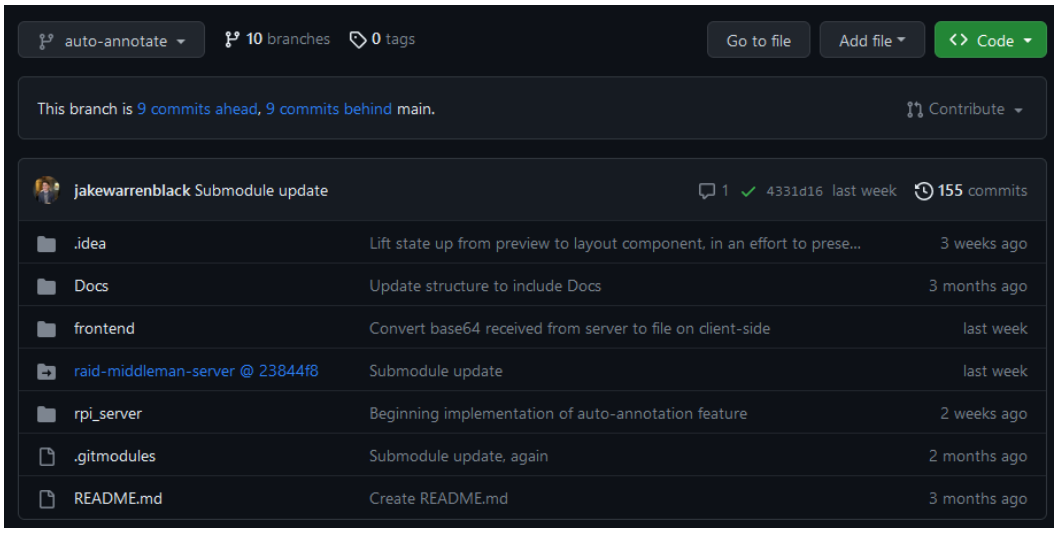


Figure 99 - Demonstrating the feature branch used for the auto-annotator.

An interesting feature of Git/GitHub which was useful for RAID's development and version control was the submodule.

A submodule allows the developer to keep one Git repository as a subdirectory of another. In practical terms, a submodule is simply a reference to another repository. In RAID's case, it was anticipated that the Raspberry Pi server and the React client might progress at a different pace to the Express middleman server, and a logical separation existed between the three. It may also have made sense to extract the Raspberry Pi server into a separate module, but after a certain point in the project's development, changes to the Raspberry Pi had slowed, and so it made sense particularly in the middleman server's case to extract it into a separate submodule.

8 Project Reflection & My Views on The Project

In summarising my experience developing this project, significant difficulty was had at every phase of implementing this project, as is evidenced by the fact the project's development has essentially restarted from scratch several times. As an undergraduate student inexperienced with professional software development, issues which seemed at times insurmountable have been overcome, albeit with near untold difficulty.

Beginning with initial conceptual difficulties imagining how the project could work beyond a mere prototype, through to incurring unacceptable costs from AWS, a false-start in mistakenly implementing object detection on the client-side, and one of the most time-consuming and difficult to overcome bugs - the race condition. As described early in the implementation phase, a race condition was encountered in which the YOLO algorithm, OpenCV accessing the webcam feed, and WebSocket attempting to stream frames over the Internet at the same time caused the application to reach a deadlock.

Solving this issue absorbed significant valuable time from the project's development, after attempting to use gevent, eventlet, and threading async modes with Flask, starting the SocketIO server as a background task, providing Flask-SocketIO with and YOLO with separate threads, and adding inference results to a queue while waiting for webcam streaming to finish, and finally using gunicorn to run the app on Ubuntu, and using Waitress to run the app in Windows, no luck was had.

Potential solutions examined even included switching entirely to a raw C language implementation of GStreamer. Neither I nor my supervisor could suggest any solution to this issue. After meeting with another lecturer more familiar with lower-level computing concepts, no solution was reached.

In trying to solve this issue, in desperation I even reached out to a StackOverflow user by finding his LinkedIn profile and asking him had he solved the same issue I was experiencing. The answer was unfortunately no. In the end, this issue was not in fact solved, but was worked around by completely abandoning Python and Flask and switching to OpenCV4NodeJS. This library was considered from the beginning but did not seem to work. The library was only fixed after I had notified the user of the problem in a GitHub issue, which led to a significant turning point in the project's development.

