Real-Time Object Detection System for Drone-Based Search and Rescue

Connor Mattless

N00213409

Supervisor: Joachim Pietsch

Second Reader: Timm Jeschawitz

Major Project

DL836 BSc (Hons) in Creative Computing

Year 4

1 Abstract

Recent advancements in Artificial Intelligence (AI), Machine Learning (ML), Deep Learning, and Object Detection have transformed various industries by enabling efficient, cost-effective, and automated processes. These technologies hold huge potential to enhance Search and Rescue (SAR) operations, particularly in critical tasks such as victim identification, location tracking, and hazard detection. This project explores the integration of AI in SAR scenarios, focusing on their capacity to improve operational accuracy and efficiency.

2 Acknowledgements

I would like to express my sincere gratitude to my supervisor, Joachim Pietsch, and my second reader, Timm Jeschawitz, for their invaluable guidance, constructive feedback, and dedicated support that helped bring this project to fruition.

I also appreciate the insightful contributions and encouragement of my colleagues at the UCD Centre for Cybersecurity & Cybercrime Investigation throughout the development of this project.

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: Connor Mattless

Signed: [illegible]

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

3 Table of Contents

1 Abstract 1

2 Acknowledgements 2

3 Table of Contents 4

4 Table of Figures 7

5 Introduction 9

6 Research 11

6.1 Introduction 11

6.2 Object Detection 11

6.2.1 What is Object Detection? 11

6.2.2 Types of Object Detection 16

6.3 Object Tracking 17

6.3.1 What is Object Tracking? 17

6.3.2 Types of Object Tracking 18

6.4 Applications of Object Detection & Tracking in SAR 19

7 Requirements 22

7.1 Introduction 22

7.2 Requirements Gathering 22

7.2.1 Similar Applications 22

7.2.2 User Requirements 26

7.2.3 Functional Requirements 26

7.2.4 Non-Functional Requirements 26

7.2.5 Use Case Diagram 27

7.2.6 Conclusion 27

8 Design 28

8.1 Program Design 28

8.1.1 Technologies 28

8.1.2 Design Patterns 32
8.1.3 Application Architecture 33
8.1.4 User Interface Design 33
8.1.5 Conclusion 34
9 Implementation 35
9.1 Introduction 35
9.2 Predevelopment 35
9.2.1 Virtual Environment 35
9.2.2 Continuous Integration Pipeline 35
9.3 Pre-recorded Video & OpenCV2 38
9.4 Initial Model Implementation 39
9.5 Improved Model 41
9.6 PySide 6 Video Player 43
9.7 Dialogs and Signals 46
9.7.1 Signals in PySide / Qt 46
9.7.2 Dialog handling 46
9.7.3 Implementation 47
9.8 DeepSort 47
9.8.1 What is DeepSort? 47
9.8.2 Why DeepSort? 48
9.8.3 How does the detection work? 49
9.9 Settings 50
9.9.1 Implementation 51
9.10 Multi-Threading 52
9.10.1 Threading & Processing 53
9.10.2 Multi-Threading Implementation 54
9.10.3 Multi-Processing Implementation 54

9.10.4 Threading Conclusion 55
9.11 Metadata Extraction 56
9.12 Archive Processor 58
9.13 Play / Pause 60
9.14 Close Video 62
9.15 Binary Compilation 64
9.15.1 Creating a binary 64
9.15.2 Automating Compilation 66
9.15.3 Conclusion 68
9.16 Implementation Conclusion 68
10 Testing 69
10.1 Unit Testing 69
10.2 Integration Testing 71
10.3 Manual Testing 72
11 Further Development 73
12 Conclusion 74
12.1 Final Words 75
13 Bibliography 76

4 Table of Figures

Figure 1: CNN diagram (MathWorks, n.d.) 13
Figure 2: Convolutional Layer (Modi, 2023) 14
Figure 3: Example batch output from CNN (Mattless, MP_video_processing, 2024)15
Figure 4: Object Detection v MOT (Klingler, 2023) 18
Figure 5: mAP50 results of Archangel dataset + YOLOv5 (Shen, et al., 2023)........ 20
Figure 6: USRI Loc8 (USRI, n.d.).......... 22
Figure 7: Loc8 Report generator (Aermatica3D, n.d.).......... 23
Figure 8: ADIAT Image Viewer (Texsar, 2024).......... 24
Figure 9: MRMap Map view (Brookes, 2008) 25
Figure 10: Use Case Diagram.......... 27
Figure 11: Ultralytics Hub 29
Figure 12: Google Colab 30
Figure 13: Block Diagram.......... 33
Figure 14: Main Page Wireframe 34
Figure 15: V Env Command 35
Figure 16: Dockerfile.......... 36
Figure 17: GitHub CI Yaml 37
Figure 18: Expanded Pipeline 38
Figure 19: Showing a video in Python 39
Figure 20: Notebook Model Training 40
Figure 21: Model Training Report.......... 41
Figure 22: Large Model mAP 42
Figure 23: 'Lite' Model mAP 43
Figure 24: Initial Video Player 44
Figure 25: VideoPlayer Class.......... 45
Figure 26: Dialog Handler Class 47
Figure 27: Dialog Handler Class Method.......... 47
Figure 28: YoloDeepSortProcessor 48
Figure 29: VideoPlayer Model Methods 49
Figure 30: Drawing a rectangle & ID 50
Figure 31: Settings Dialog Class 50
Figure 32: SettingsDialog Model Loading.......... 51
Figure 33: Multi-Processing Implementation 53
Figure 34: Multi-Processing Worker 55
Figure 35: MetadataProcessor 57
Figure 36: Metadata Viewer 58
Figure 37: VideoQueue Class 60
Figure 38: Adding Play / Pause Button.......... 61
Figure 39: toggle_play_pause Method 62
Figure 40: Closure signal 63
Figure 41: Close button QPushButton.......... 63

Figure 42: VideoPlayer close method.. 64
Figure 43: DroneLink Spec File.. 65
Figure 44: Binary Compilation Jobs 67
Figure 45: CI Test Job.. 70

5 Introduction

Over the past decade, the widespread availability of affordable and capable drones has revolutionised many fields, including environmental monitoring, agriculture, infrastructure inspection, and public safety. In particular, Search and Rescue (SAR) operations have begun integrating drones into their workflows to enhance search efficiency, improve crew safety, and expand operational reach. Drones serve as force multipliers, enabling SAR teams to cover large and often hazardous areas more quickly and with fewer personnel on the ground.

At the same time, advances in Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) have unlocked new capabilities in real-time image processing, object detection, and classification. These technologies are now mature enough to be embedded into resource-constrained systems such as drones, making it feasible to perform onboard analysis of video streams during flight. This integration creates the possibility for automated systems that can detect, classify, and track objects of interest—such as missing persons or potential hazards—in real-time.

Despite these advancements, many SAR operations remain heavily manual, often relying on human operators to visually scan live video feeds. This is not only time-consuming and fatiguing but also prone to human error, especially in low-visibility or high-stress environments. Additionally, the majority of object detection solutions available are either closed source, prohibitively expensive, or require significant post-processing effort, making them unsuitable for smaller organisations with limited resources.

This project seeks to address these limitations by developing an open, low-cost software solution—tentatively titled DroneLink—that enhances drone-based SAR capabilities through integrated real-time object detection and tracking. The application is designed to process both live and recorded video feeds, detect human objects using pretrained deep learning models (e.g., YOLO), and maintain tracking through DeepSORT-based multi-object tracking algorithms. Emphasis is placed on usability, performance, and modularity to ensure the system is accessible to SAR personnel with varying levels of technical expertise.

The primary objective is to provide a tool that can assist SAR teams in rapidly locating individuals, minimising search time, and reducing the cognitive load on operators. Additionally, the application supports exporting processed video and metadata for reporting or review, improving operational transparency and post-mission analysis.

By combining ML-based detection algorithms with an intuitive graphical interface and support for widely available hardware, this project demonstrates how modern AI technologies can be practically applied to life-saving applications in the field.

6 *Research*

The following section builds on research previously conducted in an earlier submitted report, 'Investigating Object Detection, Tracking & its Applications in Search and Rescue' (Mattless, 2024)

6.1 *Introduction*

In recent years, advancements in Artificial Intelligence (AI), Machine Learning (ML), and Object Detection have significantly increased their capabilities, driving transformative changes across many industries. These technologies enable processes that may have been time-consuming, costly, or required specialised expertise to be executed with remarkable efficiency, often autonomously, through the usage of neural networks. From identifying patterns in complex datasets to automating intricate tasks, AI and ML have the potential to reshape how entire industries operate, with object detection emerging as a key component in applications that require precise identification and localisation of objects within images or video streams.

This paper aims to explore the use cases of such technologies and their potential to greatly benefit Search and Rescue (SAR) operations. By utilising the advancements in AI, ML and Object Detection SAR mission efficiency could be improved through the more accurate identification and location of individuals in distress, especially in challenging environments or adverse conditions. Additionally, the integration of tracking algorithms enables the constant monitoring of detection objects, giving SAR crews constant, real-time updates on their location, movement and condition.

This paper investigates the algorithms and how they function, their practical applications and the challenges they would encounter in a SAR context.

6.2 *Object Detection*

6.2.1 *What is Object Detection?*

Object detection is a computer vision technology that combines object classification and localisation to identify and position objects within an image or video. It involves recognising the type (class) of objects (e.g., car, person) and determining its exact location within a scene. Object detection models typically employ feature extraction techniques and neural network architecture like Convolutional Neural Networks (CNNs) to process visual data and make predictions. Applications range from autonomous vehicles such as cars and drones to surveillance and healthcare. (Murel & Kavlakoglu, 2024)

6.2.1.1 How does object detection work?

Modern object detection uses ‘Deep Learning’ which is a subset of Machine Learning.

Machine learning has manually selected features that are curated by the person training the model, this can be effective with smaller datasets and has the benefit of being less computationally intensive. Additionally, Machine learning algorithms are trained and used on structured or tabular (e.g., Excel Spreadsheets) making them useful for uses like recommender systems or predictive analytics (e.g., sales forecasting).

Deep Learning (DL) is a subset of ML that uses neural networks with layers. These are designed to find, learn and combine features from raw data without manual intervention.

The most common neural networks used in object detection are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) (Carmatec, 2024).

6.2.1.1.1 Convolutional Neural Networks

A CNN is composed of a series of layers. Input, output, and many hidden layers in between.

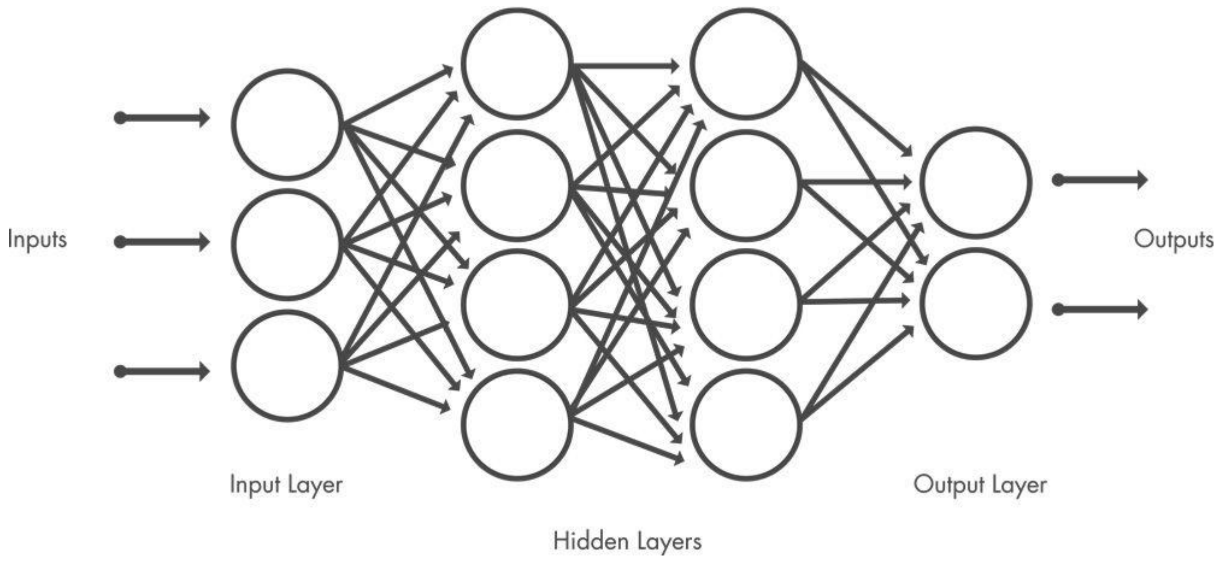


Figure 1: CNN diagram (MathWorks, n.d.)

CNNs first apply filters (which are small samples of the input) to an input image to detect features like simple edges, textures or shapes. The filters are applied to the whole image in a convolution operation. A convolutional operation involves sliding these filters across the input image, multiplying corresponding pixel values at each position, and then summing the results to produce a single output value. The convolution operation can then combine features to create complex or abstract feature detection. These features found in the filter are then combined with the input to create a feature map.

Between the convolutional layers are pooling layers, they are responsible for downsizing the feature maps to reduce the number of parameters, controlling overfitting and reducing weight oscillations. Pooling layers work by summarising regions of the feature map. Typically, this is done by taking the max pooling or the average pooling pixel values within each region. This reduces the spatial dimensions, improving the computational efficiency, and helps to better generalise the model by only keeping the most relevant information.

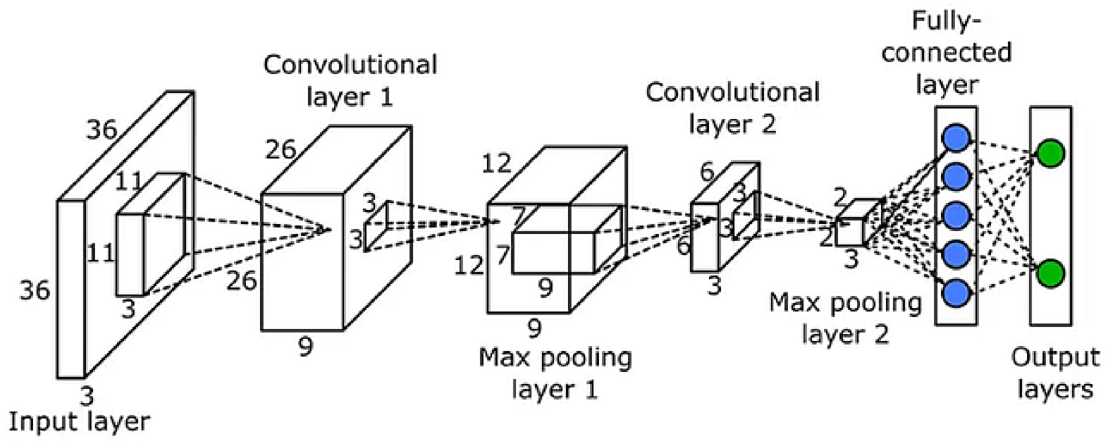


Figure 2: Convolutional Layer (Modi, 2023)

Next the activation layer applies a non-linear activation function, such as Rectified Linear Unit (ReLu) to the output of the pooling layer. This allows the model to learn more complex representations of the input data.

A normalisation layer is then applied. The normalisation layer adjusts the scale of the input data to keep the range of data values consistent. This helps prevent extreme values from effecting the output data and ensures that all inputs are treated more equally. Typically, in CNNs batch normalisation is used where a mean is calculated across the input batch. This can reduce the time it takes for a model to find a good solution during the training, further reduce overfitting and allows for larger learning rates as the model will be less sensitive.

The input is then run through a dropout layer that randomly disables (drops out) neurons during the training. This benefits the model by ensuring that the model doesn't simply memorise the training data but instead generalises.

Finally, a dense layer is run. The dense layer combines the features and makes a final prediction. The activations from the previous layers are flattened and passed as inputs to the dense layer which then produces a final output. (Modi, 2023)

The output in terms of object detection would be a bounding box around feature with a selected class (e.g., person, car, etc)

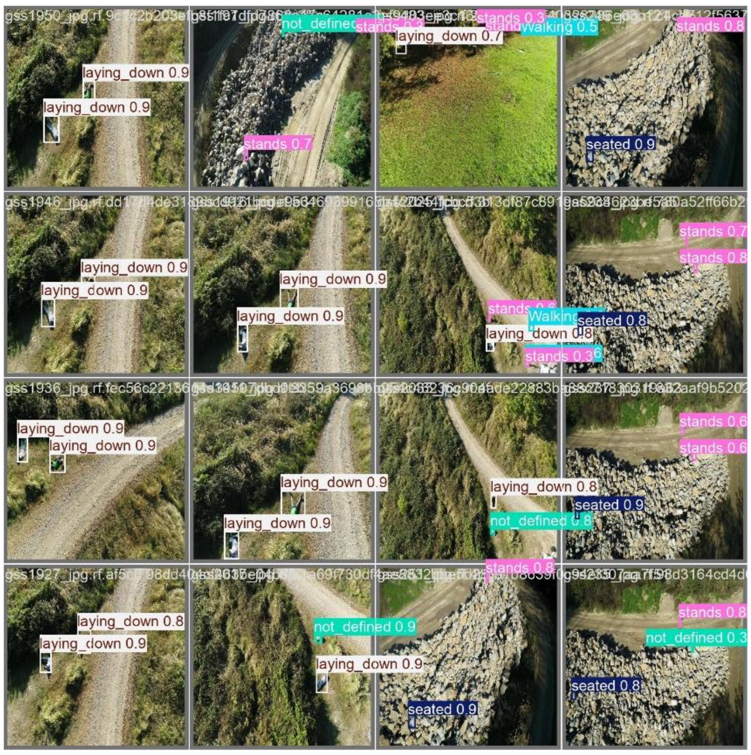


Figure 3: Example batch output from CNN (Mattless, MP_video_processing, 2024)

6.2.1.1.2 Recurrent Neural Networks

RNNs are specifically designed to process sequential or temporal data, such as text, audio, or video, which makes them distinct from CNNs. While both RNNs and CNNs share basic neural network elements like layers and weights (as shown in Figure 1), RNNs introduce a unique 'hidden state' mechanism. This hidden state acts as memory, enabling the model to retain information from previous steps in the sequence. This feature makes RNNs ideal for tasks where the order and context of inputs are important. (GeeksForGeeks, 2024).

There are various types of RNNs, each suited to different tasks. For tracking objects in a video, a Many-to-Many (N-to-N) RNN is the most appropriate choice, as it processes each frame of the video sequentially and generates an output for every input. There are many types of RNN, some of which blur the line between RNN and CNN, but that is out of scope for this paper. (GeeksForGeeks, 2024).

An RNN works by processing each element of the input sequence (e.g., a frame from a video or a word in a sentence) one at a time. This sequential approach allows the RNN to capture the temporal relationships between elements. If the input data is textual or categorical, it is first converted into numerical representations, a process known as feature representation. (GeeksForGeeks, 2024).

At the start of the sequence, the hidden state is initialised as a vector of zeros. For each input in the sequence, the RNN updates its hidden state based on the current input and the previous hidden state. This step is repeated for every element in the sequence, progressively accumulating information from all previous states. (GeeksForGeeks, 2024).

The final hidden state is then used to compute the network's output, which is compared to the actual target output to calculate the error. To train the RNN, the model uses Backpropagation Through Time (BPTT), a method that backpropagates the error through each time step to update the network's weights and improve performance (GeeksForGeeks, 2024).

6.2.2 *Types of Object Detection*

Modern object detection techniques can be broadly categorised into two categories, anchor and non-anchor-based methods. (Amjoud & Amrouch, 2023)

Anchor based algorithms can be further categorised into two categories. One-stage detectors and two-stage detectors. Both of these detectors utilise boundaries around potential objects.

Two-stage detectors split the detection process into a two-step process: Region Proposal and Classification, with the classification step containing some location refinement.

Region proposal identifies regions of interest (ROIs) in the feature map that are likely to contain objects. The second stage, Classification, processes these ROIs and attempts to classify the objects and refine the bounding boxes.

Two-stage detectors are known for their high accuracy in both localisation and recognition but are generally unsuitable for real-time processing as they require significant compute power. Some examples of two-stage detectors are R-CNN, Faster R-CNN and Mask R-CNN.

One-stage detectors combine the Region Proposal and Classification stages into a single stage. This has the effect of making the model slightly less accurate but much faster allowing it to be used in real-time processing. The most notable example of a one-stage detector is YOLO (You Only Look Once) (Jiao, et al., 2019)

Non-anchor-based techniques remove the need for predefined bounding boxes, they instead predict the centre point of an object in the image and directly regress the objects size and shape. This approach to object detection reduces the computational complexity and avoids errors relating to the anchor box design.

There are also transformer bases approaches to object detection that have recently been developed but these approaches require large datasets and significant computational power in return for a simpler and more unified detection pipeline (Amjoud & Amrouch, 2023).

6.3 *Object Tracking*

6.3.1 *What is Object Tracking?*

Object tracking is the process of identifying and continuously tracking objects across a series of frames in a video stream after the object has been initially detected. In the context of video tracking, a neural network processes sequential frames and estimates the object's current and previous locations, maintaining continuity as long as possible.

The tracking process typically involves several key steps. The first is target initialisation, where the object of interest is defined. A bounding box is drawn around the object in the initial frame, and the tracker uses this reference to locate and track the object in subsequent frames, updating the bounding box as needed.

Next, the tracker performs appearance modelling, which focuses on capturing the visual features of the object to handle variations in appearance caused by changes in lighting, angle, or scale. This is analogous to feature extraction and helps the tracker maintain accuracy across frames. (Barla, 2021)

Motion prediction is then used to estimate the object's trajectory based on its previous positions. Techniques such as Kalman filters or particle filters can be employed to predict the object's likely position in the next frame, facilitating smooth and reliable tracking. (Barla, 2021)

Finally, object association ensures that the object detected in the current frame is correctly matched with the same object from previous frames. This step is crucial for maintaining tracking continuity, even in cases where the object undergoes occlusion, changes appearance, or momentarily exits the frame. (Barla, 2021)

Object tracking has numerous real-world applications, such as in security, where it can be used to monitor individuals across multiple cameras in a surveillance system. (Barla, 2021)

6.3.2 *Types of Object Tracking*

At a high level there are two types of object tracking: Single Object Tracking (SOT) and Multi-Object Tracking (MOT).

MOT differs from general object detection in that instead of just trying to classify each object in a scene it will also assign a unique ID to each object to distinguish each discrete object even if they have the same class.

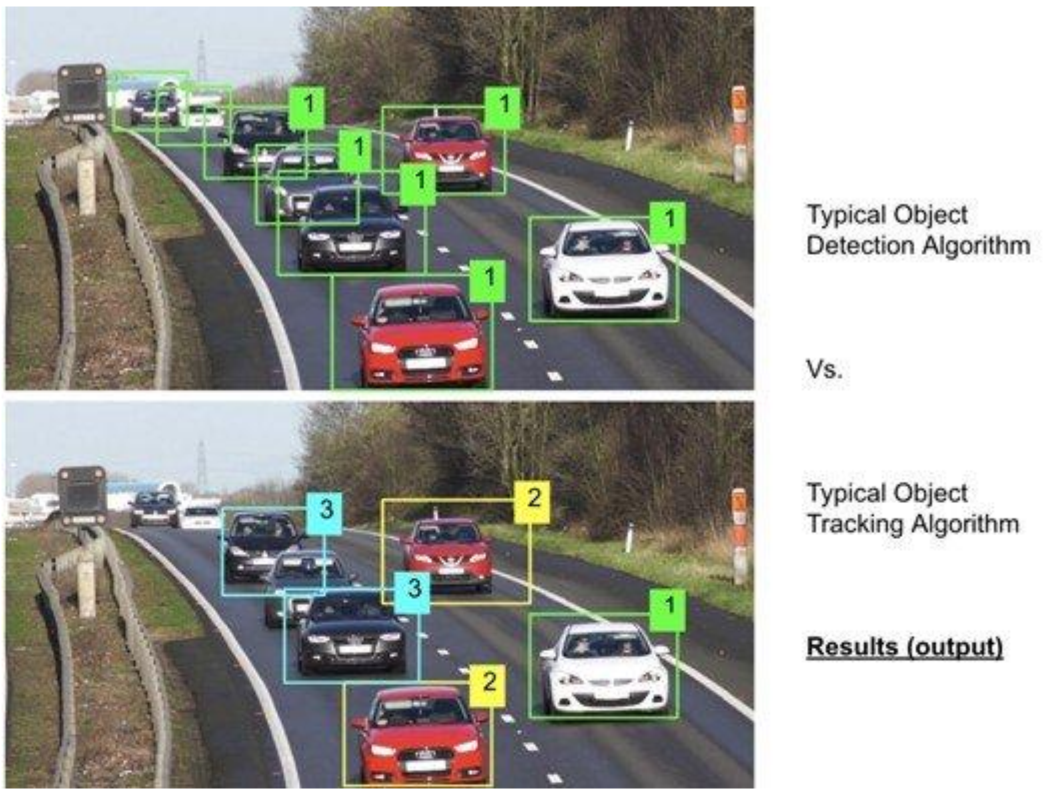

Typical Object Detection Algorithm
Vs.
Typical Object Tracking Algorithm
Results (output)
1
2
3

Figure 4: Object Detection v MOT (Klingler, 2023)

As seen above in Figure 4, a general object detection algorithm just sees ‘car’ whereas a MOT is attempting to identify different cars as being different from each other. (Klingler, 2023)

Single Object Tracking, sometimes called Visual Object Tracking, focuses on continuously tracking a single object through an image sequence after its initial detection. The object is initialised in the first frame. SOTs are designed to handle changes in the object’s appearance, size and orientation as well as challenges such as lighting conditions and occlusion. SOT is designed for real-time processing and as such it is used in applications such as security systems for tracking individuals or for tracking hand / eye movements.

A drawback of SOT is that it cannot handle new objects appearing in the middle of a sequence (Klingler, 2023).

6.4 *Applications of Object Detection & Tracking in SAR*

There are a number of areas that object detection & tracking can greatly benefit search and rescue operations and crews. Over the last 15 years drones have gone from being a prototype technology in the SAR field to becoming a staple piece of equipment for rescue workers. The advancements and miniaturisation of optical and Forward Looking Infrared (FLIR) cameras mean that a single person portable drone can carry a whole suite of sensors as well as perform autonomous actions whilst reporting to a ground station or controller. A common usage of drones is to utilise an on-board FLIR camera to locate victims in the wilderness. (Dukowitz, 2024).

In parallel to civilian and humanitarian applications, drones are increasingly being utilised in combat. Most notably in the ongoing conflict in Ukraine. There, commercial and military grade drones have been repurposed for reconnaissance, target acquisition and even direct engagement. These drone systems often rely on video feeds and manual control. Integrating real-time object detection and tracking software could offer significant advantages to the platform. For instance, automated identification and geolocation of opposing forces or equipment could streamline intelligence gathering pipelines and reduce operator workload. The same core technologies developed for SAR use could be adapted to support autonomous targeting, situational awareness, and mission planning in complex combat environments.

Integrating object detection and tracking would add yet another powerful dimension to the drone-based camera system by allowing the possibility for the drone to autonomously find and report potential victims. Combining the object tracking with the drones integrated GPS data would allow the system to report potential areas for crews to investigate, which could improve efficiency in the allocation of SAR resources and time (McGee, Mathew, & Gonzalez, 2020).

SAR does present a number of challenges to the usage of object detection and tracking, particularly as SAR teams frequently operate in environments that may have adverse conditions or in environments where poor thermal contrast or the target could be partially occluded. Addressing these issues presents a significant challenge as it requires advanced model training due to the number of variables present in drone imagery. Training the model on datasets specifically designed for the purpose would be required. In research conducted into the usage of human detection from drones using YOLO models with a specifically created and labelled data set called '*Archangel*' with significant fine tuning found a max detection rate of around 75 mAP50 for standing poses but significantly less for other poses. (Shen, et al., 2023). mAP50 refers to the mean Average Precision over the threshold of 50%. This metric measures how accurately a model detects objects by comparing the predicted bounding boxes to the ground truth boxes. A higher mAP value indicates better performance. In this case a score of 75 mAP50 indicates that the model correctly identified and localised objects with at least 50% overlap 75% of the time.

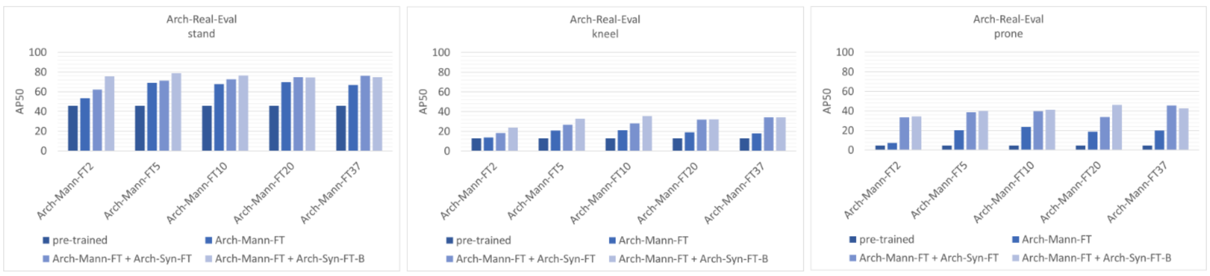


Figure 5: mAP50 results of Archangel dataset + YOLOv5 (Shen, et al., 2023)

Recent advancements in thermal imaging, particularly with YOLOv5 models, have demonstrated robust performance in detecting individuals in challenging environments. For example, drones equipped with thermal cameras can detect individuals in harsh conditions such as low-light or occluded areas. In research involving simulated SAR missions, thermal object detection combined with multiple-target tracking achieved high tracking precision under adverse conditions. However, challenges like track breakage due to rapid drone movements or occlusions remain, emphasizing the need for refined detection models and better dataset diversity (Yeom, 2024).

Aside from victim identification, object detection can be used for hazard detection and avoidance for ground crews, enabling the better planning of routes when moving or to identify hazards in disaster zones such as active fires or flooding. Detecting potential debris and hazards for SAR operators has the potential to enhance both the safety and effectiveness of SAR teams and operations (Nehete, Dharrao, Pise, & Bongale, 2024).

Environmental factors such as occlusions and weather conditions can greatly degrade the quality and accuracy of the detections.

In conclusion, the integration of detection and tracking algorithms would represent a significant leap in SAR capabilities and has the potential to save lives in real world conditions. I believe that there should be future efforts into researching and refining these technologies to expand and improve their use cases in this field.

7 Requirements

7.1 *Introduction*

The purpose of the requirements phase was to allow the discovery of what the application should be able to do and what the end user would find essential and what would be classified as a ‘nice-top-have’

From the research it as concluded that development should proceed in Python as the ‘Pythonic’ workflow appeared to have the best documentation and available tools. The development of the application would include the creation of a GUI as well as training models to process the video. The model would be based on Ultralytics’ YOLO series of machine learning models.

7.2 *Requirements Gathering*

7.2.1 *Similar Applications*

7.2.1.1 Loc8

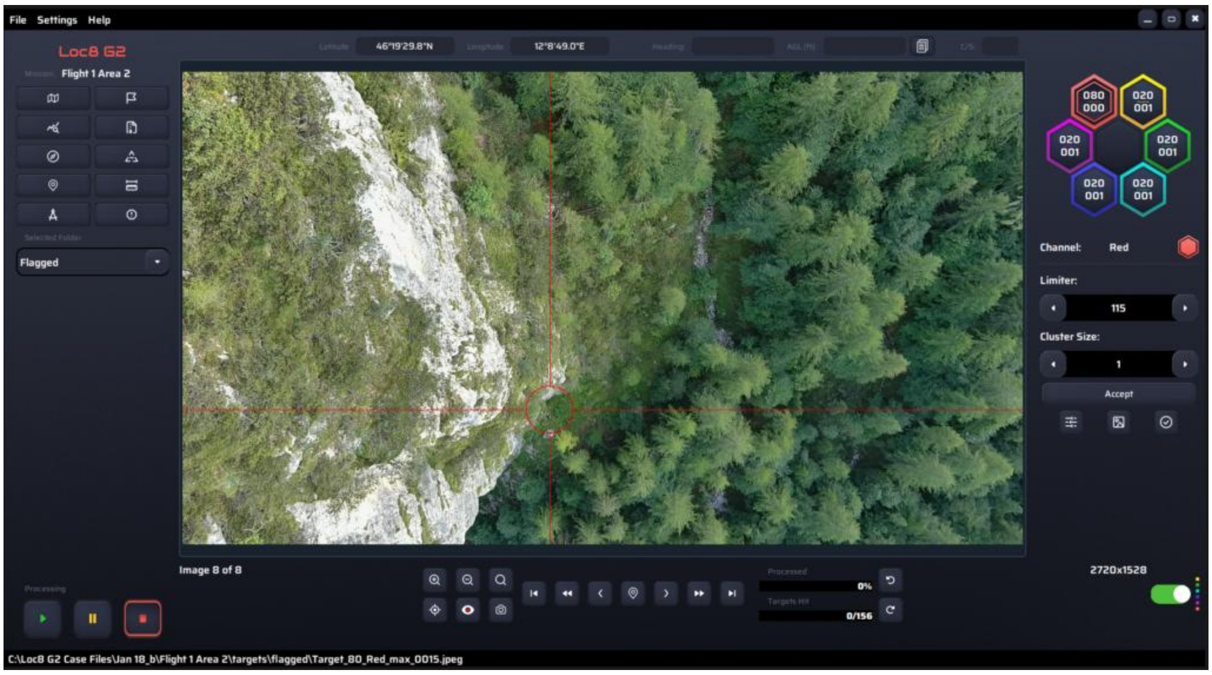
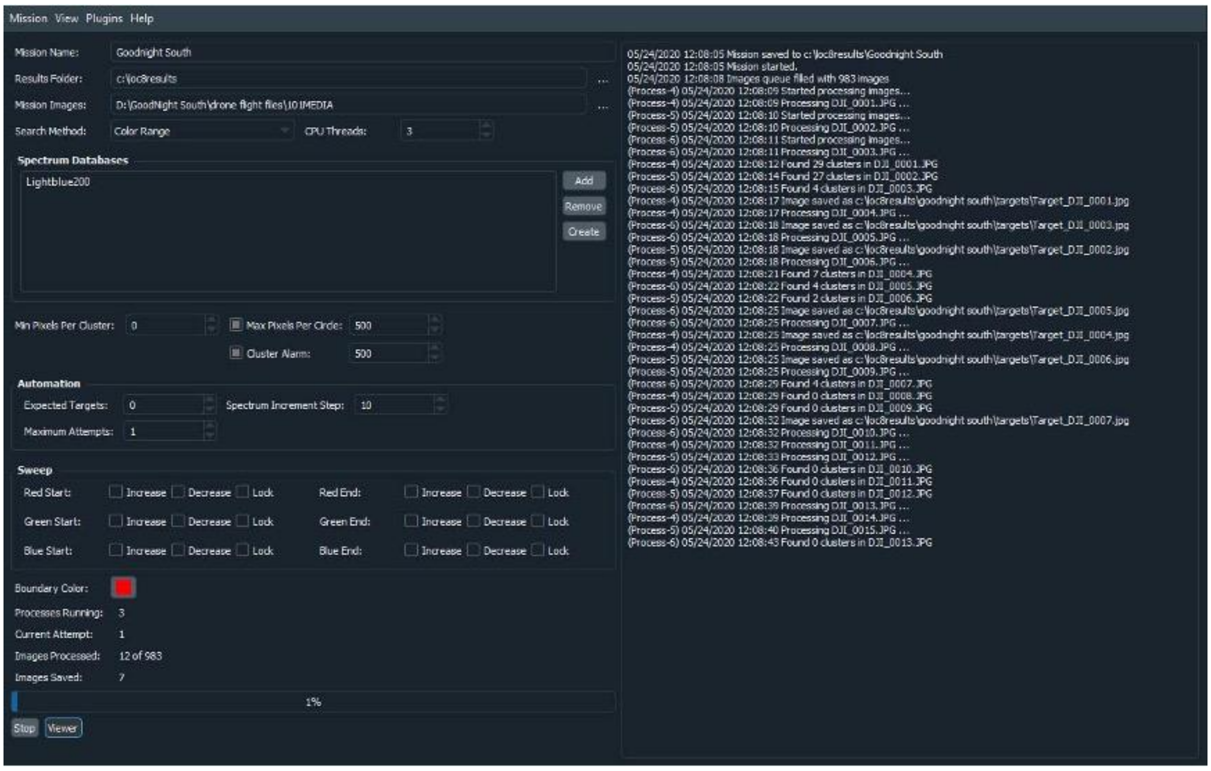


Figure 6: USRI Loc8 (USRI, n.d.)

Loc8 is a software package developed by USR (“Unmanned Systems Research”) that allows a user to scan images from drones to allow the location people or other objects. Loc8 does not use machine learning in its approach to this task. Instead, using pixel analysis to identify the target using colour and other information.



Mission Name: Goodnight South
Results Folder: c:\loc8results
Mission Images: D:\GoodNight South\drone flight files\10 IMEDIA
Search Method: Color Range
CPU Threads: 3
Lightblue200
Min Pixels Per Cluster: 0
Max Pixels Per Circle: 500
Cluster Alarm: 500
Expected Targets: 0
Spectrum Increment Step: 10
Maximum Attempts: 1
Red Start: ☐ Increase ☐ Decrease ☐ Lock
Red End: ☐ Increase ☐ Decrease ☐ Lock
Green Start: ☐ Increase ☐ Decrease ☐ Lock
Green End: ☐ Increase ☐ Decrease ☐ Lock
Blue Start: ☐ Increase ☐ Decrease ☐ Lock
Blue End: ☐ Increase ☐ Decrease ☐ Lock
Boundary Color: 🟥
Processes Running: 3
Current Attempt: 1
Images Processed: 12 of 983
Images Saved: 7
1%

```
05/24/2020 12:08:05 Mission saved to c:\loc8results\Goodnight South
05/24/2020 12:08:05 Mission started.
05/24/2020 12:08:08 Images queue filled with 983 images
(Process-4) 05/24/2020 12:08:09 Started processing images...
(Process-4) 05/24/2020 12:08:09 Processing DJI_0001.JPG ...
(Process-5) 05/24/2020 12:08:10 Started processing images...
(Process-5) 05/24/2020 12:08:10 Processing DJI_0002.JPG ...
(Process-6) 05/24/2020 12:08:11 Started processing images...
(Process-6) 05/24/2020 12:08:11 Processing DJI_0003.JPG ...
(Process-4) 05/24/2020 12:08:12 Found 29 clusters in DJI_0001.JPG
(Process-5) 05/24/2020 12:08:14 Found 27 clusters in DJI_0002.JPG
(Process-6) 05/24/2020 12:08:15 Found 4 clusters in DJI_0003.JPG
(Process-4) 05/24/2020 12:08:17 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0001.jpg
(Process-4) 05/24/2020 12:08:17 Processing DJI_0004.JPG ...
(Process-6) 05/24/2020 12:08:18 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0003.jpg
(Process-6) 05/24/2020 12:08:18 Processing DJI_0005.JPG ...
(Process-5) 05/24/2020 12:08:18 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0002.jpg
(Process-5) 05/24/2020 12:08:18 Processing DJI_0006.JPG ...
(Process-4) 05/24/2020 12:08:21 Found 7 clusters in DJI_0004.JPG
(Process-6) 05/24/2020 12:08:22 Found 4 clusters in DJI_0005.JPG
(Process-5) 05/24/2020 12:08:22 Found 2 clusters in DJI_0006.JPG
(Process-6) 05/24/2020 12:08:25 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0005.jpg
(Process-6) 05/24/2020 12:08:25 Processing DJI_0007.JPG ...
(Process-4) 05/24/2020 12:08:25 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0004.jpg
(Process-4) 05/24/2020 12:08:25 Processing DJI_0008.JPG ...
(Process-5) 05/24/2020 12:08:25 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0006.jpg
(Process-5) 05/24/2020 12:08:25 Processing DJI_0009.JPG ...
(Process-6) 05/24/2020 12:08:29 Found 4 clusters in DJI_0007.JPG
(Process-4) 05/24/2020 12:08:29 Found 0 clusters in DJI_0008.JPG
(Process-5) 05/24/2020 12:08:29 Found 0 clusters in DJI_0009.JPG
(Process-6) 05/24/2020 12:08:32 Image saved as c:\loc8results\goodnight south\targets\Target_DJI_0007.jpg
(Process-6) 05/24/2020 12:08:32 Processing DJI_0010.JPG ...
(Process-4) 05/24/2020 12:08:32 Processing DJI_0011.JPG ...
(Process-5) 05/24/2020 12:08:33 Processing DJI_0012.JPG ...
(Process-6) 05/24/2020 12:08:36 Found 0 clusters in DJI_0010.JPG
(Process-4) 05/24/2020 12:08:36 Found 0 clusters in DJI_0011.JPG
(Process-5) 05/24/2020 12:08:37 Found 0 clusters in DJI_0012.JPG
(Process-6) 05/24/2020 12:08:39 Processing DJI_0013.JPG ...
(Process-4) 05/24/2020 12:08:39 Processing DJI_0014.JPG ...
(Process-5) 05/24/2020 12:08:40 Processing DJI_0015.JPG ...
(Process-6) 05/24/2020 12:08:43 Found 0 clusters in DJI_0013.JPG
```

Figure 7: Loc8 Report generator (Aermatica3D, n.d.)

7.2.1.1.1 Advantages

Loc8 has many advantages. It allows its users to quickly scan images produced by drones and generate a map of the located target from the image. Loc8 also allows the generation of a report that outlines the actions taken in the application and it's processing of images. Due to the low hardware requirements of the pixel analysis and the no requirement for an internet connection it is ideal for teams that may not have the best computing equipment or limited access to the Internet.

7.2.1.1.2 Disadvantages

Loc8 also has some notable disadvantages to its use. Primarily it is closed source and requires a licence to use. This means that many smaller organisations may not be able to access the tool due to budgetary constraints. Larger organisations may require that such software is "open source, closed community" meaning that law enforcement and other similar agencies can inspect the source code before deciding to use the tool.

Additionally, Loc8 has no facility to accept live video from a drone and is solely for processing images after-the-fact.