Even after reaching this point, issues persisted after realising the service I had chosen on which to base the custom model training aspect of the project, Roboflow, did have a Python SDK and a NodeJS CLI, but did not yet have an entirely completed or documented NodeJS CLI, nor was every aspect of the API fully documented. After manually converting most of the Python SDK with repurposed functions from the NodeJS CLI, issues persisted. Only with the help of the Roboflow support team was I able to completely implement the custom model training aspect of the project. A bug existed in the Roboflow API which was fixed as a result of my help, after spending an entire night in correspondence with a Roboflow team member in the United States. The result of this correspondence was a bug being discovered in the API and having my own pull request merged to Roboflow's API code snippets repository. With regard to facilitating image annotation, no suitable library could be found for this application’s purposes, and so a custom fork of the React-BBox-Annotator library was used in combination with React-Dropzone-Uploader. The development of this custom fork in itself required a significant devotion of time and effort, especially with regard to completely overhauling the library to support an array of images in gallery format, rather than only a single image, and supporting images in both portrait and landscape format. In the final week of implementation, another issue was encountered with Roboflow's API involving an annotated dataset being interpreted by Roboflow as being unannotated, for which a workaround was hastily found by including in the dataset the values the server interpreted as being unannotated.

8.1 Completing a large software development project

From the point of view of completing a large software development project, RAID has given me an insight into the necessity for experts in various fields when developing such a project. This project has spanned domain which in terms of an entire software development team would span frontend, backend, machine learning specialists, and project managers. While frontend and backend software development are by no means unfamiliar to me, machine learning and project management were two completely new domains. The difficulty I've experienced developing this project has imbued me with a new sense of appreciation for the expertise required across the various domains of software development, as well as the management roles involved in information technology.

8.2 Working with a supervisor

My supervisor, Mohammed Cherbatji, has been invaluable in terms of motivation, project management advice, and debugging. Mohammed provided feedback on my code and facilitated one-to-one sessions in which we discussed potential avenues for solving some of the difficult issues which arose over the course of this project's development.

8.3 Technical skills

From the viewpoint of technical skills, this project's development has both provided an introduction to entirely new fields and allowed me to further develop my skills in domains with which I was already familiar. Prior to this project's development, I was unaware of several concepts, which were entirely new to me, and I have since learned a significant amount about:

- The concept of Git submodules
- The GStreamer software suite
- The capabilities of OpenCV
- SocketIO and the WebSocket protocol
- The MQTT protocol
- The intricacies of wiring a Raspberry Pi's GPIO pins
- The MJPEG video compression format
- PassportJS and the workings of the OAuth 2.0 protocol
- Advanced knowledge of Linux (PopOS) and the Linux command line

I have also gained significant experience in programming with Python, including the Flask web server and the various multithreading libraries available in the Python language. The project has also enhanced my personal contributions to open-source projects by necessitating my contribution to two of Roboflow's GitHub repositories, as well as opening several GitHub issues across multiple repositories, including the OpenCV4NodeJS and React-BBox-Annotator repositories.

8.4 Further competencies and skills

With regard to further competencies and skills, as my career goal is to become a third-level lecturer in the area of software development, this project has beyond a doubt imbued me with a plethora of skills which could be applied to my future studies, and eventually to the workplace. As discussed in the previous section, I have gained insight into many technologies which were previously completely unknown to me, and which will undoubtedly stand to me regarding technical skill in the future. Aside from technical skill, I have also gained a personal insight to the role of project manager and SCRUM master in a large software development project, and while at this point in my career I have not yet had the opportunity to work professionally in either of these roles, the insight this project has provided me with will no doubt contribute to my overall career development.

9 Business Opportunities

I believe business opportunities exist with regard to this project. Based on user feedback in the survey section of this document, the general public seem to be interested in the concept. If the RAID application were to be developed further as a marketable product, it would be necessary for every user to have a Raspberry Pi, or Raspberry Pi-like device unique to their account. If this were accomplished, the rest of the software aspect of the project would be ready to facilitate an entire minimum viable product in the sense that individual users could train a custom model to recognise their pet and act as a deterrent to any wild animals or neighbouring pets they wished to deter from their premises, or an alternative use-case might be a user who owned multiple pets and wished to deter a specific pet from a food bowl, bed, or section of their home, for example if a new puppy or kitten for the purpose of toilet training, or getting a new pet accustomed to the portions of the home they are allowed to be in.