7.2.1.2 Texsar ADIAT

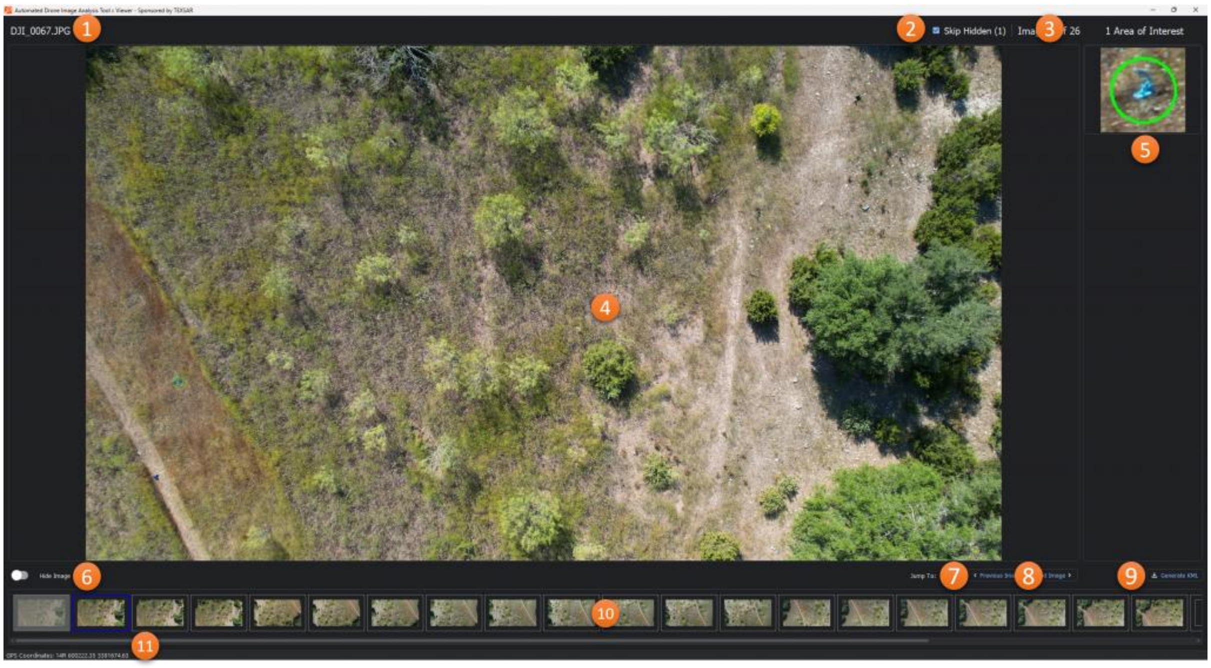


Figure 8: ADIAT Image Viewer (Texsar, 2024)

Automated drone image analysis tool (ADIAT) developed by Texsar. Similar to Loc8 ADIAT allows the processing of images from drone footage using pixel colour data.

7.2.1.2.1 Advantages

ADIAT is feature rich allowing the user to fine tune the processing of the images though various means such as altering the minimum object area, the colour to search for, or even changing the algorithm that is used to search the image for objects. If the user also inputs a ‘SRT’ type file into the application, then location data can be processed alongside the images allowing for geolocation. The geolocation information can then subsequently be exported to a KML file for use on Google Earth and similar programs.

7.2.1.2.2 Disadvantages

ADIAT, similar to Loc8 can only process images or videos that have been uploaded after the flight of the drone has finished. Because of the configurability of the tool, it also has a steep learning curve that may require end users to receive training on the tool to be properly utilized as the detection algorithms are required to be tuned by the user.

7.2.1.3 MRMap

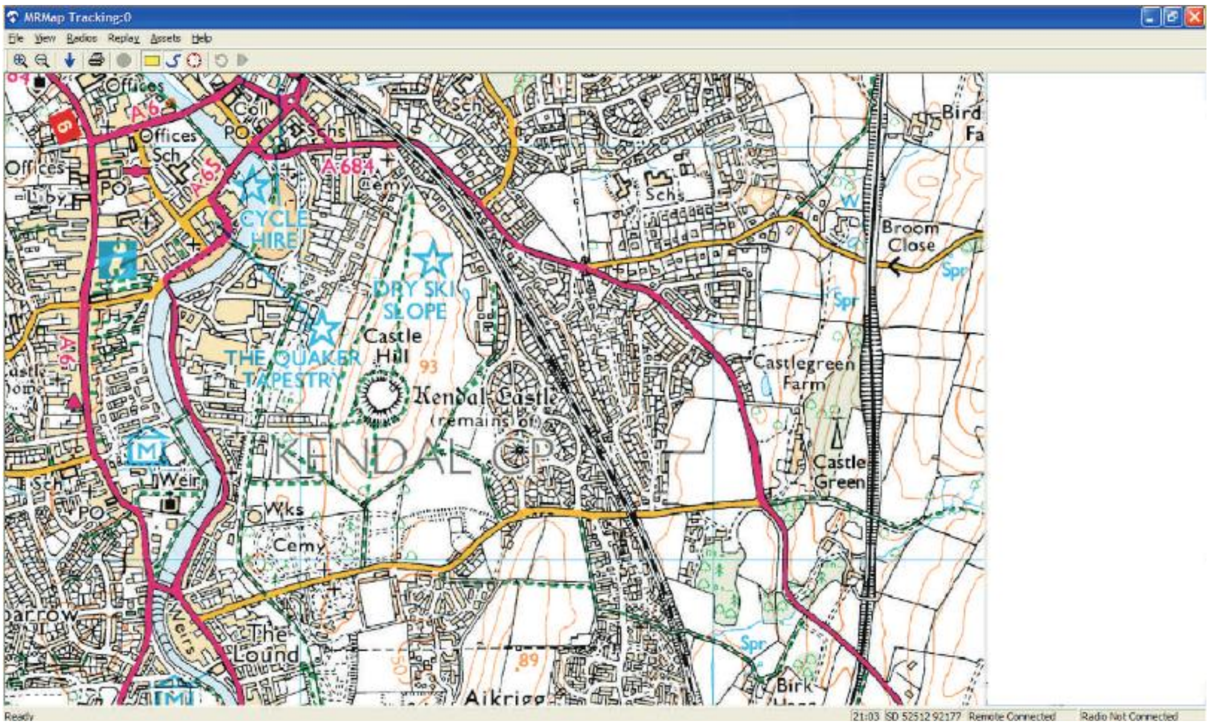

MRMap Tracking:0
File View Radios Replay Assets Help
Ready
21:03 SD 52512 92177 Remote Connected
Radio Not Connected

Figure 9: MRMap Map view (Brookes, 2008)

MRMap was “conceived and developed by Mountain Rescue Team personnel in the Lake District (UK).” (MRMap, n.d.) It is currently in widespread use within SAR teams across the UK and Ireland.

7.2.1.3.1 Advantages

MRMaps allows for near real-time tracking of SAR assets via radio pings meaning that people and assets can be tracked even in areas where there is limited internet access.

The ability to receive real-time data from drones as well as set waypoints and search areas for drones gives SAR teams a huge capability, allowing them to search areas in a hands-off way.

7.2.1.3.2 Disadvantages

Similar again to the previous tools mentioned, the real-time analysis works by processing pixel colour data which requires the user to know what the colour of their target may be, possibly leading to issues with detection.

Because the tool has been in development for almost 30 years there are many systems that users may struggle to integrate with the tool due to the code base being reliant on legacy systems and packages.

7.2.2 *User Requirements*

As part of my requirements gathering, I reached out to Dublin Wicklow Mountain Rescue Team (DWMRT) as well as members of An Garda Síochána (AGS). Unfortunately, my contact with DWMRT fell through as they were not able to allocate the resources I would need to conduct proper research into their specific use case in the timeframe that this project required. I did, however, manage to speak to some active and former members of AGS in an unofficial capacity in regard to what features that they think would be useful to support similar mission profiles.

7.2.3 *Functional Requirements*

- Realtime Video Processing: The application should be able to take in a video source from the machine it is running on and run the model on the incoming video stream.
- Pre-recorded Video Processing: The application should be able to run the model on a video clip uploaded to the program from the local file system.
- Metadata Extraction: The application should be able to extract any available metadata from the video source.
- Video Archival: The program should be able to export the processed video back to the local filesystem with the processing information included in the file.
- Target Identification: The application should aid the user in identifying a potential target of interest.

7.2.4 *Non-Functional Requirements*

- Accurate: The program should be able to detect human objects with a ‘greater than chance’ probability.
- Performant: The program should perform at a usable framerate on a range of computer hardware that are commonly available.
- Usable: The program should have a user-friendly GUI that conforms to design norms.

- Reliable: The program should not be prone to crashing or performing unexpectedly when running.

7.2.5 ***Use Case Diagram***

To envisage what the user’s interactions with the program will be, a Use Case Diagram was created. The diagram displays the actions that a user can take within the program and what the outcome of those actions would be.

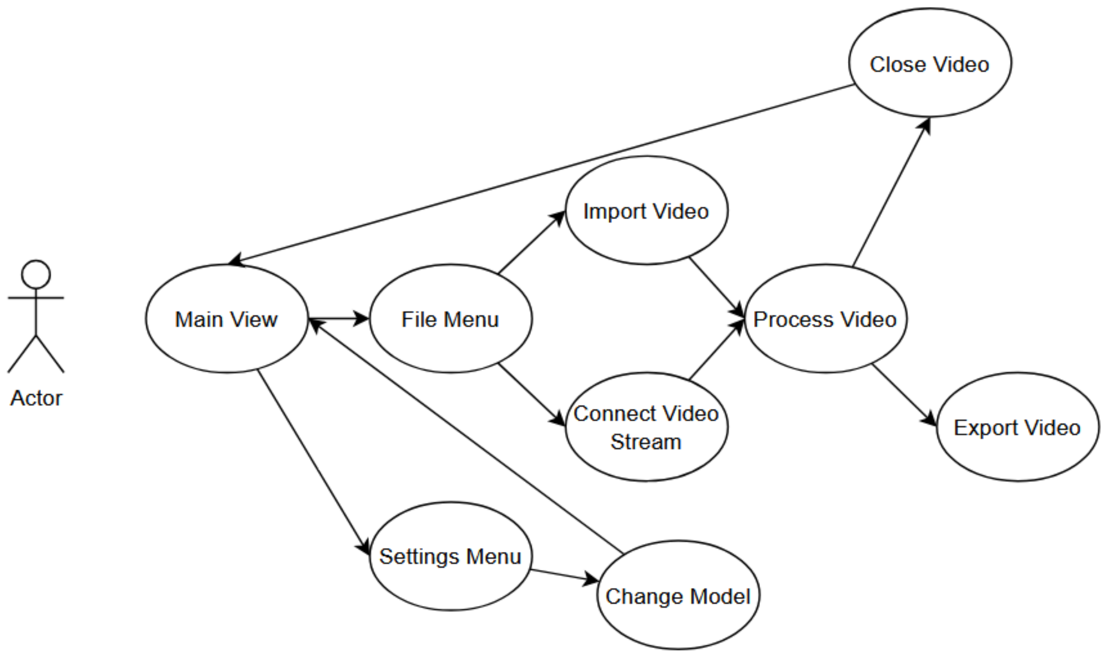


Figure 10: Use Case Diagram

7.2.6 ***Conclusion***

Having completed research into similar tools, spoken with industry professionals and planned how my application will flow it was now clearer how the program should proceed in relation to its features and what its users will need to find the software useful.

8 Design

The system was designed with a strong emphasis on modularity, performance, and ease of future expansion. Given the complex challenge of integrating real-time video processing, deep learning inference, and GUI responsiveness within a single application, consideration was given to both software and system architecture. The overall architecture uses a class-based, modular approach where modules such as video input, detection, metadata extraction, and user interface logic—are encapsulated in their own files. This modular approach allows for scalable, disconnected development, simplified testing, and straightforward maintenance.

8.1 *Program Design*

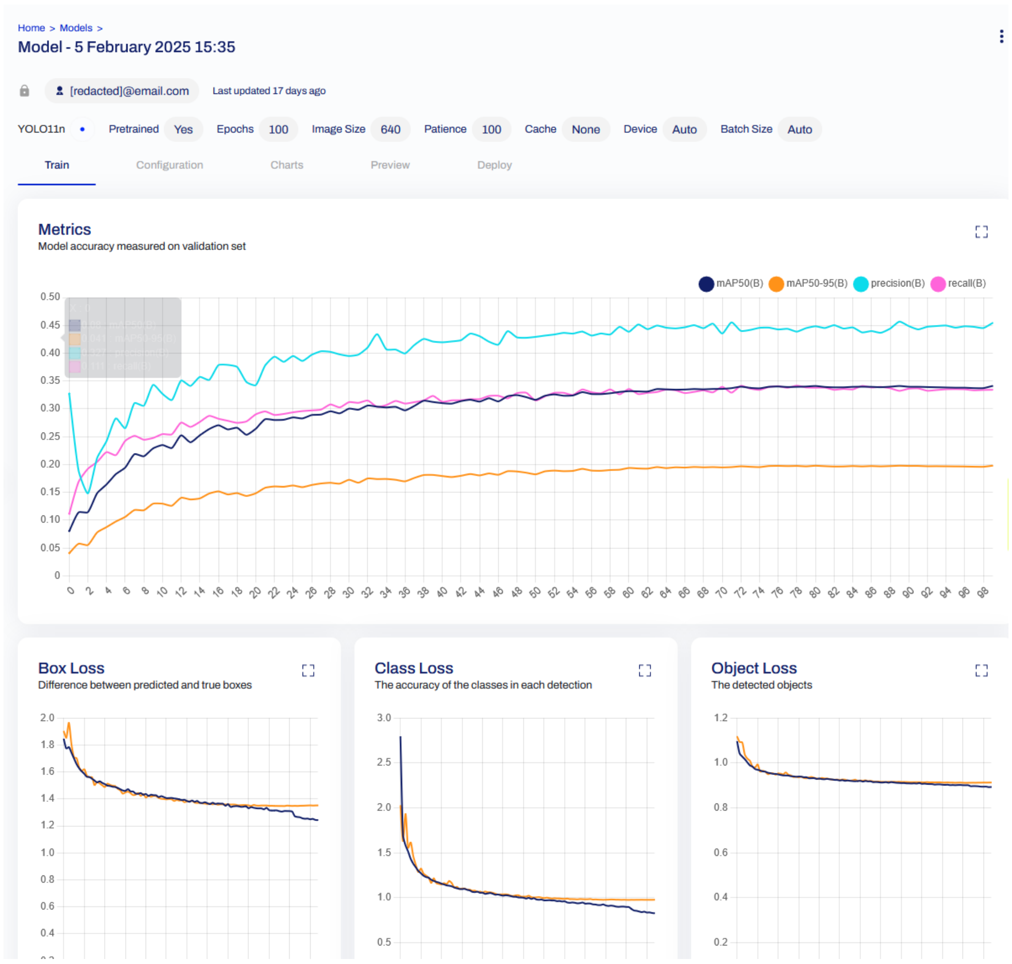
The primary application was written in Python 3.12 due to its extensive ecosystem of libraries, strong community support, and its native compatibility with PyTorch, OpenCV, and PySide. The decision to build the GUI in PySide6 (Qt for Python) was driven by its robust widget system, native feel, and extensive documentation.

8.1.1 *Technologies*

8.1.1.1 Visual Studio Code & PyCharm

For the development of the application, I chose to use Microsoft's Visual Studio Code (VS Code) and Jet Brain's PyCharm. I was using two code editors because they both had strengths and weaknesses as well as my development was done across multiple devices. I found VS Code better for rapid development where I would be writing large amounts of code as it has, in my opinion, a better development experience and quality of life features. On the other hand, PyCharm is specifically designed for Python development, and it has comprehensive debugging and package management tools that can make debugging easier.

8.1.1.2 *Ultralytics Hub*


Home > Models >

Model - 5 February 2025 15:35

[redacted]@email.com Last updated 17 days ago

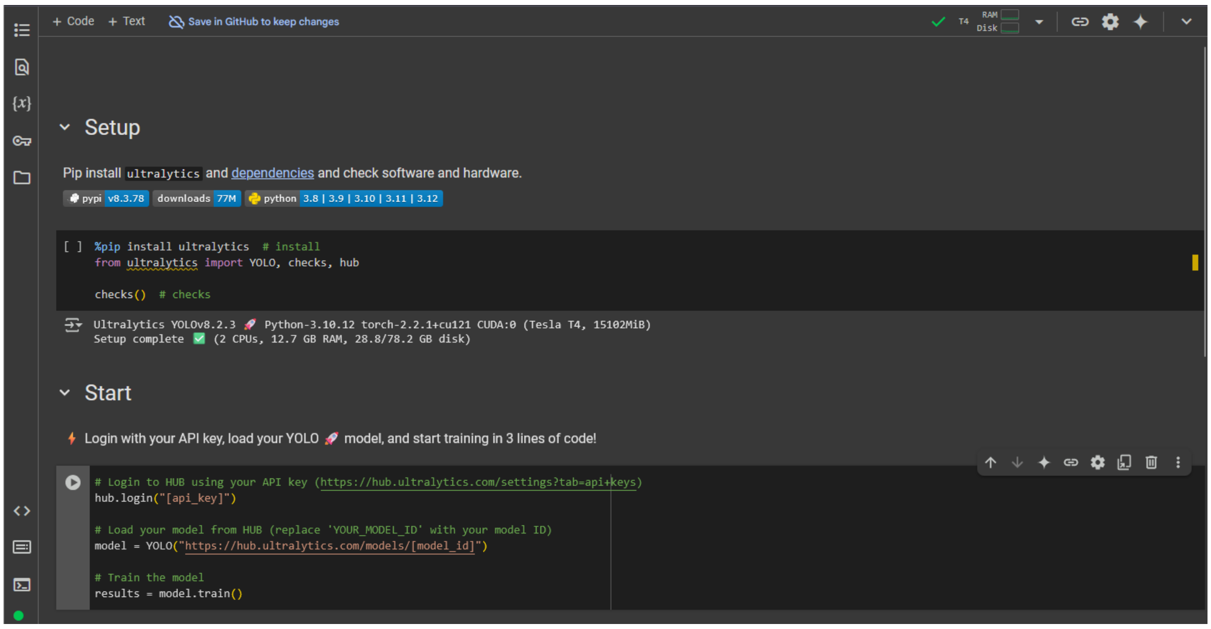
YOLO11n • Pretrained Yes Epochs 100 Image Size 640 Patience 100 Cache None Device Auto Batch Size Auto

Figure 11: Ultralytics Hub

Initially I was training my models purely locally on my machine. This proved to be an extremely time-consuming endeavour. To save development time I moved to using a cloud solution, Ultralytics Hub. This web interface provided my Ultralytics allowed me to select a desired model and then upload my dataset to it for training. Once the training had concluded I was able to download the weights from the model and use it on my local computer.

Ultralytics Hub also provides me with easy to digest data about the accuracy of my trained model, allowing me to compare between my model versions.

8.1.1.3 Google Colab



Setup

Pip install `ultralytics` and dependencies and check software and hardware.

pypi v8.3.78 downloads 77M python 3.8 | 3.9 | 3.10 | 3.11 | 3.12

```
%pip install ultralytics  # install
from ultralytics import YOLO, checks, hub

checks()  # checks
```

```
Ultralytics YOLOv8.2.3 🚀 Python-3.10.12 torch-2.2.1+cu121 CUDA:0 (Tesla T4, 15102MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 28.8/78.2 GB disk)
```

Start

⚡ Login with your API key, load your YOLO 🚀 model, and start training in 3 lines of code!

```
# Login to HUB using your API key (https://hub.ultralytics.com/settings?tab=api+keys)
hub.login("[api_key]")

# Load your model from HUB (replace 'YOUR_MODEL_ID' with your model ID)
model = YOLO("https://hub.ultralytics.com/models/[model_id]")

# Train the model
results = model.train()
```

Figure 12: Google Colab

Together with Ultralytics Hub I used Google Colab for the compute power behind the training of the model. Initially this was quite limited in its compute resources, although still faster than training locally. I decided to purchase the ‘Pro’ version of Colab as it greatly increased the compute power, I had available to me as well as allowed my training to continue when my computer was switched off, meaning I could train my models overnight.

8.1.1.4 Hyperparameter Tuning

To improve model performance and training efficiency, several hyperparameters were adjusted during the training permutations.

- **Batch Size**: Batch size was adjusted to fit within Colab Pro’s memory constraints. A batch size of 16 provided a good balance between accuracy and memory cost. Batch size affects how many images are processed at once during the training.
- **Epochs**: Epochs control how many times the model sees the full dataset. More epochs give the model more opportunity to learn, but too many lead to overfitting of the training dataset. During the training I initially used 50 epochs but later raised the number to 100 after observing underfitting in the validation.

- **Learning Rate**: I lowered the initial learning rate from 0.01 to 0.005 to stabilise early training convolutions. The learning rate determines how much the model's weights are adjusted during the training. A high learning rate can cause divergence which is when the model's loss (error) does not decrease and instead gets worse and worse over the course of the training.
- **Momentum**: The recommended momentum value of 0.937 was used, as recommended in YOLO configurations. Momentum helps the more retain passed gradient direction to smooth and accelerate training. This reduces the oscillation in the model's weights during training.

8.1.1.5 Docker / GitHub Actions

CI/CD pipeline environments are containerised and built with GitHub Actions to produce binary artifacts using PyInstaller for OS-specific deployment and automated testing.

8.1.1.6 DeepSORT

Used for Multi-Object Tracking (MOT). It assigns persistent IDs to objects across frames using appearance embeddings and Kalman filtering. This provides stable tracking of objects and reduces ID problems common in simple SORT algorithms.

8.1.1.7 OpenCV2

Employed for video decoding, image frame extraction, drawing bounding boxes, and basic video processing tasks. It provides low-level control over frame operations and is fully compatible with NumPy arrays and PyTorch tensors, making it well-suited for pre- and post-processing.

8.1.1.8 PySide6

Used for building the graphical user interface. It provides a full-featured widget set, layout management, and a signal-slot system to decouple GUI actions from backend logic. PySide6 is modern, Qt6-compatible, and has better support for event-driven programming than alternatives like Tkinter or wxPython.

8.1.2 *Design Patterns*

For this project, a modular, object-oriented approach was adopted. Each functional component of the application is encapsulated in its own class or group of classes. This is aided by an instance based pattern for modules like the detection processor, and a singleton pattern for shared state (e.g., frame queue management, video player). Key design principles include:

- **Signal-Slot Architecture:** All GUI interactions and user inputs are connected to backend logic using PySide6's built-in Signal and Slot mechanism. This promotes non-blocking behaviour and prevents direct coupling between GUI and logic layers.
- **Encapsulation and Separation of Concerns:** All major components, such as detection, processing, display, and storage are decoupled and communicate only through well-defined interfaces. This allows easier refactoring and allows independent unit testing.
- **Threading and Multiprocessing:** A hybrid threading-multiprocessing approach is used. Threads handle video capture and GUI responsiveness, while multiprocessing handles compute-heavy detection and frame rendering, taking full advantage of multi-core CPUs.

8.1.3 *Application Architecture*

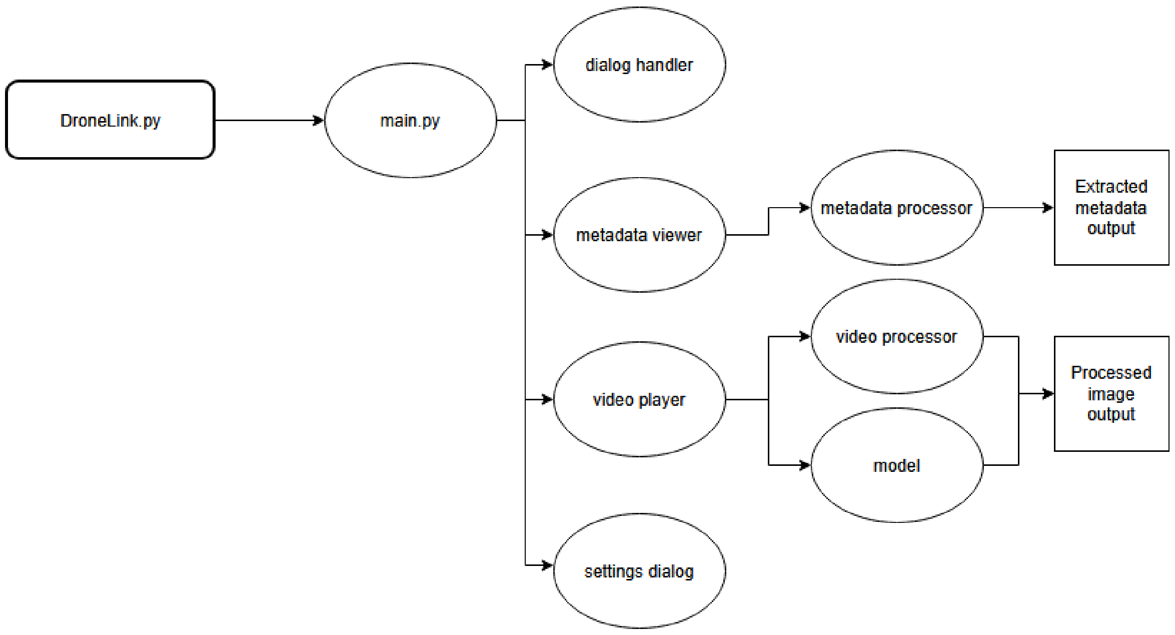


Figure 13: Block Diagram

8.1.4 *User Interface Design*

The inherent limitations present in PySide 6, particularly with styling, significantly constrained my design choices. This is due to the styling options defined by Qt, which restricts customisation using style sheets or other visuals that would otherwise be available in other frameworks. Despite these limitations development continued in PySide 6 because it is currently the most modern and well documented GUI package for Python.

To establish a foundation to develop upon I created some simple wireframes in Figma (Fig. 14). This would serve as the blueprint for the application.

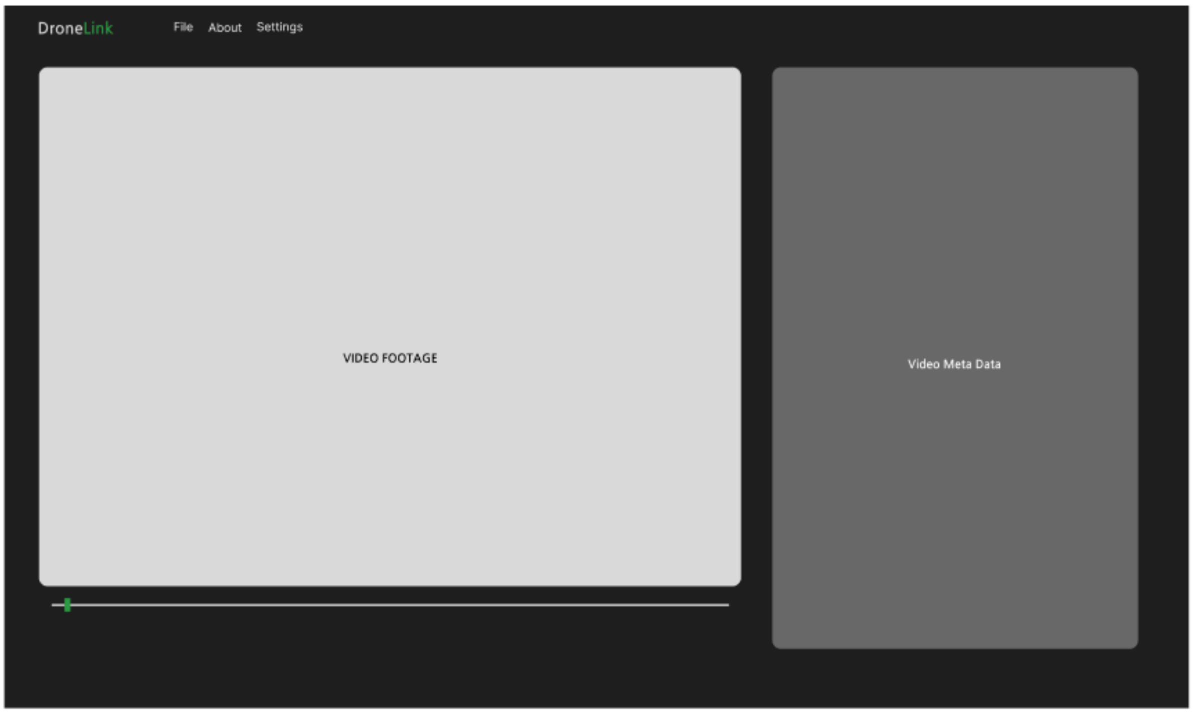


Figure 14: Main Page Wireframe

8.1.5 ***Conclusion***

Although there were certain constraints related to the visual design, the utilisation of PySide 6 facilitates the development of a modern and easy to use GUI. Additionally, the modular approach to the code base means that it is easily maintainable and extendable. Having these design principles eases the development process by removing the need for ad hoc modifications to core design elements or principles.

9 *Implementation*

9.1 *Introduction*

This section provides a comprehensive overview of the development process employed in the creation of the program, tentatively titled ‘DroneLink’. It outlines the sequential steps taken during the development and expands on the rationale behind the chosen methodologies. The implementation was guided by a modular design architecture outlined in the design phase.

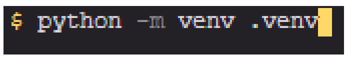
9.2 *Predevelopment*

Prior to starting any functional code or training models, it was important to set up a functional development environment, pipeline, and repository. This ensures that any packages installed were local to the project, any code written was tested on the main branch and finally that work in progress code wasn’t merged into the main branch.

9.2.1 *Virtual Environment*

Before starting development in earnest, a virtual environment had to be created to create an isolated dependency environment as well prevent packages from being installed globally on the machine.

This was performed by running this command:



```
$ python -m venv .venv
```

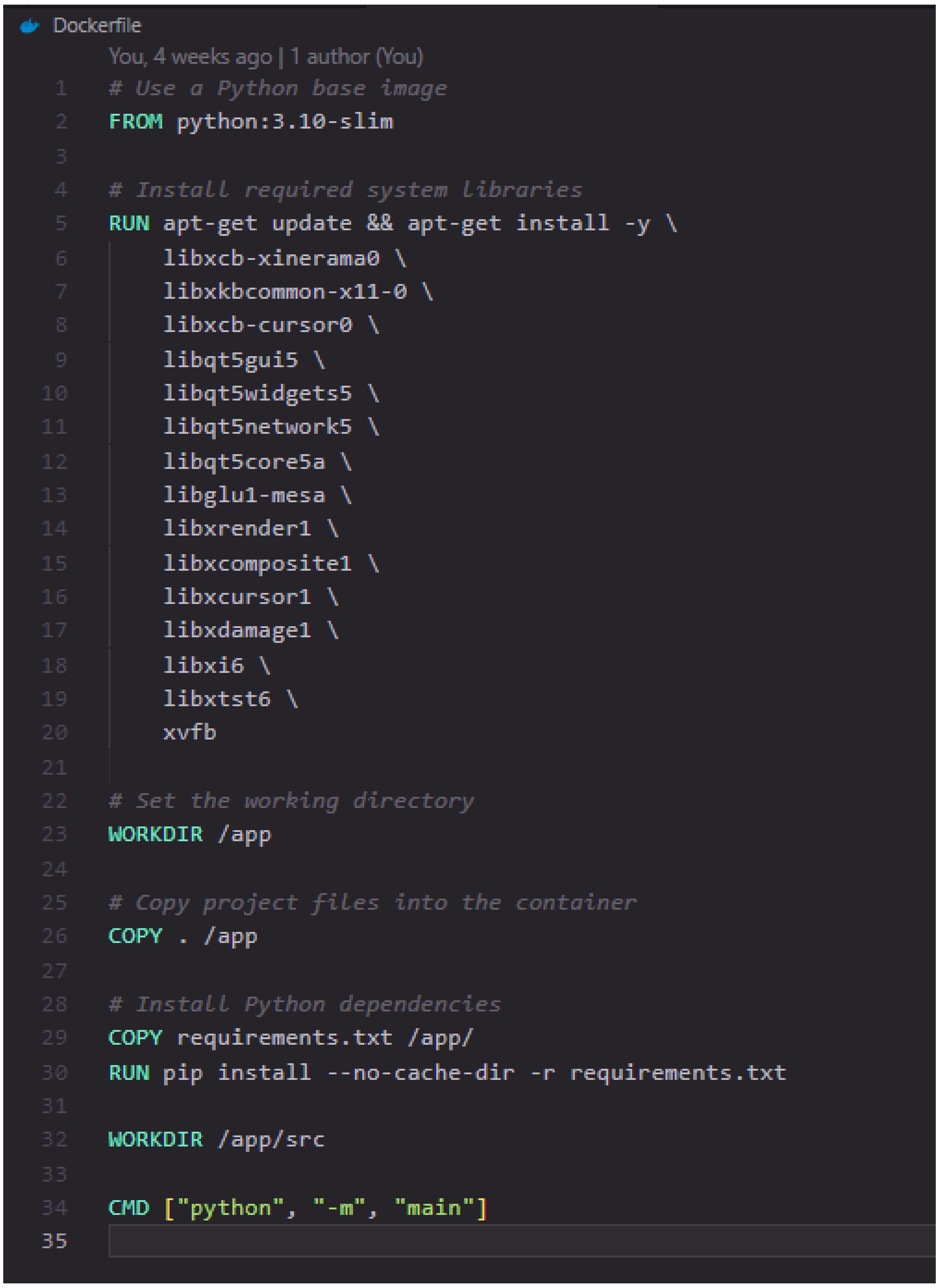
Figure 15: V Env Command

From this point the virtual environment could be activated and PIP packages installed.

9.2.2 *Continuous Integration Pipeline*

In order to run automated unit and integration tests on my code a CI pipeline that runs a Docker environment was needed. This ensures that the code is being tested in the same environment every single time the tests are run.

The Docker image was created by writing a Dockerfile. This file defines *layers* for the docker image to run when building the image.



Dockerfile

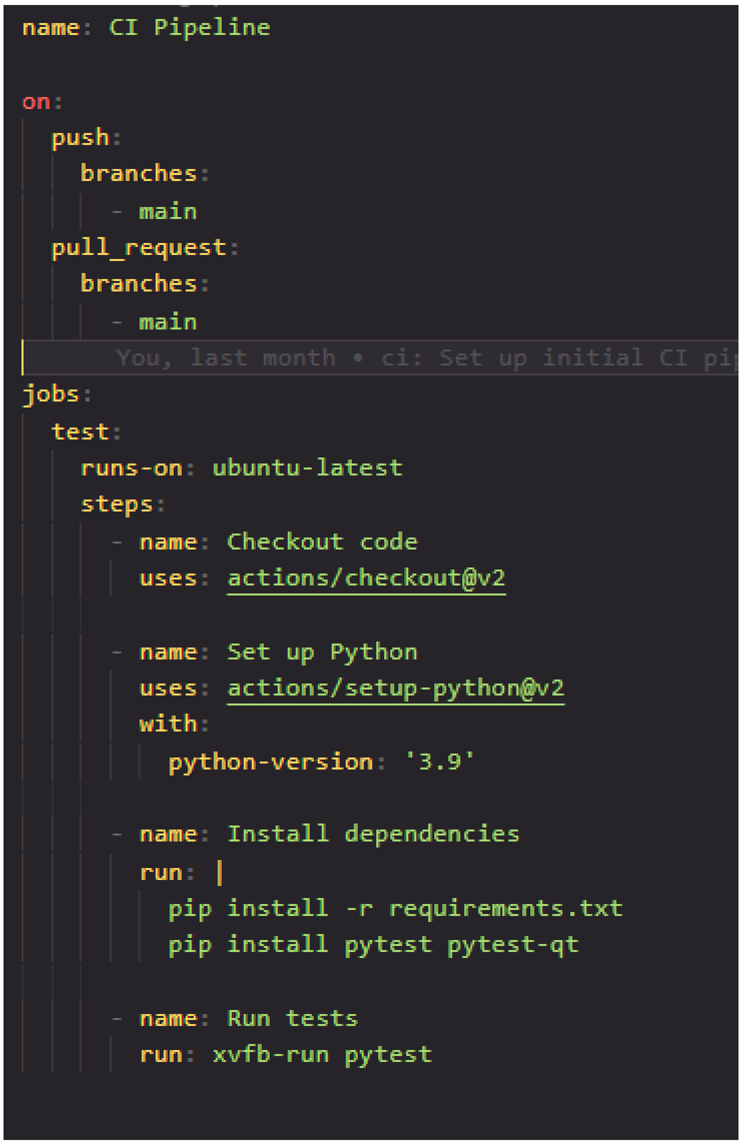
You, 4 weeks ago | 1 author (You)

```dockerfile
# Use a Python base image
FROM python:3.10-slim

# Install required system libraries
RUN apt-get update && apt-get install -y \
    libxcb-xinerama0 \
    libxkbcommon-x11-0 \
    libxcb-cursor0 \
    libqt5gui5 \
    libqt5widgets5 \
    libqt5network5 \
    libqt5core5a \
    libglu1-mesa \
    libxrender1 \
    libxcomposite1 \
    libxcursor1 \
    libxdamage1 \
    libxi6 \
    libxtst6 \
    xvfb

# Set the working directory
WORKDIR /app

# Copy project files into the container
COPY . /app

# Install Python dependencies
COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt

WORKDIR /app/src

CMD ["python", "-m", "main"]
```

Figure 16: Dockerfile

Unfortunately, it was during the implementation of this Dockerfile that I found out that PySide 6 couldn't be run in a dockerised environment as PySide 6 requires some sort of display in order to function. This led to much of the development not being ran through the pipeline correctly whilst I researched ways to have a containerised testing environment.

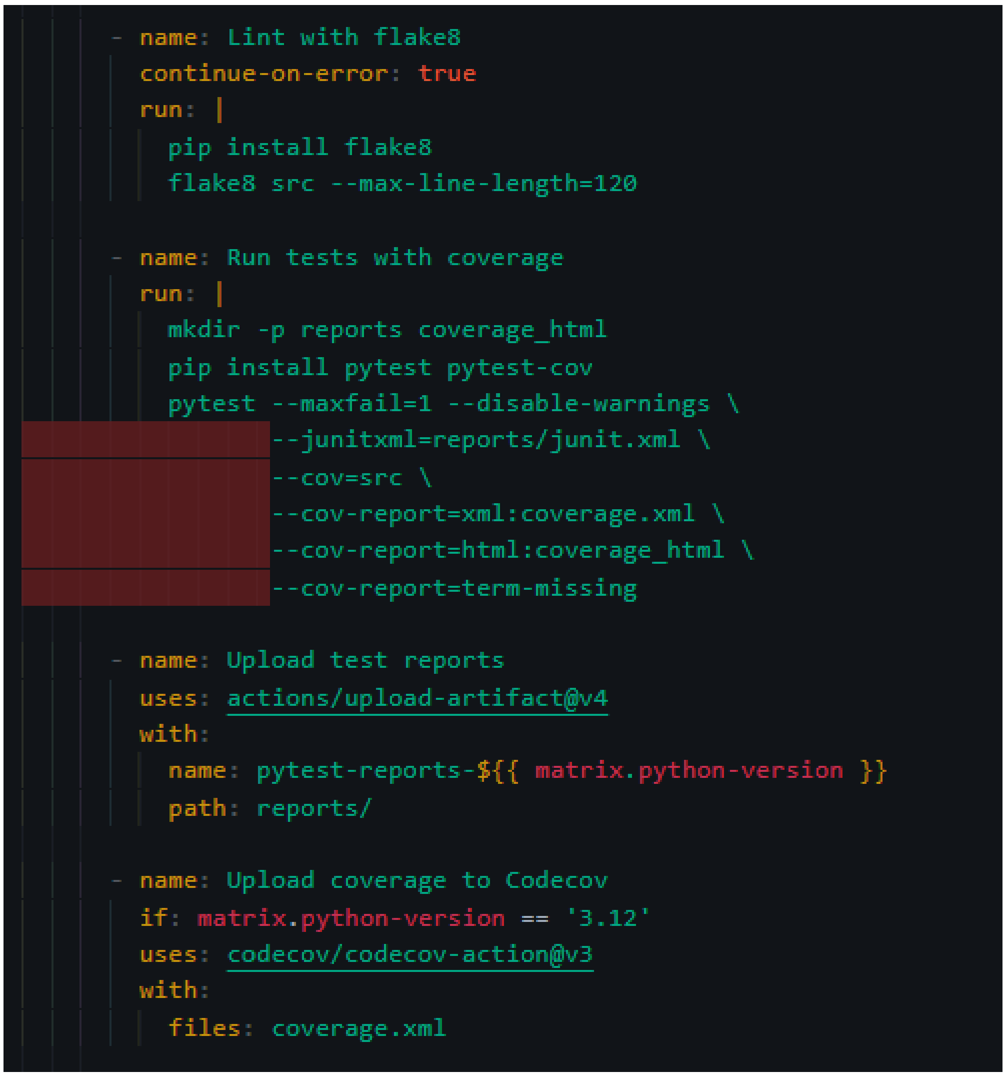
Ultimately, I decided to run the tests outside of an isolated environment in my pipeline for simplicity due to time constraints. I did, however, find later that I could have used a utility called ‘FlatPak’ which can run PySide 6 in partial isolation.



```yaml
name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
        You, last month • ci: Set up initial CI pi
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-qt

      - name: Run tests
        run: xvfb-run pytest
```

Figure 17: GitHub CI Yaml

I after the issues I encountered, I decided to implement linting, package caching and expand the testing to multiple Python versions in the pipeline. This was done by adding new line items in the pipeline YAML file.