In the future, I am interested in pursuing this project's development further as a potentially marketable product, provided funding were available for marketing RAID's software alongside a Raspberry Pi-like device unique to each user. Furthermore, as the number of users scaled, it would be necessary to purchase a subscription to Roboflow's enterprise tier to facilitate faster and more accurate model training to ensure users were satisfied with the level of accuracy with which the software could identify their pet. Based on user feedback in the testing section, it would also be necessary to implement additional features, and improve the user experience in general in order to alleviate issues users seemed to have experienced while using the application, namely a lack of visual feedback while images were being uploaded, and similarly while the user's unique model was being trained. In conclusion, despite the limitations in existence in the RAID software currently, I believe with further improvements and funding, RAID has potential to be a marketable product, as a niche does seem to exist in the pet owner market for such a system.

10 Conclusion

In a high-level sense, the overall goal of this project was to create a novel system capable of distinguishing one animal of the same species from another. Based on the prior works outlined in the research chapter of this document, similar studies have achieved the same result as RAID. However, this project sought to take this concept to a previously unexplored level by wrapping a model capable of recognising animals with a user-friendly client which allows users to upload and annotate images of their pet with a view to allowing the system to distinguish one animal from another, be it animals of differing species, or even an animal of the same species as the user's pet, and allow users to view a live video feed including bounding boxes surrounding detected animals.

With this accomplished, the project sought to use a Raspberry Pi with a connected Piezo buzzer to automatically sound when an animal other than the user's pet is detected, thereby creating a unique pest deterrent system. Additional functionality includes the ability for the application to automatically capture screenshots of detected animals, categorise them by species, and allow the user to view a gallery of these images. The application also allows the user to manually activate the Raspberry Pi's Piezo buzzer over the Internet, via an MQTT server, similar to the automatic screenshot function, allows the user to manually capture screenshots from the video feed for later viewing.

10.1 Technologies

The technologies used throughout the project's development were many and varied, and included Python, Flask, GStreamer, PopOS Linux, Ubuntu Linux, Amazon services including S3 and Rekognition, and various Python multithreading libraries including gunicorn and eventlet. However, in terms of the final implementation of the project, the core technologies used were as follows:

Client:

- React
- TailwindCSS
- Vite
- React-BBox-Annotator (Custom fork)
- React-Dropzone-Uploader

Server:

- NodeJS
- Express
- SocketIO
- MQTT/HiveMQ
- Mongoose
- PassportJS
- Roboflow (Customised implementation designed by combining NodeJS translation of PythonSDK and NodeJS CLI)

Raspberry Pi:

- OpenCV/OpenCV4NodeJS
- NodeJS
- Express
- SocketIO
- MQTT/HiveMQ
- GPIO pins and Piezo buzzer, breadboard, jumper wires
- YOLOv4-Tiny

10.2 Research

The research section of the project yielded valuable information as to the feasibility of the proposed project. In particular, it confirmed that it was, in fact, possible to use computer vision techniques to distinguish one animal from another. Of particular interest was the study on pig faces described in the research section, which outlines the fact that the computer vision model initially used by the researchers was more than capable of distinguishing one pig face from another, despite having only been trained on human faces. Similar results were reported by researchers across several studies and animal species, which confirmed the possibility of creating the RAID project, and kickstarted development.

10.3 Design

The design phase of the project's implementation forced me as the developer to think more clearly about how the project should be implemented. Until this point, my mental understanding of how the project should work was vague, but creating a high-fidelity prototype of the application caused me to reconsider my design choices several times, as did the initial survey, which provided a significant amount of valuable feedback which further influenced the final design.

10.4 Implementation

As described previously in the concluding chapters, as well as throughout the implementation chapter, implementing the finalised design in practical terms was no easy task, and was littered with roadblocks, false-starts, and entire redesigns, and yet the implementation as of now has for the most part been completed successfully.

10.5 Testing

While the functional testing section passed without issue, the user testing revealed some issues in terms of user experience, ease of navigation, and general understanding of the functionality. Users were somewhat confused by technical computer vision terms when viewing a trained model and were not prepared in advance for to expect the sometimes extremely long model training times required. Unfortunately, the difficulty of implementing the project's core functionality placed serious time constraints on the project's development towards the end, and no time was left available for making the changes necessary to satisfy the changes outlined in the user feedback.