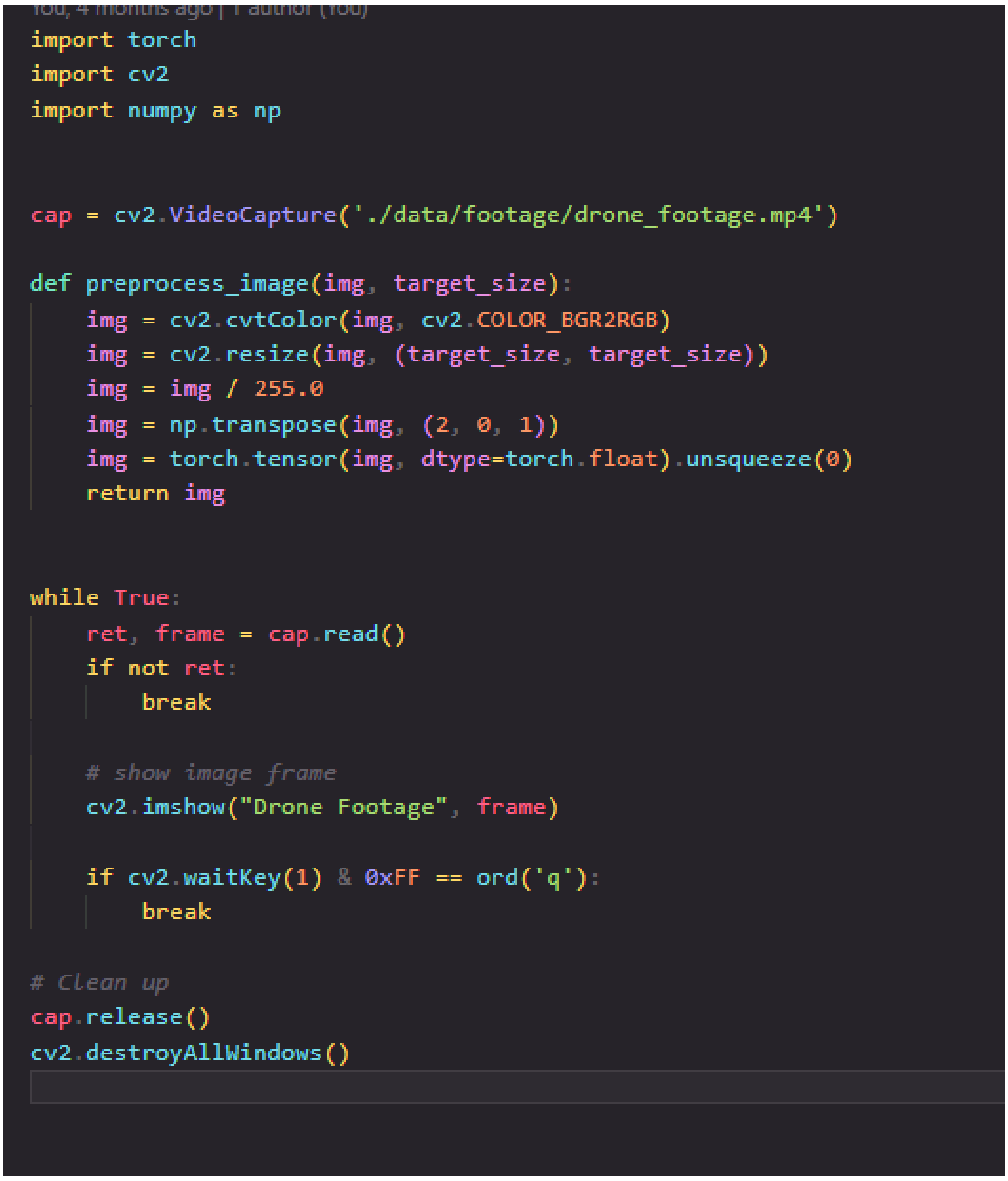
```yaml
      - name: Lint with flake8
        continue-on-error: true
        run: |
          pip install flake8
          flake8 src --max-line-length=120

      - name: Run tests with coverage
        run: |
          mkdir -p reports coverage_html
          pip install pytest pytest-cov
          pytest --maxfail=1 --disable-warnings \
                 --junitxml=reports/junit.xml \
                 --cov=src \
                 --cov-report=xml:coverage.xml \
                 --cov-report=html:coverage_html \
                 --cov-report=term-missing

      - name: Upload test reports
        uses: actions/upload-artifact@v4
        with:
          name: pytest-reports-${{ matrix.python-version }}
          path: reports/

      - name: Upload coverage to Codecov
        if: matrix.python-version == '3.12'
        uses: codecov/codecov-action@v3
        with:
          files: coverage.xml
```

Figure 18: Expanded Pipeline

This will be covered further in the Testing section.

9.3 ***Pre-recorded Video & OpenCV2***

The first task was to get a pre-recorded video to play in Python. To do this a package called OpenCV2 was utilized to parse the video. “Open CV is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products”. (OpenCV, n.d.)



```python
import torch
import cv2
import numpy as np


cap = cv2.VideoCapture('./data/footage/drone_footage.mp4')

def preprocess_image(img, target_size):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (target_size, target_size))
    img = img / 255.0
    img = np.transpose(img, (2, 0, 1))
    img = torch.tensor(img, dtype=torch.float).unsqueeze(0)
    return img


while True:
    ret, frame = cap.read()
    if not ret:
        break

    # show image frame
    cv2.imshow("Drone Footage", frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Clean up
cap.release()
cv2.destroyAllWindows()
```

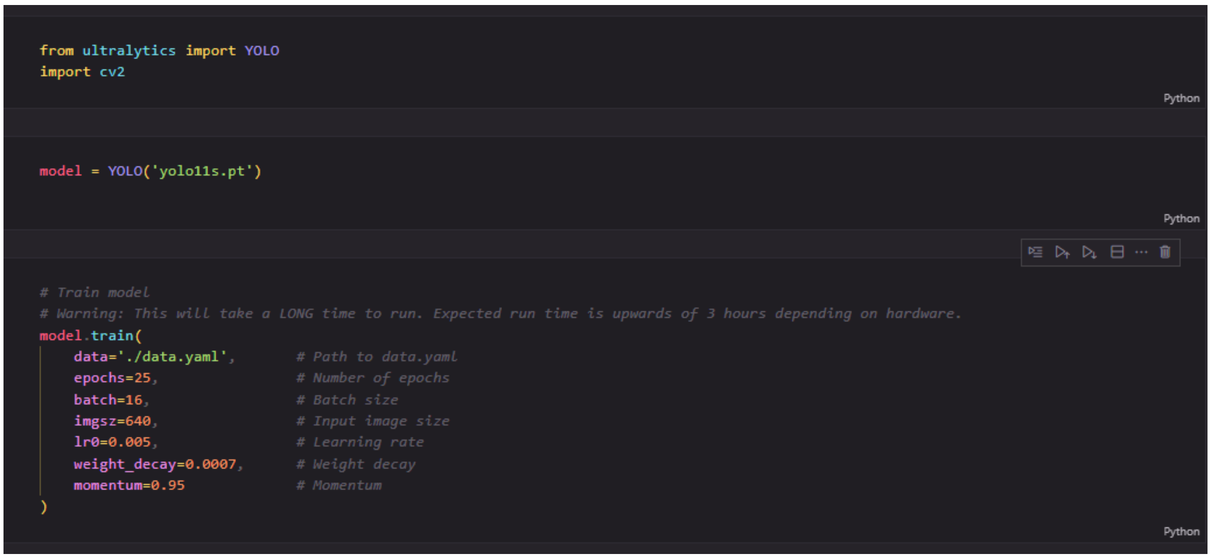
Figure 19: Showing a video in Python

This implementation imported the video using OpenCV’s VideoCapture method. This parses the video as a list of images that are displayed sequentially on the machine.

In the code, the video frames are pre-processed to be ready for processing by a ML model. This code snippet would become the core for the whole project.

9.4 *Initial Model Implementation*

Initially the models for this project were trained locally ona PC. This was done in a Jupyter notebook for rapid development and repeatability.



```python
from ultralytics import YOLO
import cv2
```

```python
model = YOLO('yolo11s.pt')
```

```python
# Train model
# Warning: This will take a LONG time to run. Expected run time is upwards of 3 hours depending on hardware.
model.train(
    data='./data.yaml',          # Path to data.yaml
    epochs=25,                   # Number of epochs
    batch=16,                    # Batch size
    imgsz=640,                   # Input image size
    lr0=0.005,                   # Learning rate
    weight_decay=0.0007,         # Weight decay
    momentum=0.95                # Momentum
)
```

Figure 20: Notebook Model Training

Initially, Ultralytics' YOLO11s model was used as a starting point for training with a generic dataset of labelled drone footage of humans. This dataset had a variety of classes relating to the activity of the person detected, such as 'running', 'walking, and 'laying_down' amongst others. The main goal of using this dataset wasn't to be accurate or useful for the end product. It just needed to function to a demonstrable degree so that a minimum viable product could be presented.

As seen in figure 19, it was chosen to train the model with 25 epochs in batches of 16 images. This resulted in a training time upwards of 3 hours on a Nvidia RTX 4080 graphics card. This lengthy training time was surprising as the dataset wasn't especially large, and the hardware was extremely capable.

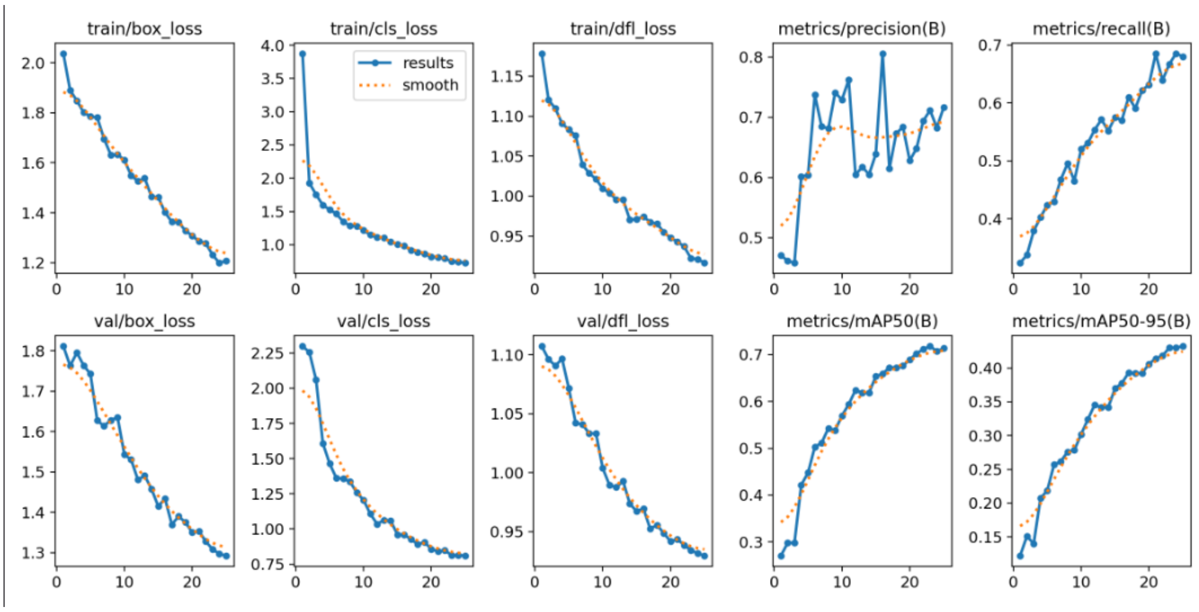


Figure 21: Model Training Report

After training the output validation curves reported that this model had an accuracy of approximately 65%, which is only slightly higher than chance at detecting human objects, this translated to poor performance in the benchmark video being used to test the model on real-world drone footage. This poor performance made it clear that a much improved and varied dataset and model would be required and the difference in accuracy indicates that the model was over-trained.

9.5 ***Improved Model***

Following the poor performance of the locally trained model a solution to the hardware limitations was found in with Ultralytics Hub and Google Colab Pro.

Initially a comparatively very large model was trained on the VisDrone dataset, this model had a reported accuracy of 60-70% in its validation tests but in actual use it appeared much better at detecting humans due to the larger and more varied training data. This dataset and the resulting model also had a slightly different classification set, meaning that vehicles and humans were now tracked, but poses would not be classified. It was felt that this was a good trade-off for the gain in detection capabilities

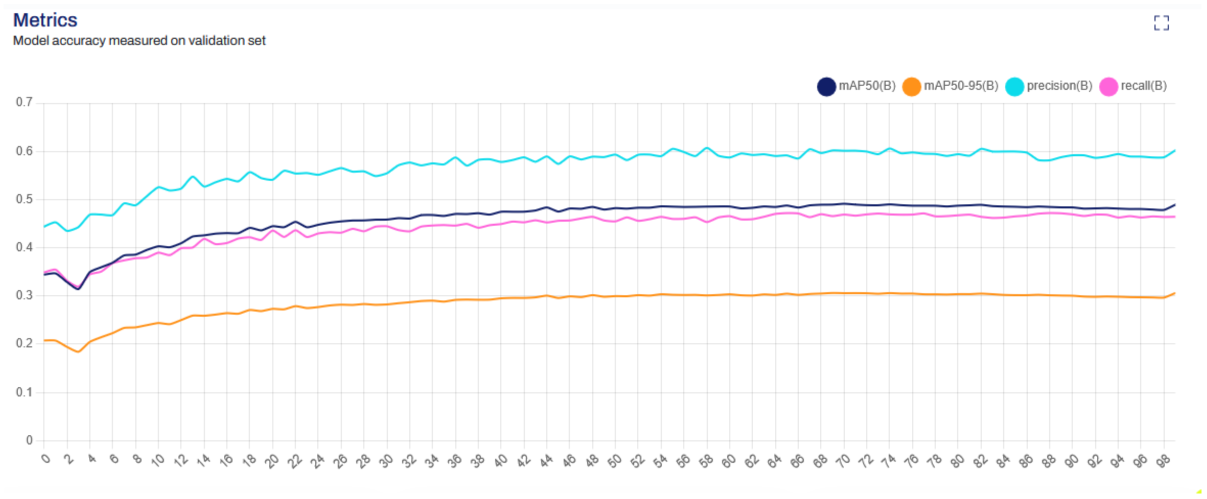


Figure 22: Large Model mAP

It is also important to distinguish between the accuracy metrics derived from training and validation versus real-world performance. While the mAP (Mean Average Precision) scores reflect the model's ability to detect objects in static images, actual usage often involves processing 30 frames per second of temporal data. If the primary objective is simply to detect a human at any point in a video, then the evaluation criteria differ from the standard per-image assessment. objective of the model is to detect a human at all then true accuracy of the model can be significantly lower and still yield useful results.

Despite the better performance of the new model on the benchmark video, the program experienced severe performance issues due to the size of the model and the processing it required on each frame. It was decided that a 'lite' model would be trained for usage on machines with limited performance. This 'lite' model had similar accuracy single image accuracy to the first model but with better edge case detection whilst being significantly more performant than the large model.

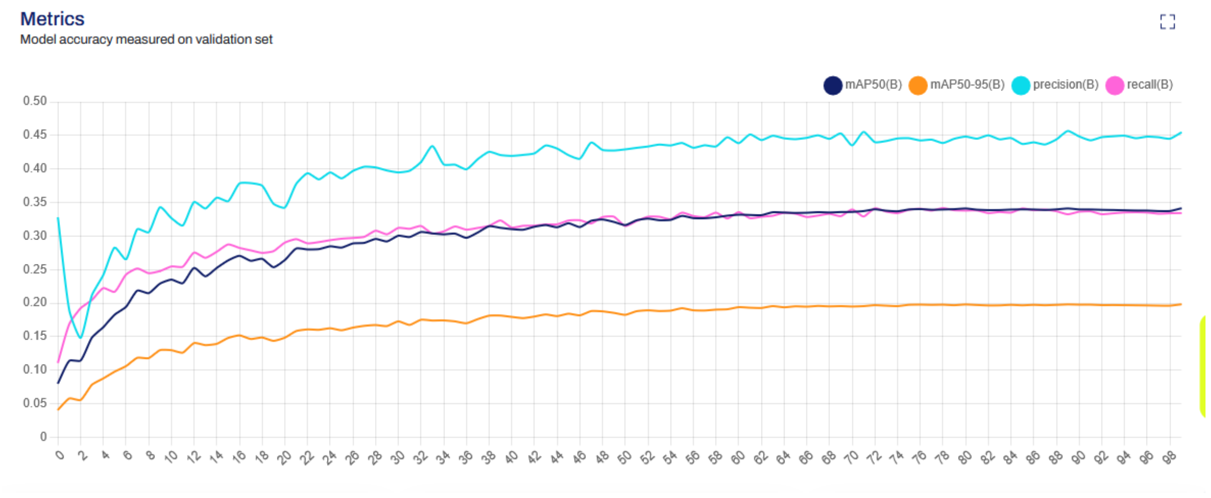
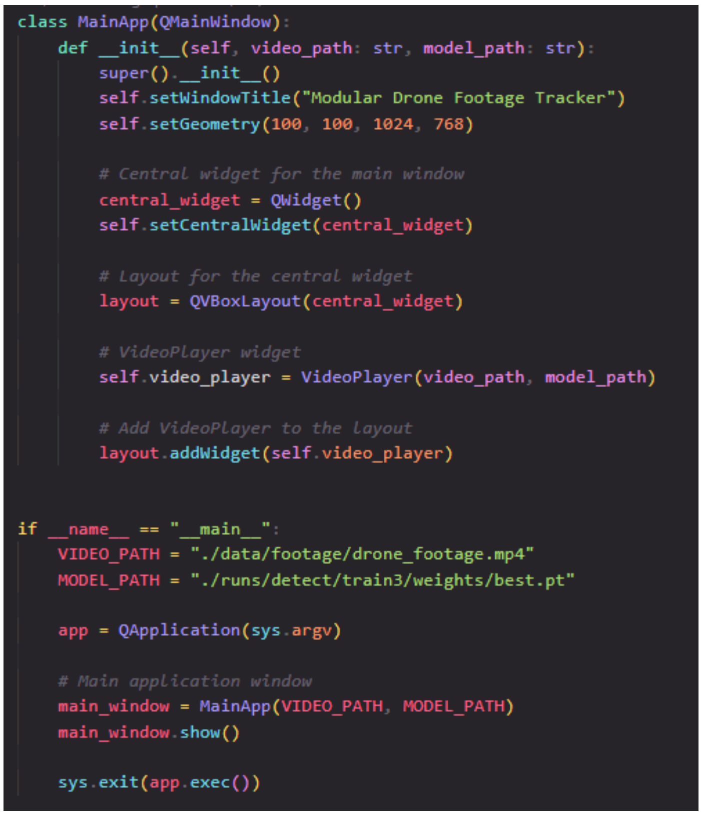


Figure 23: 'Lite' Model mAP

9.6 ***PySide 6 Video Player***

The user interface for the program was created using the Python package PySide 6 which is a python wrapper for the Qt C++ library. The decision was made to go with PySide 6 because it has a much more modern appearance over other libraries such as Tkinter and is very object oriented due to it being a C++ package at its core. It is also very well documented compared to more obscure or newer packages like DearPyGui. This made it the obvious choice for development despite the steep learning curve to the package.

The first goal in implementing the GUI was just to be able to display the video in a GUI container, with no interactivity. This was achieved simply by creating a QApplication in the main file and creating a video processor class module that served the frames to another class module called video player. This modular approach would be needed later in the project to ensure that I could process both live and pre-recorded frames.

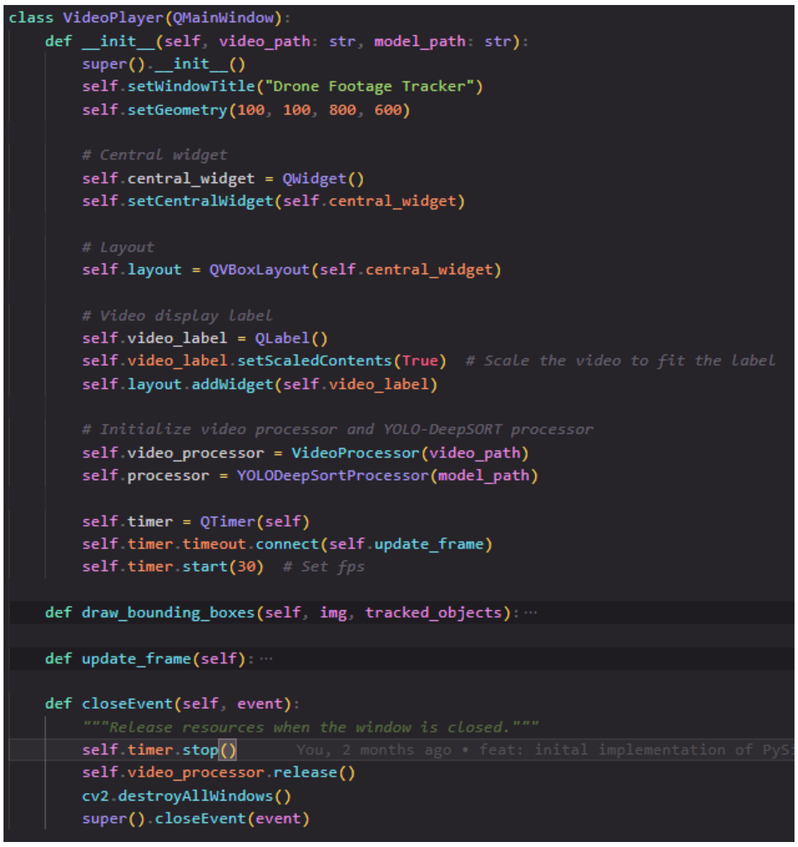


```python
class MainApp(QMainWindow):
    def __init__(self, video_path: str, model_path: str):
        super().__init__()
        self.setWindowTitle("Modular Drone Footage Tracker")
        self.setGeometry(100, 100, 1024, 768)

        # Central widget for the main window
        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        # Layout for the central widget
        layout = QVBoxLayout(central_widget)

        # VideoPlayer widget
        self.video_player = VideoPlayer(video_path, model_path)

        # Add VideoPlayer to the layout
        layout.addWidget(self.video_player)


if __name__ == "__main__":
    VIDEO_PATH = "./data/footage/drone_footage.mp4"
    MODEL_PATH = "./runs/detect/train3/weights/best.pt"

    app = QApplication(sys.argv)

    # Main application window
    main_window = MainApp(VIDEO_PATH, MODEL_PATH)
    main_window.show()

    sys.exit(app.exec())
```

Figure 24: Initial Video Player

Figure 23 illustrates the initial implementation of the video player module, which accepts variables for the video and model paths. Initially, these paths were hard coded for testing purposes.

The GUI is constructed by first initializing a PySide QApplication instance and then instantiating QWidgets within a MainWindow subclass; each discrete element of the interface is represented by a QWidget. The VideoPlayer class is subsequently attached to its corresponding widget, reinforcing a modular design. This approach allows future enhancements with ease, as adding new features simply involves integrating additional widgets.



```python
class VideoPlayer(QMainWindow):
    def __init__(self, video_path: str, model_path: str):
        super().__init__()
        self.setWindowTitle("Drone Footage Tracker")
        self.setGeometry(100, 100, 800, 600)

        # Central widget
        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        # Layout
        self.layout = QVBoxLayout(self.central_widget)

        # Video display label
        self.video_label = QLabel()
        self.video_label.setScaledContents(True)  # Scale the video to fit the label
        self.layout.addWidget(self.video_label)

        # Initialize video processor and YOLO-DeepSORT processor
        self.video_processor = VideoProcessor(video_path)
        self.processor = YOLODeepSortProcessor(model_path)

        self.timer = QTimer(self)
        self.timer.timeout.connect(self.update_frame)
        self.timer.start(30)  # Set fps

    def draw_bounding_boxes(self, img, tracked_objects):···

    def update_frame(self):···

    def closeEvent(self, event):
        """Release resources when the window is closed."""
        self.timer.stop()        You, 2 months ago • feat: inital implementation of PyS
        self.video_processor.release()
        cv2.destroyAllWindows()
        super().closeEvent(event)
```

Figure 25: VideoPlayer Class

When the VideoPlayer class is instantiated, it inherits the properties of the MainWindow. This allows it to set its own QWidget as the central widget and assign it to the MainWindow. VideoPlayer also instantiates a VideoProcessor instance which handles turning the video into frames using cv2's VideoCapture method and returning them to the video player.

9.7 *Dialogs and Signals*

In order for menus and actions to execute methods in the program in a non-blocking way, signals are utilised.

9.7.1 *Signals in PySide / Qt*

Signals in PySide 6 are a core aspect of Qt’s event driven architecture. A signal is essentially a notification that a specific event or action has occurred. Signals are emitted by objects when something notable happens, such as user input, data availability or a state change. They can be connected to special function / method decorators called ‘slots’, which handle these events.

Signals and slots provide a decoupled communication mechanism, meaning that objects don’t directly invoke methods on one another. Instead, they emit signals that interested parties connect to and react to the emissions. This design architecture encourages modularity, reusability and easier maintenance of the code.

9.7.2 *Dialog handling*

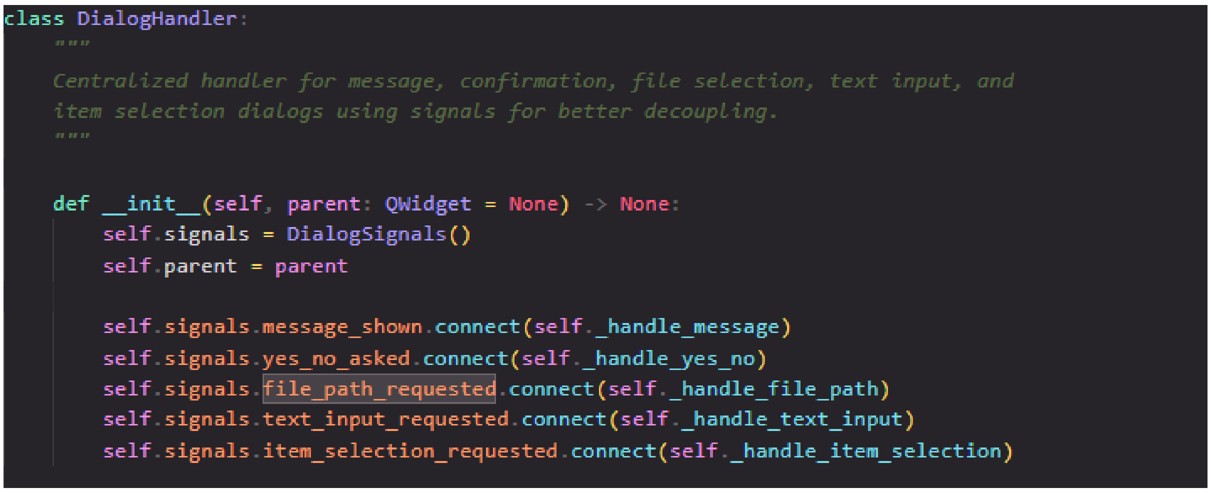
In the project, dialogs are managed though a dedicated class called ‘DialogHandler’. This class centralises GUI operations like file selection, confirmations and informational alerts, decoupling these user interactions form the main application logic. The signals serve as the primary means of communicating the results of the dialogs back to the applications logic.

Specifically, the DialogHandler class defines several signals:

- file_path_response: Emits when the user selects a file or saves one via a dialog. It provides information back to the main app, including the selected file path and whether the processing should be initiated.
- message_shown and yes_no_asked: Allow user feedback loops and confirmations in a structured and predictable manner.

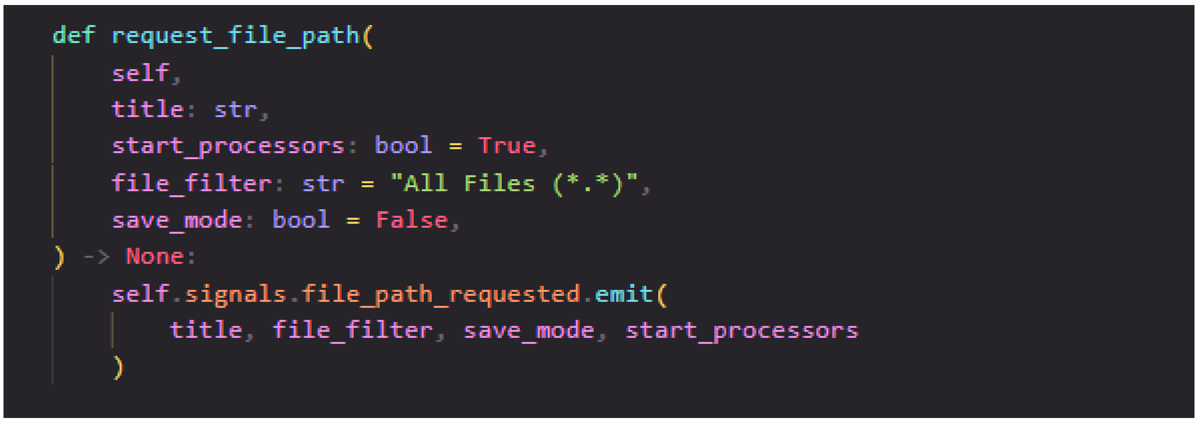
When a user selects a file from a dialog, the dialog emits a signal with the relevant details. The main application (MainApp) then connects to these signals, processing the returned information though defined slots.

9.7.3 ***Implementation***



```
class DialogHandler:
    """
    Centralized handler for message, confirmation, file selection, text input, and
    item selection dialogs using signals for better decoupling.
    """

    def __init__(self, parent: QWidget = None) -> None:
        self.signals = DialogSignals()
        self.parent = parent

        self.signals.message_shown.connect(self._handle_message)
        self.signals.yes_no_asked.connect(self._handle_yes_no)
        self.signals.file_path_requested.connect(self._handle_file_path)
        self.signals.text_input_requested.connect(self._handle_text_input)
        self.signals.item_selection_requested.connect(self._handle_item_selection)
```

Figure 26: Dialog Handler Class



```
    def request_file_path(
        self,
        title: str,
        start_processors: bool = True,
        file_filter: str = "All Files (*.*)",
        save_mode: bool = False,
    ) -> None:
        self.signals.file_path_requested.emit(
            title, file_filter, save_mode, start_processors
        )
```

Figure 27: Dialog Handler Class Method

As shown in Figures 25 and 26, signals are implemented as attributes of the DialogSignals class, instantiated within DialogHandler. Upon initialisation of DialogHandler, each signal defined within DialogSignals is immediately connected to a specific hander method. These handler methods encapsulate the logic for interacting with PySide’s GUI elements.

9.8 ***DeepSort***

9.8.1 ***What is DeepSort?***

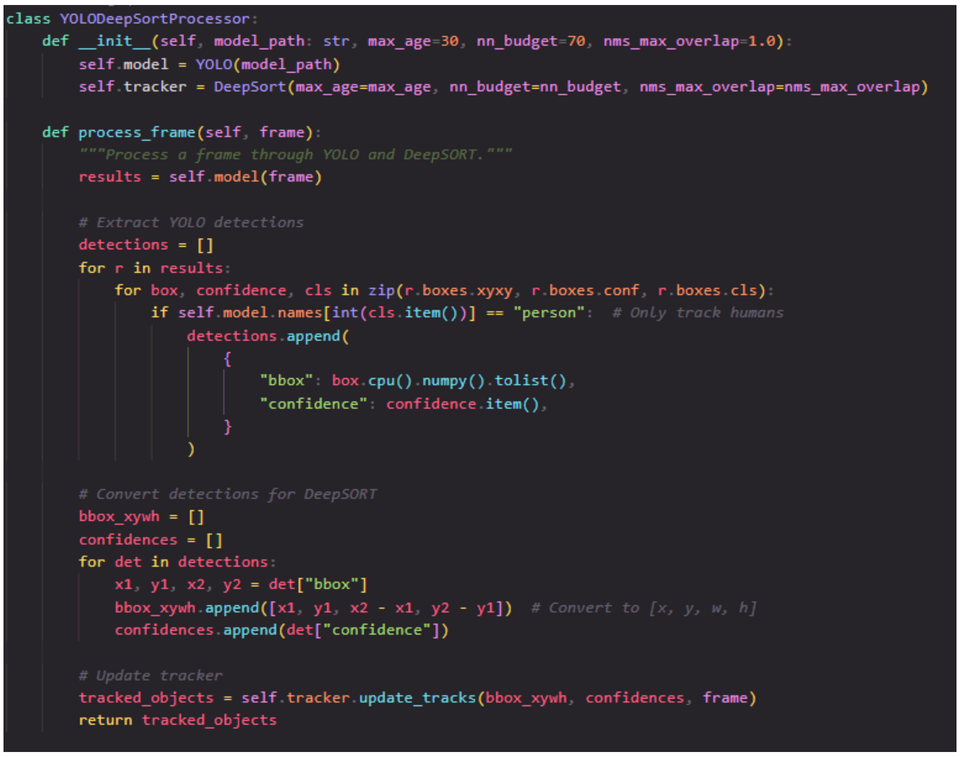
“Deep SORT (Simple Online and Realtime Tracking) is an algorithm used for multi-object tracking in video streams. It is an extension of the SORT (Simple Online and Realtime Tracking) algorithm, which uses the Kalman filter for object tracking. Deep

SORT incorporates a deep association metric based on appearance features learned by a deep convolutional neural network.” (Amjoud & Amrouch, 2023)

Simply put, DeepSort is an algorithm that detects multiple images through a series of frames, associates an ID number with each object and attempts to maintain the track whilst the object is present and detected in the frames.

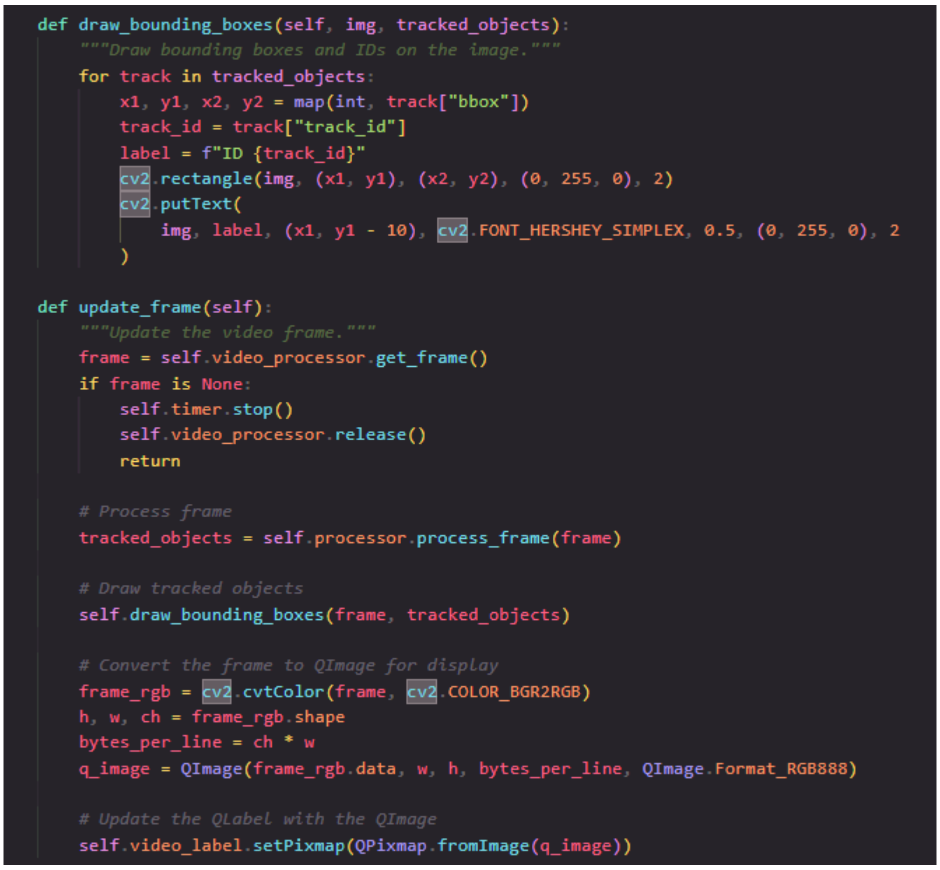
9.8.2 ***Why DeepSort?***

DeepSort was chosen for this project as it has great features that benefit the program, namely a feature called “Re-ID embeddings”. This is a feature, built into DeepSort, that keeps track of detected objects, even if they are temporarily obstructed from view. This is important in the context of this project because when an object is detected it is assigned an ID number. Keeping this ID number the same throughout the detection period can help the user from confusing detected objects with each other. Although, this does have the drawback of being computationally expensive.



```python
class YOLODeepSortProcessor:
    def __init__(self, model_path: str, max_age=30, nn_budget=70, nms_max_overlap=1.0):
        self.model = YOLO(model_path)
        self.tracker = DeepSort(max_age=max_age, nn_budget=nn_budget, nms_max_overlap=nms_max_overlap)

    def process_frame(self, frame):
        """Process a frame through YOLO and DeepSORT."""
        results = self.model(frame)

        # Extract YOLO detections
        detections = []
        for r in results:
            for box, confidence, cls in zip(r.boxes.xyxy, r.boxes.conf, r.boxes.cls):
                if self.model.names[int(cls.item())] == "person":  # Only track humans
                    detections.append(
                        {
                            "bbox": box.cpu().numpy().tolist(),
                            "confidence": confidence.item(),
                        }
                    )

        # Convert detections for DeepSORT
        bbox_xywh = []
        confidences = []
        for det in detections:
            x1, y1, x2, y2 = det["bbox"]
            bbox_xywh.append([x1, y1, x2 - x1, y2 - y1])  # Convert to [x, y, w, h]
            confidences.append(det["confidence"])

        # Update tracker
        tracked_objects = self.tracker.update_tracks(bbox_xywh, confidences, frame)
        return tracked_objects
```

Figure 28: YoloDeepSortProcessor



```python
def draw_bounding_boxes(self, img, tracked_objects):
    """Draw bounding boxes and IDs on the image."""
    for track in tracked_objects:
        x1, y1, x2, y2 = map(int, track["bbox"])
        track_id = track["track_id"]
        label = f"ID {track_id}"
        cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(
            img, label, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2
        )

def update_frame(self):
    """Update the video frame."""
    frame = self.video_processor.get_frame()
    if frame is None:
        self.timer.stop()
        self.video_processor.release()
        return

    # Process frame
    tracked_objects = self.processor.process_frame(frame)

    # Draw tracked objects
    self.draw_bounding_boxes(frame, tracked_objects)

    # Convert the frame to QImage for display
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    h, w, ch = frame_rgb.shape
    bytes_per_line = ch * w
    q_image = QImage(frame_rgb.data, w, h, bytes_per_line, QImage.Format_RGB888)

    # Update the QLabel with the QImage
    self.video_label.setPixmap(QPixmap.fromImage(q_image))
```

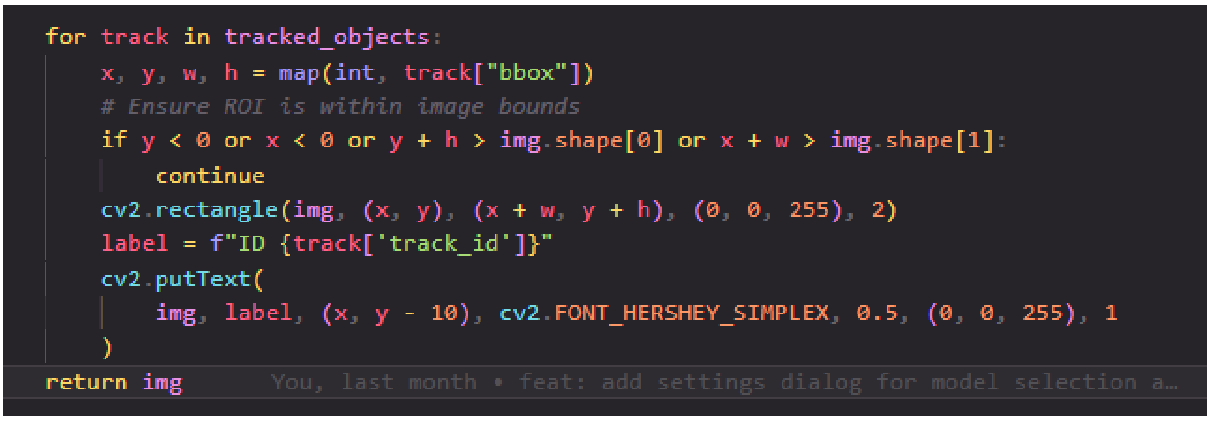
Figure 29: VideoPlayer Model Methods

9.8.3 *How does the detection work?*

After successfully getting the video running in the GUI the next task was to implement the model processing on the frames. This is done by the YOLODeepSortProcessor.