10.6 Overall result

The overall result of the application to me as the developer is satisfactory. While the time constraints of the project did not allow for creating a polished, finished project, the core functionality of the project has all been implemented, and works as expected. Multiple forms of authentication have been implemented as an extra feature, which was not anticipated at first, as was the ability for the application to function over the Internet as opposed to merely over a LAN connection. Beyond this, the application does what it set out to do. It is possible for a user to upload and annotate their own images of their pet and train a custom object detection model to recognise this pet. When the user's custom model has been trained, the Raspberry Pi works in cooperation

with the Express server to run inference via the hosted model endpoint to determine whether a detected animal is the user's pet or not, and the system reacts accordingly by sounding the Piezo buzzer. Regardless of the animal detected, an automatic screenshot is captured and saved to an image directory specific to the species. Using the same business logic described in the aforementioned screenshot and buzzer functionality, the user may also capture a screenshot or trigger the buzzer manually. In general, despite the great difficulty encountered in seeing the project through to its conclusion, all necessary features have been implemented.

10.7 What was Learned & Potential for Further Development

This project has provided a highly challenging opportunity to develop my programming, project planning, and research skills, and has without a doubt greatly broadened and enhanced my skills as a software developer. In terms of the potential for future development, I believe an opportunity exists for this application to become a marketable product with more time and resources.

I hope to continue developing this project with a view to enhancing the existing functionality and user interface to create a more streamlined, polished experience, and in particular hope to spend time implementing the auto-annotation functionality described in the implementation chapter, as I believe alleviating the burden of annotating the large number of images required in order to train an accurate object detection model will be key to creating an easy to use solution to the problem this project sought to solve, creating a polished, accessible product pet owners will enjoy using.

In conclusion, the project has been a highly challenging, yet beneficial experience, and I look forward to continuing its development in the futur

References

Billah, M., Wang, X., Yu, J., & Jiang, Y. (2022, February 10). *Real-time goat face recognition using convolutional neural network.* Retrieved from Science Direct: https://www.sciencedirect.com/science/article/abs/pii/S0168169922000473#preview-section-references

Burnett, M., & Jellisejevs, P. (2020, October 13). *An Introduction to React JSX.* Retrieved from SitePoint: https://www.sitepoint.com/an-introduction-to-jsx/

Chen, Y.-C., Hidayati, S. C., Cheng, W.-H., Hu, M.-C., & Hua, K.-L. (2016). *Locality Constrained Sparse Representation for Cat Recognition.* Retrieved from Sci-Hub: https://sci-hub.st/10.1007/978-3-319-27674-8_13

Color Theory: Yellow as a Branding Colour. (n.d.). Retrieved from Branding Compass: https://brandingcompass.com/branding/color-theory-yellow-as-a-branding-color/

Crouse, D., Jacobs, R. L., Richardson, Z., Klum, S., Jain, A., Baden, A. L., & Tecot, S. R. (2017, February 17). *LemurFaceID: a face recognition system to facilitate individual identification of lemurs.* Retrieved from Springer Link: https://link.springer.com/article/10.1186/s40850-016-0011-9

Dechalert, A. (2021, Merch 26). *Bundling JavaScript: The Good, the Bad, and the Ugly.* Retrieved from Matcha: https://matcha.fyi/bundling-javascript/

deeplizard. (2017, December 9). *Convolutional Neural Networks (CNNs).* Retrieved from YouTube: https://www.youtube.com/watch?v=YRhxdVk_sIs

Dloghin, C. (2022, November 16). *Medium.* Retrieved from Raspberry Pi 4 or Jetson Nano — Which one is better?: https://dloghin.medium.com/raspberry-pi-4-or-jetson-nano-which-one-is-better-b0a9d185abb6

Findlay, T. (2022, January 13). *What's Vite? A Guide to Modern, Super-Fast Project Tooling.* Retrieved from Telerik: https://www.telerik.com/blogs/whats-vite-guide-modern-super-fast-project-tooling

Hansen, M. F., Smith, M. L., Smith, L. N., Salter, M. G., Baxter, E. M., Farish, M., & Grieve, B. (2018, March 28). *Towards on-farm pig face recognition using convolutional neural networks.* Retrieved from Elsevier: https://reader.elsevier.com/reader/sd/pii/S0166361517304992?token=C725BAD2084E8B3E1CB5C529DFDD26537DD07F4780387B91312C7B17163462BB82E8C826866E1BE9640955CD7D57B9CC&originRegion=eu-west-1&originCreation=20221228172103

Hou, J., He, Y., Yang, H., Connor, T., Gao, J., Wang, Y., . . . Zhou, S. (2020, January 13). *Identification of Animal Individuals Using Deep Learning: A Case Study of Giant Panda.* Retrieved from Elsevier: https://reader.elsevier.com/reader/sd/pii/S000632071931609X?token=88482B4E604F2CE701845771F4CB8847B650C9753BAFD399FD52CAFE15532BD0A9E0D1025755BFB4555ABECD9405959D&originRegion=eu-west-1&originCreation=20221228175009

Introduction to Server-Side Development with Node.js and Express. (n.d.). Retrieved from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Koffer, P. (2022, September 14). *How does ReactJS Work?* Retrieved from mDevelopers: https://mdevelopers.com/blog/how-does-reactjs-work-

Kortli, Y., Jridi, M., Al Falou, A., & Atri, M. (2020, January 07). *Face Recognition Systems: A Survey.* Retrieved from Sci-Hub: https://sci-hub.st/10.3390/s20020342

Kumar, S., & Kumar Singh, S. (2016, December 14). *Monitoring of pet animal in smart cities using animal biometrics.* Retrieved from Elsevier: https://reader.elsevier.com/reader/sd/pii/S0167739X16307385?token=4DD1D55AEBA4862A23F25EAA13C6777AA9E90FF55856685091145439FAA6E71E23A1718CF456A9D4571961282EFF478E&originRegion=eu-west-1&originCreation=20221228174438

Kumar, S., Pandey, A., Satwik, K. S., Kumar, S., Singh, S. K., Singh, A. K., & Mohan, A. (2017, October 28). *Deep learning framework for recognition of cattle using muzzle point image pattern.* Retrieved from Science Direct: https://www.sciencedirect.com/science/article/abs/pii/S0263224117306991#preview-section-references

Lin, T.-Y., & Kuo, Y.-F. (2018, July 29). *Cat face recognition using deep learning.* Retrieved from Sci-Hub: https://sci-hub.st/10.13031/aim.201800316

Loos, A., & Ernst, A. (2013, August 19). *An automated chimpanzee identification system using face detection and recognition.* Retrieved from Springer Link: https://link.springer.com/article/10.1186/1687-5281-2013-49

Madarkar, J., Sharma, P., & Singh, R. P. (2021, March 06). *Sparse representation for face recognition: A review paper.* Retrieved from IET: https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/ipr2.12155

Makabee, H. (2012, February 5). *Separation of Concerns.* Retrieved from Effective Software Design: https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/

Mopkar, K. (2021, September 27). *What is Tailwind CSS? A Beginner's Guide.* Retrieved from FreeCodeCamp: https://www.freecodecamp.org/news/what-is-tailwind-css-a-beginners-guide/

Norouzzadeh, M. S., Nguyen, A., Kosmala, M., Swanson, A., Palmer, M. S., Packer, C., & Clune, J. (2018, June 05). *Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning.* Retrieved from PNAS: https://www.pnas.org/doi/abs/10.1073/pnas.1719367115

Peabody, B. (n.d.). *Server-Side I/O Performance: Node vs. PHP vs. Java vs. Go.* Retrieved from Toptal: https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

Popat, P., Sheth, P., & Jain, S. (2018, December 30). *Animal/Object Identification Using Deep Learning on Raspberry Pi.* Retrieved from Springer Link: https://link.springer.com/chapter/10.1007/978-981-13-1742-2_31

Pounder, L. (2020, July 10). *Raspberry Pi vs Arduino: Which Board Is Best For You?* Retrieved from Tom's Hardware: https://www.tomshardware.com/features/raspberry-pi-vs-arduino

Reddy, N. (2020, April 20). *What is NodeJS?* Retrieved from YouTube: https://www.youtube.com/watch?v=yEHCfRWz-EI