The update_frame method from the VideoPlayer calls the YOLODeepSortProcessor's process_frame method. This method runs the model on the frame and returns the bounding box coordinates of the detected objects, the confidence level of the detection and the classification. These variables are then passed into DeepSort to update the tracked objects

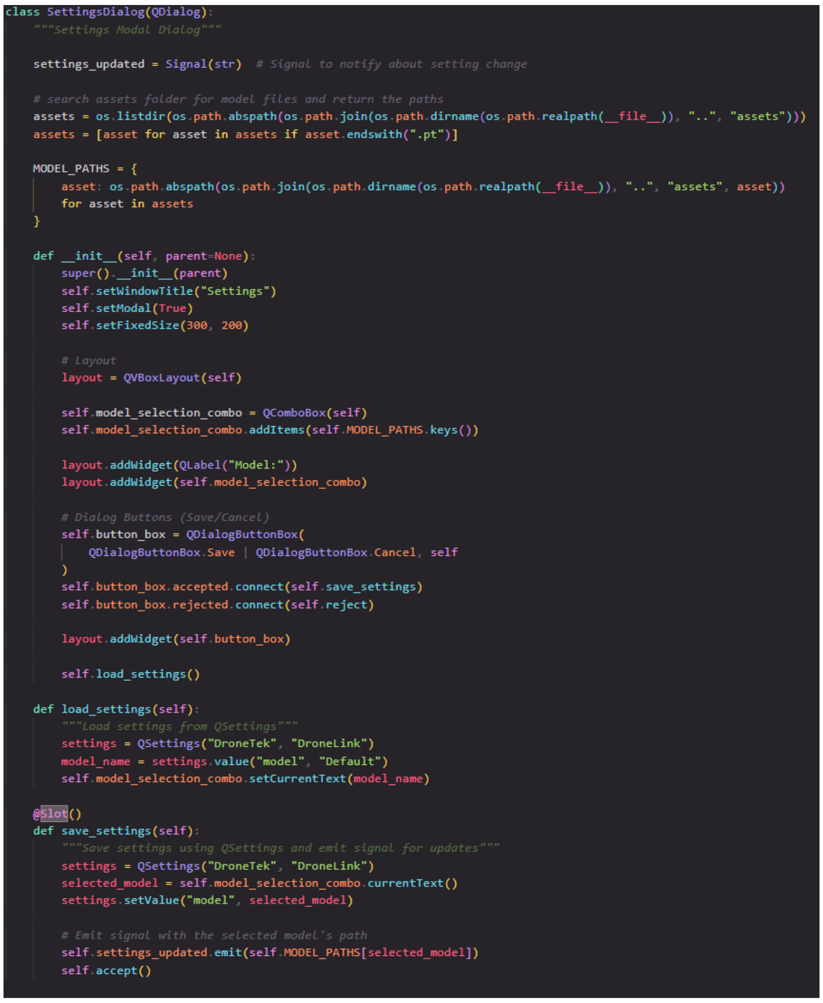
Once the frame has been processed, the bounding boxes are drawn using the coordinates from the tracked objects using CV2's rectangle method and the tracking ID is written above the box (Figure 27)



```python
for track in tracked_objects:
    x, y, w, h = map(int, track["bbox"])
    # Ensure ROI is within image bounds
    if y < 0 or x < 0 or y + h > img.shape[0] or x + w > img.shape[1]:
        continue
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)
    label = f"ID {track['track_id']}"
    cv2.putText(
        img, label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1
    )
return img        You, last month • feat: add settings dialog for model selection a…
```

Figure 30: Drawing a rectangle & ID

9.9 ***Settings***



```python
class SettingsDialog(QDialog):
    """Settings Modal Dialog"""

    settings_updated = Signal(str)  # Signal to notify about setting change

    # search assets folder for model files and return the paths
    assets = os.listdir(os.path.abspath(os.path.join(os.path.dirname(os.path.realpath(__file__)), "..", "assets")))
    assets = [asset for asset in assets if asset.endswith(".pt")]

    MODEL_PATHS = {
        asset: os.path.abspath(os.path.join(os.path.dirname(os.path.realpath(__file__)), "..", "assets", asset))
        for asset in assets
    }

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Settings")
        self.setModal(True)
        self.setFixedSize(300, 200)

        # Layout
        layout = QVBoxLayout(self)

        self.model_selection_combo = QComboBox(self)
        self.model_selection_combo.addItems(self.MODEL_PATHS.keys())

        layout.addWidget(QLabel("Model:"))
        layout.addWidget(self.model_selection_combo)

        # Dialog Buttons (Save/Cancel)
        self.button_box = QDialogButtonBox(
            QDialogButtonBox.Save | QDialogButtonBox.Cancel, self
        )
        self.button_box.accepted.connect(self.save_settings)
        self.button_box.rejected.connect(self.reject)

        layout.addWidget(self.button_box)

        self.load_settings()

    def load_settings(self):
        """Load settings from QSettings"""
        settings = QSettings("DroneTek", "DroneLink")
        model_name = settings.value("model", "Default")
        self.model_selection_combo.setCurrentText(model_name)

    @Slot()
    def save_settings(self):
        """Save settings using QSettings and emit signal for updates"""
        settings = QSettings("DroneTek", "DroneLink")
        selected_model = self.model_selection_combo.currentText()
        settings.setValue("model", selected_model)

        # Emit signal with the selected model's path
        self.settings_updated.emit(self.MODEL_PATHS[selected_model])
        self.accept()
```

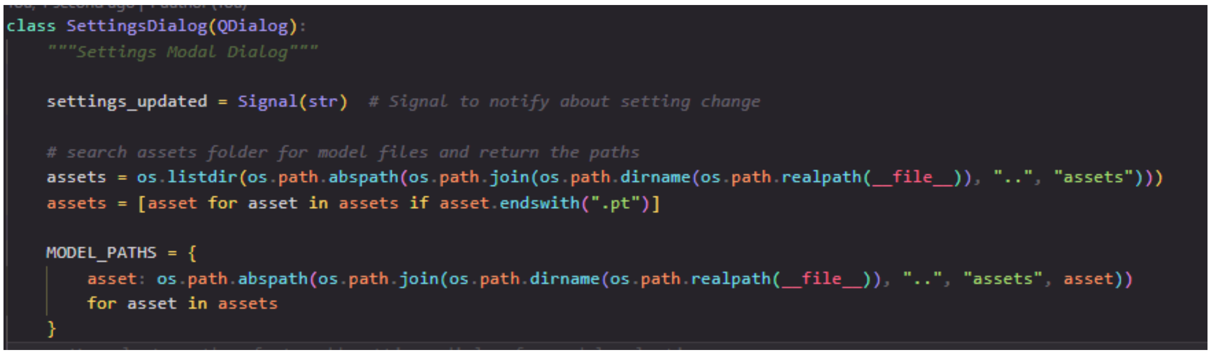
Figure 31: Settings Dialog Class

The next major feature integrated into the application was the settings menu, which introduces user-configurable control over core aspects of the system. This modal dialog provides an intuitive interface for adjusting runtime parameters, with the primary focus on allowing users to select among different object detection models included in the application.

The primary motivation for implementing this feature arose from the need to cater to user preferences and varying hardware capabilities. Different object detection models exhibit significant differences in computational requirements and performance characteristics. By allowing users to select a model that aligns with their specific needs and hardware constraints, the application ensures consistent performance across a wide range of hardware.

9.9.1 *Implementation*

The SettingsDialog class uses PySide6's widgets to provide user-friendly settings interface. Upon initialisation, the dialog dynamically identifies all available .pt (PyTorch) model files within the application's local assets directory. It then populates these model files into a QComboBox widget, facilitating straightforward model selection. This dynamic approach ensures scalability; new models can be effortlessly integrated into the application by simply placing the respective .pt files into the assets folder, without the need for additional code modifications.



```python
class SettingsDialog(QDialog):
    """Settings Modal Dialog"""

    settings_updated = Signal(str)  # Signal to notify about setting change

    # search assets folder for model files and return the paths
    assets = os.listdir(os.path.abspath(os.path.join(os.path.dirname(os.path.realpath(__file__)), "..", "assets")))
    assets = [asset for asset in assets if asset.endswith(".pt")]

    MODEL_PATHS = {
        asset: os.path.abspath(os.path.join(os.path.dirname(os.path.realpath(__file__)), "..", "assets", asset))
        for asset in assets
    }
```

Figure 32: SettingsDialog Model Loading

The settings dialog uses QSettings, Qt’s built-in persistent storage. Working similarly to web cookies, to manage application preferences and ensure consistency between sessions. When users open the settings menu, the dialog reads the previously selected model from QSettings and sets it as the default selection within the dropdown box. This provides continuity and improves user experience, minimizing the configuration effort across different usage sessions. If no previous model selection exists the user is prevented from running live or pre-recorded video analysis until they select a model from the settings menu. This is achieved by disabling the buttons in the file dropdown.

Once the Save button is pressed, the selected model preference is saved back to QSettings. The settings dialog then emits a custom-defined signal, settings_updated, containing the path to the newly selected model. This signal notifies all dependent components of the settings update. Connected slots within the main application then receive this signal and adjust the model loading to the updated path without requiring a restart.

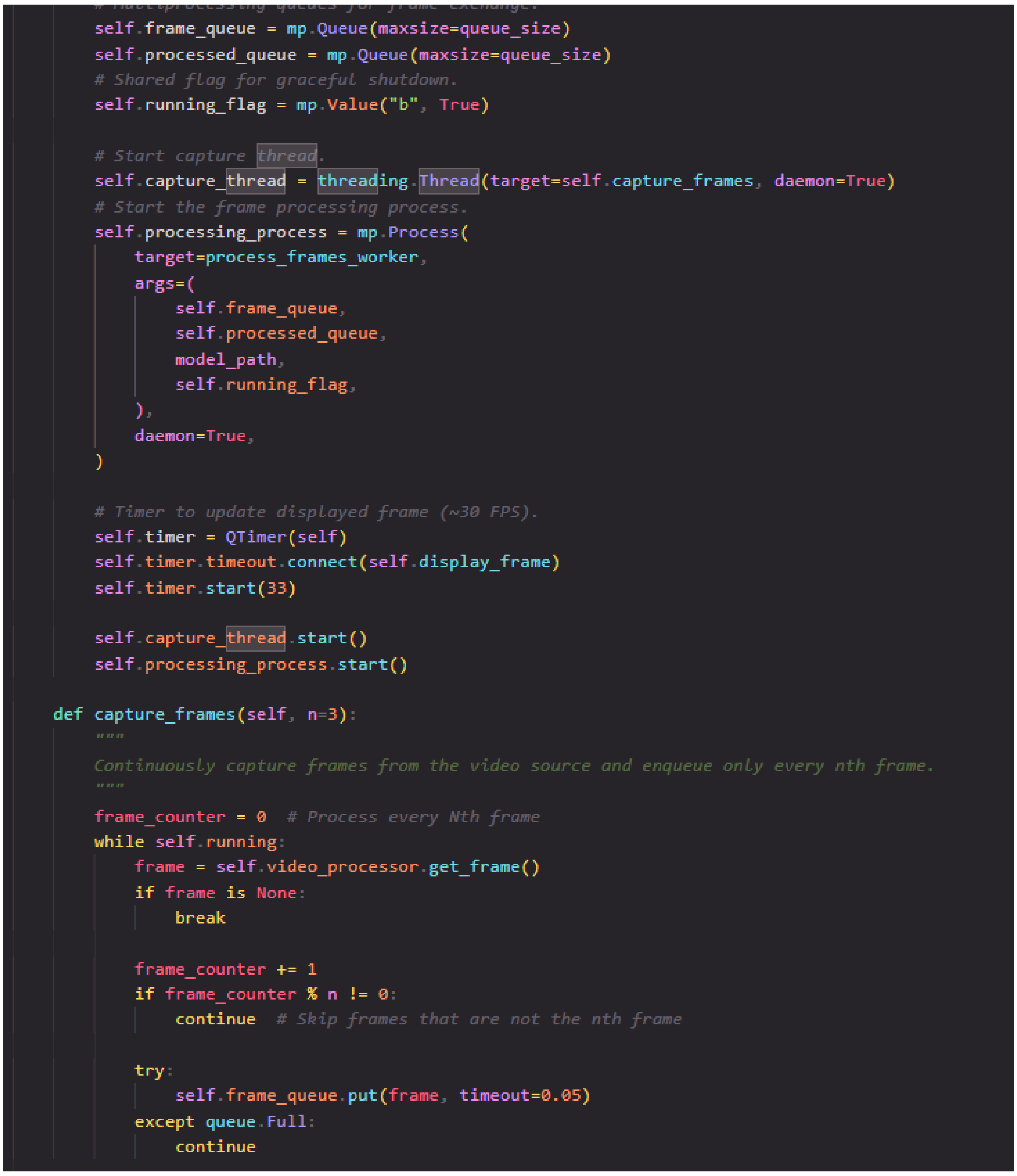
9.9.1.1 Refactoring

The introduction of the settings feature required significant refactoring of existing functions to accommodate dynamic user preferences. Initially, the object detection model path was hardcoded into the program, significantly restricting flexibility and scalability. Migrating to the new design required removing these hardcoded dependencies, instead using a configurable approach.

9.10 *Multi-Threading*

As the project progressed it became apparent that the current approach to the software design was becoming significantly more computationally intensive than initially forecasted. To address these performance concerns, multithreading and multiprocessing techniques were integrated into the software's design.

9.10.1 *Threading & Processing*



```
        # Multiprocessing queues for frame exchange.
        self.frame_queue = mp.Queue(maxsize=queue_size)
        self.processed_queue = mp.Queue(maxsize=queue_size)
        # Shared flag for graceful shutdown.
        self.running_flag = mp.Value("b", True)

        # Start capture thread.
        self.capture_thread = threading.Thread(target=self.capture_frames, daemon=True)
        # Start the frame processing process.
        self.processing_process = mp.Process(
            target=process_frames_worker,
            args=(
                self.frame_queue,
                self.processed_queue,
                model_path,
                self.running_flag,
            ),
            daemon=True,
        )

        # Timer to update displayed frame (~30 FPS).
        self.timer = QTimer(self)
        self.timer.timeout.connect(self.display_frame)
        self.timer.start(33)

        self.capture_thread.start()
        self.processing_process.start()

    def capture_frames(self, n=3):
        """
        Continuously capture frames from the video source and enqueue only every nth frame.
        """
        frame_counter = 0  # Process every Nth frame
        while self.running:
            frame = self.video_processor.get_frame()
            if frame is None:
                break

            frame_counter += 1
            if frame_counter % n != 0:
                continue  # Skip frames that are not the nth frame

            try:
                self.frame_queue.put(frame, timeout=0.05)
            except queue.Full:
                continue
```

Figure 33: Multi-Processing Implementation

Before implementing this, it was important to understand what threading and multi-processing are and how they can be useful in a project.

9.10.1.1 Threading

Threading involves executing multiple *threads* within the same process. These threads share memory and resources, this allows for fast communication and low overhead computing power.

9.10.1.2 Multi-Processing

Multi-processing is when multiple processes are executed simultaneously, each with their own separate memory space. This is well-suited to CPU bound or otherwise computationally intensive tasks. The processes run independently of each other, enabling parallel computation at the expense of higher resource overhead due to the additional management requirements of handling multiple processes.

9.10.2 *Multi-Threading Implementation*

Multi-threading was used in the project to continuously capture video frames from the video source. The thread manages the capture processes by retrieving every nth frame and enqueues the frame into a thread-safe frame queue. Using a separate thread for frame capture prevents blocking operations, which allows the GUI and other components to remain responsive and minimizes delays in frame acquisition.

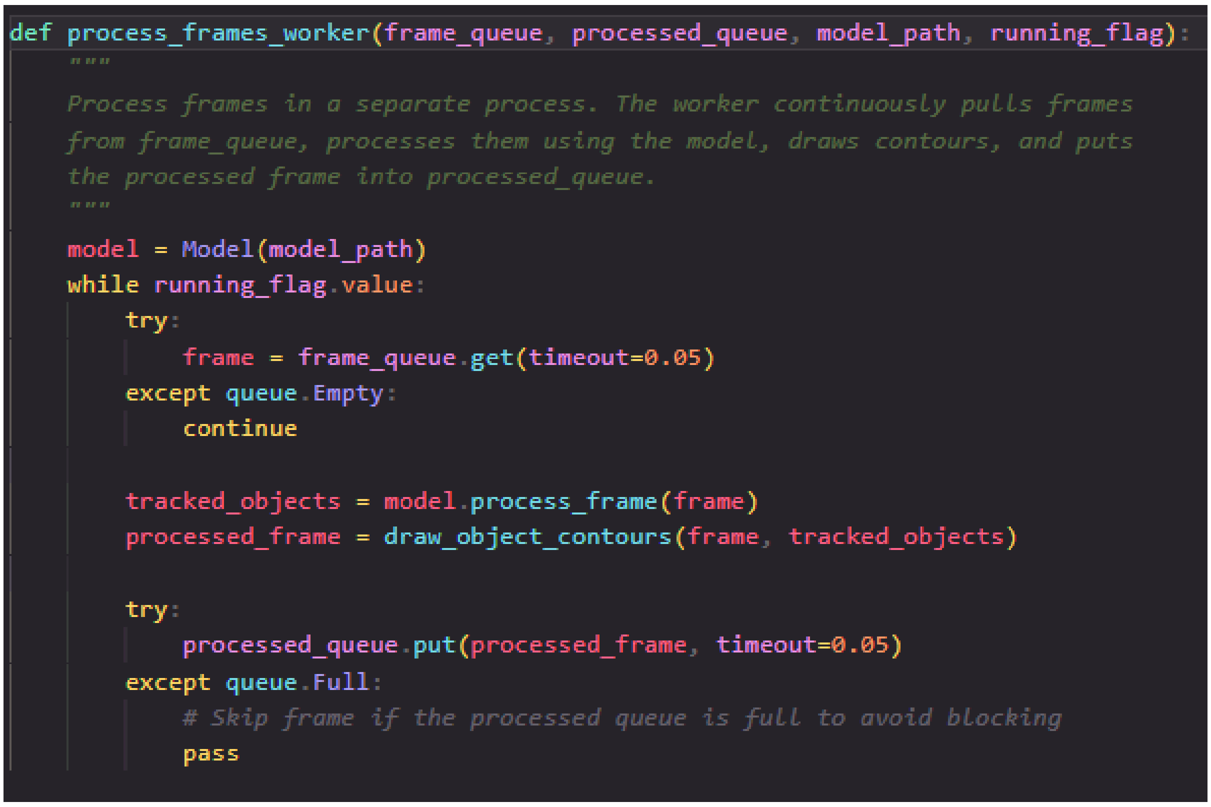
9.10.2.1 Thread Safety

In the context of this project and the frame queue, thread-safe refers to operations designed to function correctly when accessed simultaneously by multiple threads without causing data loss or race conditions. This means that, for example, the frame queue should have a locking mechanism and manage the consistency of shared data. In this project this is provided by default by the multiprocessing class’s Queue data structure.

9.10.3 *Multi-Processing Implementation*

A separate process (processing_process) handles intensive video processing tasks independently. This process continuously fetches frames from the shared queue, performs model inference, and places processed frames into another queue for display. Utilizing multiprocessing ensures that heavy computational workloads do not impede or degrade GUI responsiveness, thereby maintaining a consistent and smooth user experience.

Additionally, inter-thread and inter-process communication is managed through thread-safe and process-safe queues (mp.Queue). This strategy prevents the aforementioned race conditions and improves the responsiveness and stability of the GUI by ensuring that frame capture, processing, and display tasks occur concurrently yet independently. The combined implementation of threading and multiprocessing substantially improves the program's overall performance and responsiveness.



```python
def process_frames_worker(frame_queue, processed_queue, model_path, running_flag):
    """
    Process frames in a separate process. The worker continuously pulls frames
    from frame_queue, processes them using the model, draws contours, and puts
    the processed frame into processed_queue.
    """
    model = Model(model_path)
    while running_flag.value:
        try:
            frame = frame_queue.get(timeout=0.05)
        except queue.Empty:
            continue

        tracked_objects = model.process_frame(frame)
        processed_frame = draw_object_contours(frame, tracked_objects)

        try:
            processed_queue.put(processed_frame, timeout=0.05)
        except queue.Full:
            # Skip frame if the processed queue is full to avoid blocking
            pass
```

Figure 34: Multi-Processing Worker

The multi-processing worker function also utilises frame skipping if the frame queue exceeds its maximum size. This means that if the queue reaches its maximum size of 100 frames, then subsequent frames get skipped to prevent blocking behaviour from the program and allows the processing to catch up.

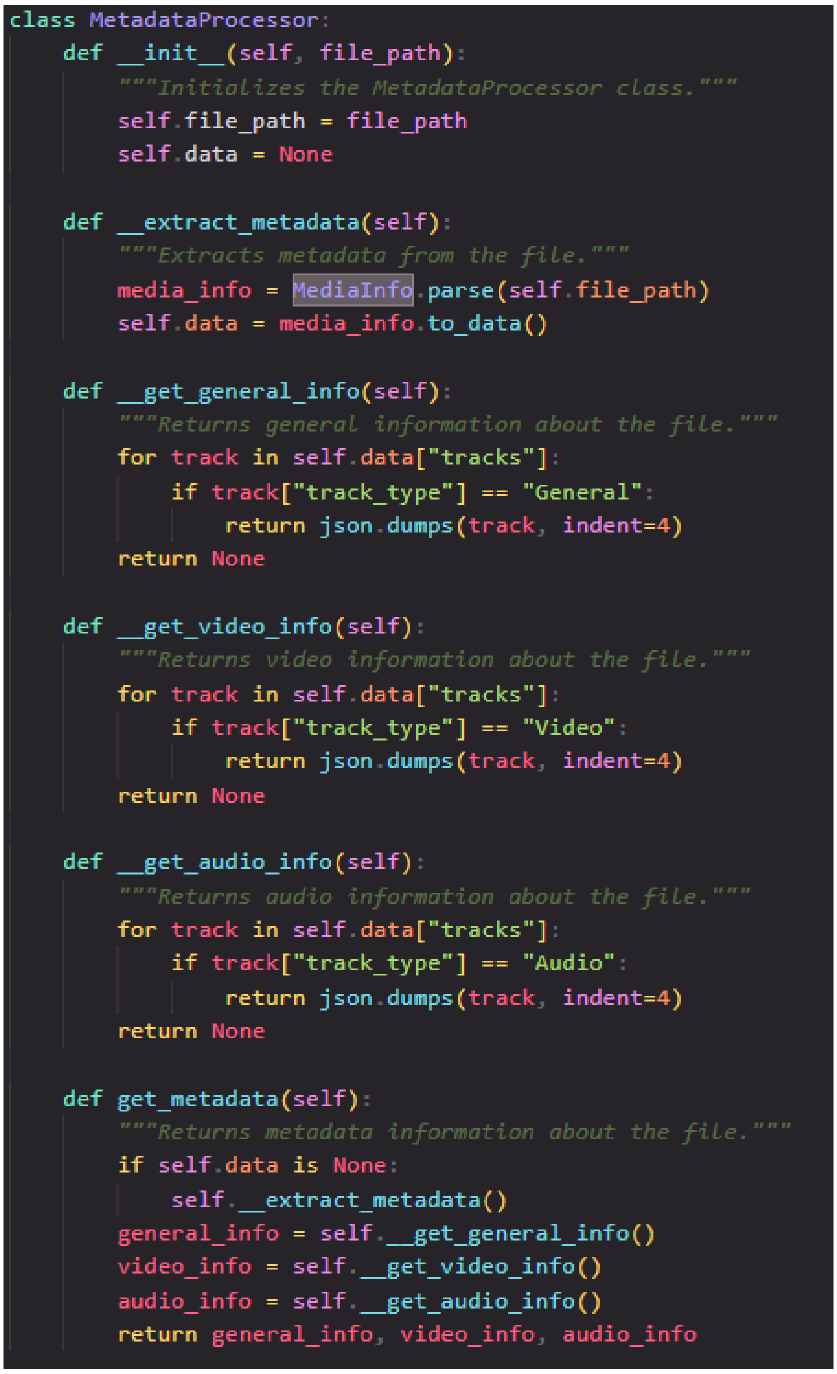
9.10.4 ***Threading Conclusion***

Learning how threading and multiprocessing functioned within the context of Python and how to integrate it into the program took a considerable

9.11 ***Metadata Extraction***

Metadata refers to the descriptive information embedded within digital files such as video or images. This data typically contains information about the date and time of the recording, camera settings, GPS data and device information. This can be useful to SAR teams in many situations, such as an image or video become relevant to a case after the video was recorded. Metadata can give detailed information about where and when the video was recorded.