Sajjad, M., Nasir, M., Muhammad, K., Khan, S., Jan, Z., Sangaiah, A. K., . . . Baik, S. W. (2017, November 16). *Raspberry Pi assisted face recognition framework for enhanced law-enforcement services in smart cities.* Retrieved from Elsevier: https://reader.elsevier.com/reader/sd/pii/S0167739X17309512?token=7FDB887948473FF5A7654F89484EA2EA6DC26F3F4C4C6BF5642DEAF6EB8D21BCC42EF6B7C381D230947E773A3EAE6CD6&originRegion=eu-west-1&originCreation=20230107223619

Santos, R. (2017, January 12). *What is MQTT and How It Works.* Retrieved from YouTube: https://www.youtube.com/watch?v=EIxdz-2rhLs

Scrum - The Diagram. (n.d.). Retrieved from Emergn: https://emergn.com

Scrum for One: Why One-Person Scrum Teams Work. (n.d.). Retrieved from LucidChart: https://www.lucidchart.com/blog/scrum-for-one

Subramanyam, V. S. (2021, January 20). *Non-Max Suppression.* Retrieved from Medium: https://medium.com/analytics-vidhya/non-max-suppression-nms-6623e6572536

Tan, C. (2022, October 20). *Raspberry Pi vs Jetson Nano: Differences.* Retrieved from All3DP: https://all3dp.com/2/raspberry-pi-vs-jetson-nano-differences/

White, T. (2021, September 20). *What is the Virtual DOM in React?* Retrieved from Dev.to: https://dev.to/turpp/what-is-the-virtual-dom-in-react-3afn

Widyastuti, R., & Yang, C.-K. (2018, October 01). *Cat's Nose Recognition Using You Only Look Once (Yolo) and Scale-Invariant Feature Transformation (SIFT).* Retrieved from Sci-Hub: https://sci-hub.st/10.1109/gcce.2018.8574870

Woodford, C. (2021, August 30). *Introduction to Neural Networks.* Retrieved from Explain That Stuff: https://www.explainthatstuff.com/introduction-to-neural-networks.html

11 Appendix

11.1 Gists

Roboflow REST endpoints:
https://gist.github.com/jakewarrenblack/a4f95f1b1d05de77c9f0dc0c784e5371

Roboflow Utility Functions:
https://gist.github.com/jakewarrenblack/9944917d60277c89817ebbf75a5cea10

11.2 GitHub Issues

OpenCV4NodeJS:

- https://github.com/UrielCh/opencv4nodejs/issues/79
- https://github.com/UrielCh/opencv4nodejs/issues/89

React-BBox-Annotator:

- https://github.com/younesZdDz/react-bbox-annotator/issues/4
- https://github.com/younesZdDz/react-bbox-annotator/issues/5

11.3 Pull Requests

Roboflow CLI: https://github.com/roboflow/roboflow-cli/pull/11

Roboflow API snippets: https://github.com/roboflow/roboflow-api-snippets/pull/15

11.4 Roboflow datasets

Loki: https://universe.roboflow.com/iadt/644e906ffe48cae1581b9c3c/model/3

Oxford Pets, combined wild animals, and Stanford backgrounds:
https://universe.roboflow.com/iadt/cats-dogs-wild-animals-stan-bg/dataset/1

11.5 Roboflow support threads

Models referring to non-existent class labels:

- https://discuss.roboflow.com/t/model-trained-via-api-refers-to-non-existent-class-labels/1931

Annotated images added to dataset as unannotated:

- https://discuss.roboflow.com/t/images-annotations-being-added-via-api-arent-immediately-added-to-dataset/2091

11.6 Figma Files

Main Prototype:
https://www.figma.com/file/oPHnUHd1nGTQMEaTZfGg28/RAID?type=design&node-id=0%3A1&t=Hy1DIKhcgE57SVtC-1

Application Architecture: https://www.figma.com/file/SRXstoyZy8ssoUtGAjdHCB/RAID---app-architecture?type=whiteboard&node-id=0%3A1&t=p3JZbnkL89MImuG4-1

11.7 Sprint review forms

https://iadt-my.sharepoint.com/:f:/g/personal/n00193326_iadt_ie/ElXmI24YNBFFqSMQ-KXtwoQBLoXFZz1fjJUQZ3nrZ_zm3Q?e=YQpTwh