The extraction of this data was done by using the pymediainfo package in the MetadataProcessor.



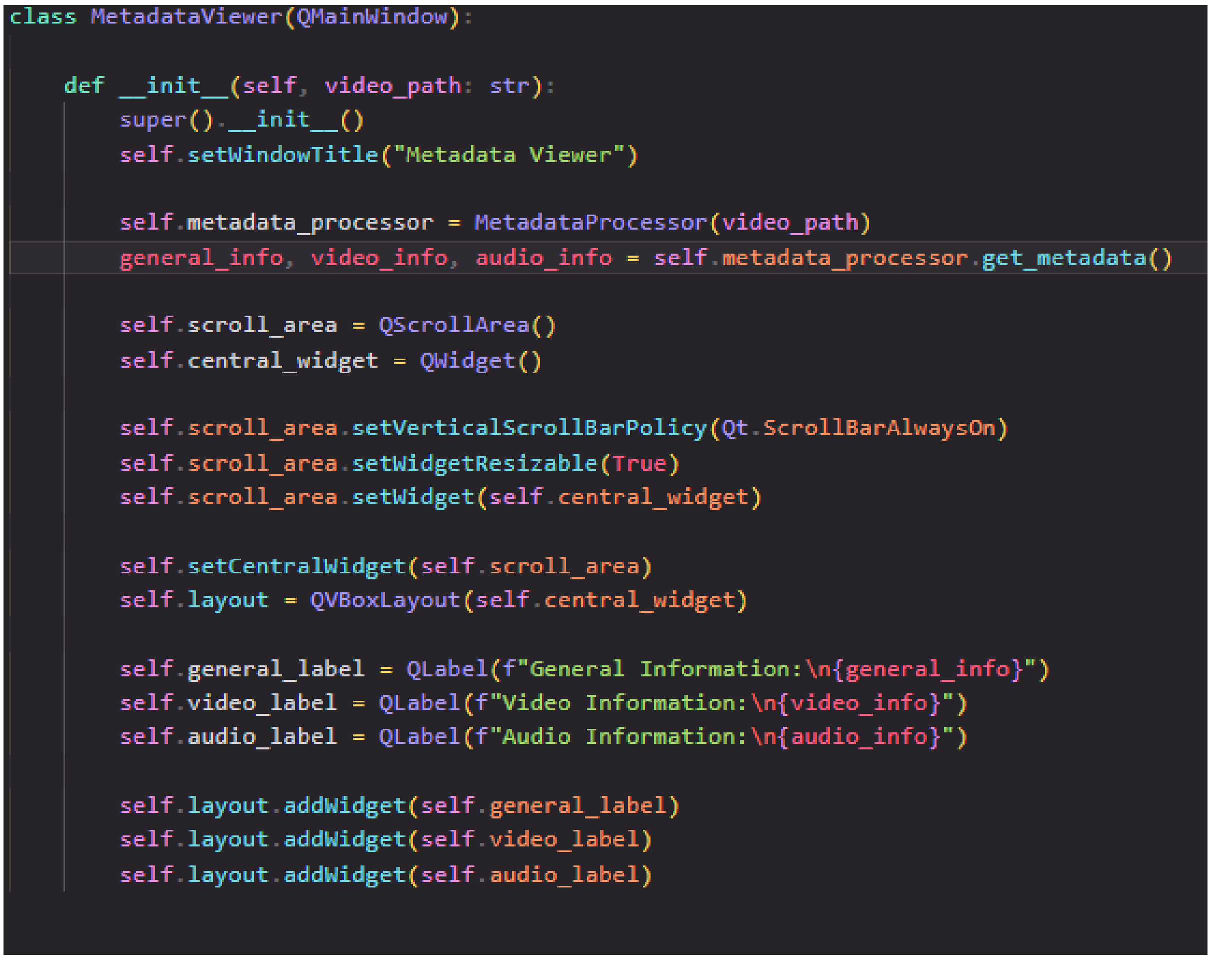
```python
class MetadataProcessor:
    def __init__(self, file_path):
        """Initializes the MetadataProcessor class."""
        self.file_path = file_path
        self.data = None

    def __extract_metadata(self):
        """Extracts metadata from the file."""
        media_info = MediaInfo.parse(self.file_path)
        self.data = media_info.to_data()

    def __get_general_info(self):
        """Returns general information about the file."""
        for track in self.data["tracks"]:
            if track["track_type"] == "General":
                return json.dumps(track, indent=4)
        return None

    def __get_video_info(self):
        """Returns video information about the file."""
        for track in self.data["tracks"]:
            if track["track_type"] == "Video":
                return json.dumps(track, indent=4)
        return None

    def __get_audio_info(self):
        """Returns audio information about the file."""
        for track in self.data["tracks"]:
            if track["track_type"] == "Audio":
                return json.dumps(track, indent=4)
        return None

    def get_metadata(self):
        """Returns metadata information about the file."""
        if self.data is None:
            self.__extract_metadata()
        general_info = self.__get_general_info()
        video_info = self.__get_video_info()
        audio_info = self.__get_audio_info()
        return general_info, video_info, audio_info
```

Figure 35: MetadataProcessor

The get_metadata method is the only publicly available method on the class, this method calls all the necessary private methods so that the data can be collected, merged and returned in a single method.

The approach was chosen for the class design to abstract some of the repeated actions away from the developer and to improve the code readability where the class methods are called.

Initially it was hoped that video metadata would contain geographic data that would allow the program to export or display the location of the video in some manner. Unfortunately, it was found that videos do not typically have geographic metadata. Despite this, the entirety of the video metadata is provided to the user in a Qwidget to the side of the video.



```python
class MetadataViewer(QMainWindow):

    def __init__(self, video_path: str):
        super().__init__()
        self.setWindowTitle("Metadata Viewer")

        self.metadata_processor = MetadataProcessor(video_path)
        general_info, video_info, audio_info = self.metadata_processor.get_metadata()

        self.scroll_area = QScrollArea()
        self.central_widget = QWidget()

        self.scroll_area.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOn)
        self.scroll_area.setWidgetResizable(True)
        self.scroll_area.setWidget(self.central_widget)

        self.setCentralWidget(self.scroll_area)
        self.layout = QVBoxLayout(self.central_widget)

        self.general_label = QLabel(f"General Information:\n{general_info}")
        self.video_label = QLabel(f"Video Information:\n{video_info}")
        self.audio_label = QLabel(f"Audio Information:\n{audio_info}")

        self.layout.addWidget(self.general_label)
        self.layout.addWidget(self.video_label)
        self.layout.addWidget(self.audio_label)
```

Figure 36: Metadata Viewer

When there is a video playing the main process calls the metadata viewer class which, in turn, uses the MetadataProcessor to take metadata from the same video source. The data is then presented to the user in a Qwidget that has been made scrollable due to the length of the data.

9.12 ***Archive Processor***

The next task in the development of the project was to implement a way for the user to export the processed footage in a way that both preserved the original footage

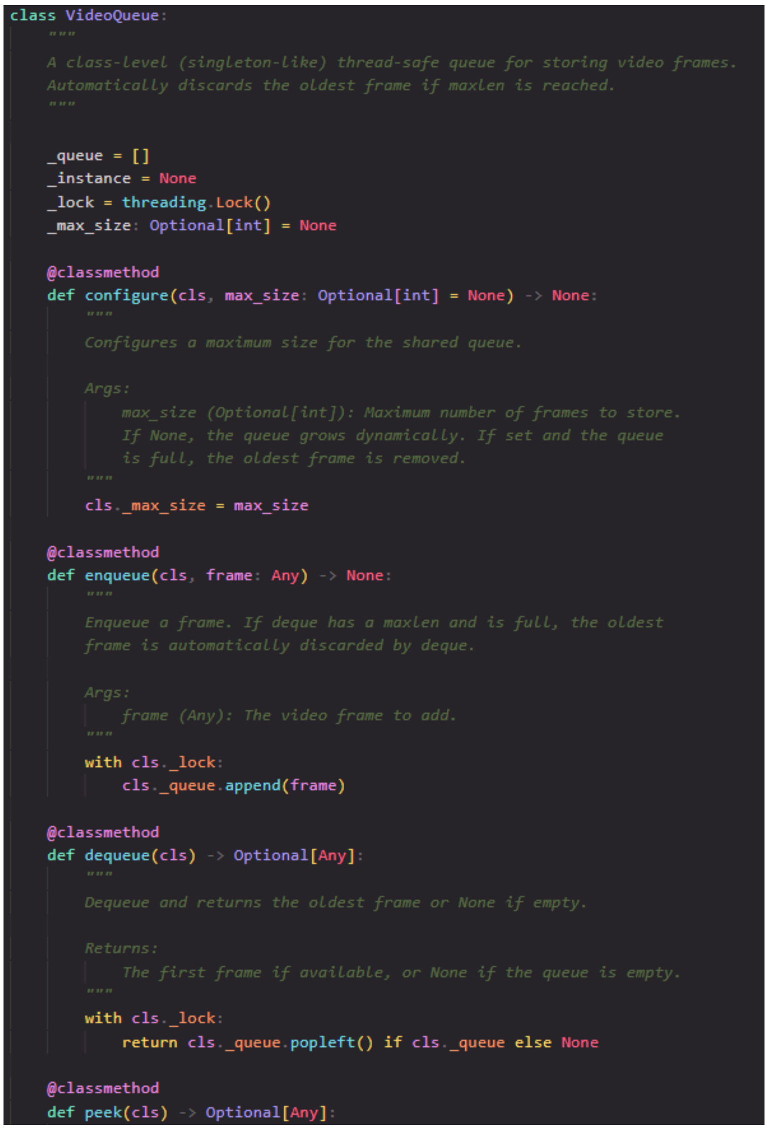
and had the object detection overlays baked into the footage. This feature unexpectedly turned out to be the hardest to implement.
Initially there was difficulty in implementing this function of the application because each operating system works in a slightly different way when saving video formats due to the required codecs.

During the development of this feature I realised that skipping frames on the frame capture method resulted in faster than real-time playback of the video. To fix this a refactor was required, the frame skipping was moved from the frame capture method to the frame processing method. This maintained the improved performance whilst allowing the video to be played back to the user at real time. This also allowed the Archive Processor to maintain the integrity of the processed file more in line with the original provided video with no loss of data.

During the initial development of the archive processor the output video codec would have to be manually changed when testing the feature between different operating systems by altering the code. This was solved by changing the codec one that would work in an operating system agnostic way. For this reason, MP4V was chosen.

The next difficulty was using the VideoQueue class that had been previously implemented. Previously, the VideoQueue class had been implemented as a regular class that can have may instances of the class as objects. In this project the VideoQueue class needed to be implemented as a singleton, meaning that there can only be a single instance of the class active in the program at any one time. It required some significant research and refactoring to the class to ensure that it could only be instantiated once. To achieve this class decorators were used calling the @classmethod decorator. This runs additional code before the method letting python know that the method modifies the class as a whole rather than a specific instance.

Instead of passing 'self' to the class methods as is standard practice the class itself is passed in as seen in figure 33 below.



```python
class VideoQueue:
    """
    A class-level (singleton-like) thread-safe queue for storing video frames.
    Automatically discards the oldest frame if maxlen is reached.
    """

    _queue = []
    _instance = None
    _lock = threading.Lock()
    _max_size: Optional[int] = None

    @classmethod
    def configure(cls, max_size: Optional[int] = None) -> None:
        """
        Configures a maximum size for the shared queue.

        Args:
            max_size (Optional[int]): Maximum number of frames to store.
            If None, the queue grows dynamically. If set and the queue
            is full, the oldest frame is removed.
        """
        cls._max_size = max_size

    @classmethod
    def enqueue(cls, frame: Any) -> None:
        """
        Enqueue a frame. If deque has a maxlen and is full, the oldest
        frame is automatically discarded by deque.

        Args:
            frame (Any): The video frame to add.
        """
        with cls._lock:
            cls._queue.append(frame)

    @classmethod
    def dequeue(cls) -> Optional[Any]:
        """
        Dequeue and returns the oldest frame or None if empty.

        Returns:
            The first frame if available, or None if the queue is empty.
        """
        with cls._lock:
            return cls._queue.popleft() if cls._queue else None

    @classmethod
    def peek(cls) -> Optional[Any]:
```

Figure 37: VideoQueue Class

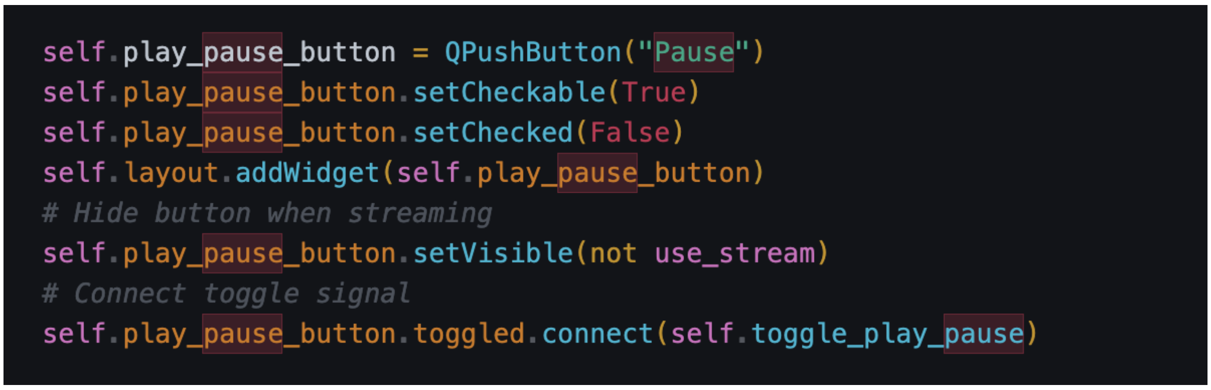
The instance property of the VideoQueue class ensures that there is only one instance of the class. The _lock property is important to ensure that the queue functions correctly in the multi-threaded environment it is being used in. _lock functions similarly to a database lock that prevents the writing of data whilst another process is writing or removing data. This helps to ensure data consistency when enqueuing and dequeuing.

9.13 ***Play / Pause***

The next feature goal was to add the ability to pause and resume playback of pre-recorded videos in the app. This was an important quality of life feature to add to the application because it allows the user to freeze a potentially important frame. Initially I wanted to include the ability to scrub forwards and backwards through the video but due to the way the video had been implemented this was impossible without major, foundational code structure and functional changes. As the video player class was not designed with this functionality in mind.

The actual implementation of the play / pause is simplistic but required careful forethought in-order function correctly with the many moving parts of the program. Specific attention was paid to how the pausing would affect the archival feature.

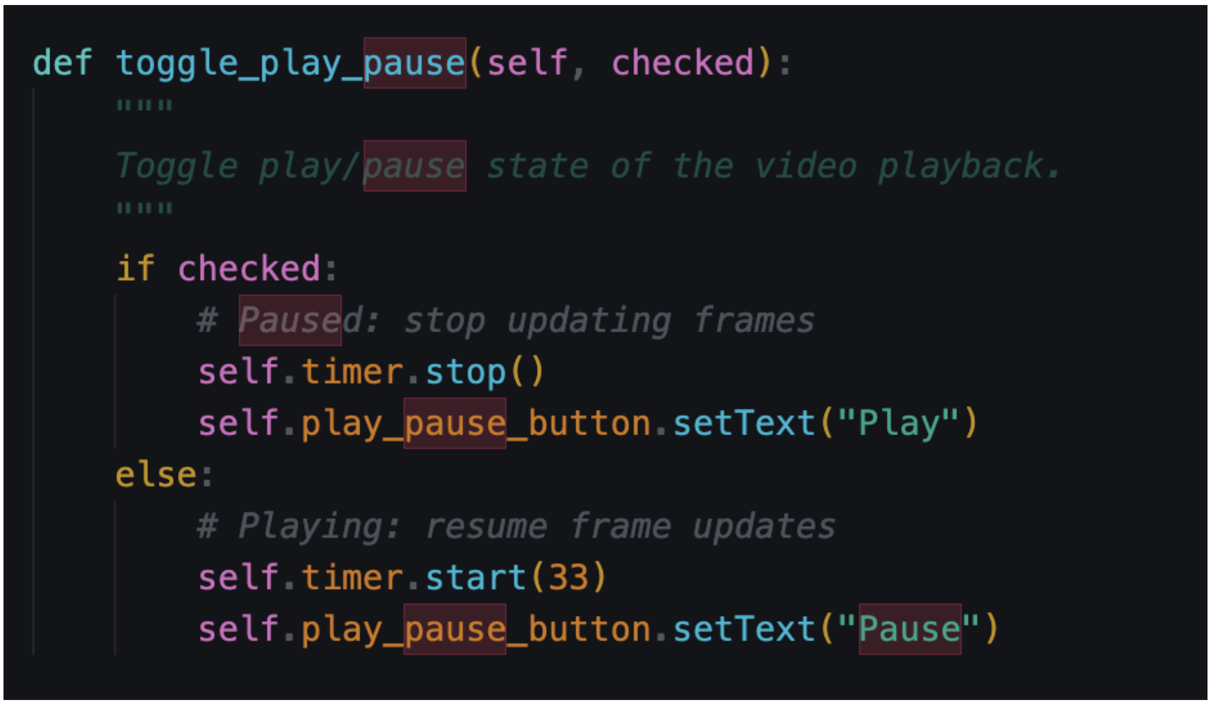
The first element added was a button that the user interacts with. This was implanted by added a QPushButton to the video player layout.



```
self.play_pause_button = QPushButton("Pause")
self.play_pause_button.setCheckable(True)
self.play_pause_button.setChecked(False)
self.layout.addWidget(self.play_pause_button)
# Hide button when streaming
self.play_pause_button.setVisible(not use_stream)
# Connect toggle signal
self.play_pause_button.toggled.connect(self.toggle_play_pause)
```

Figure 38: Adding Play / Pause Button

When the user presses this button, it calls the toggle_play_pause method of the VideoPlayer class.



```python
def toggle_play_pause(self, checked):
    """
    Toggle play/pause state of the video playback.
    """
    if checked:
        # Paused: stop updating frames
        self.timer.stop()
        self.play_pause_button.setText("Play")
    else:
        # Playing: resume frame updates
        self.timer.start(33)
        self.play_pause_button.setText("Pause")
```

Figure 39: toggle_play_pause Method

This method stops the multi-threading timer which will stop the processing of frames in the worker threads. In order to start video processing and playback again the timer is restarted with a framerate of 33 frames per second. Alongside the pausing of the timer the text in the button is updated to relay its function when pressed a subsequent time.

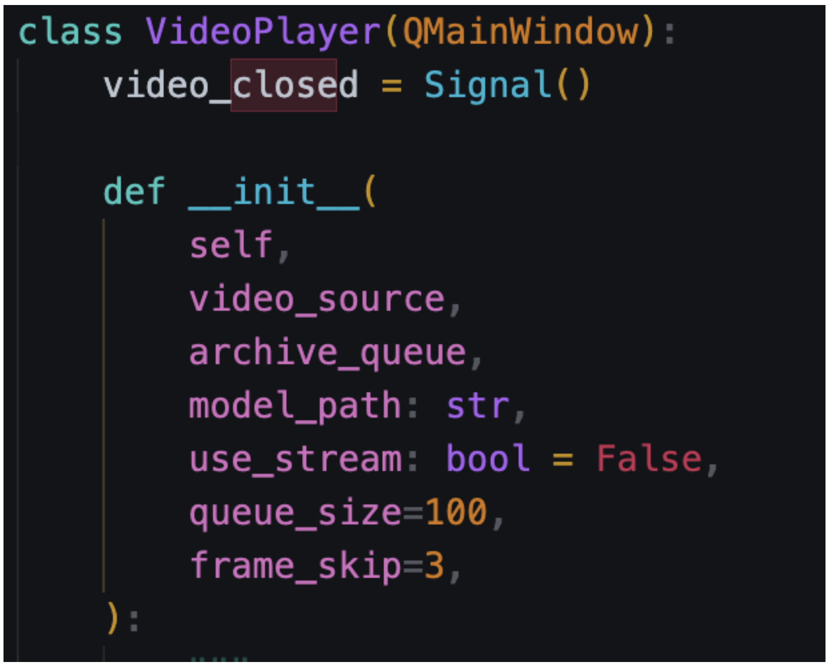
Initially during the development of this feature there were issues because the implementation was attempting to add a check to the get_frame method of the VideoProcessor. This caused issues because the archive_processor and model processing were still running meaning that errors were being generated when the video was paused because the VideoProcessor was no longer providing frames to the downstream processors.

9.14 ***Close Video***

Previously, upon a finishing a pre-recorded video or terminating the connected livestream video the last frame of the video would be permanently visible in the program. To fix this a close function was added that can be invoked by the user at any time before, during or after the playback of the video. This is a very important feature for the user experience because it allows the user to process multiple videos without having to restart the program. Additionally, it solved the issue of multiple video player instances being able to be open in the program at one time.

The initial implementation for the close method was overly complex because it involved trying to destroy the class. After attempting this route for the feature, it was realised that I could just call the parent class's close function and emit a signal to let any other processes know that the VideoPlayer class was terminating. For example, the metadata viewer knows to close itself when the video is closed.

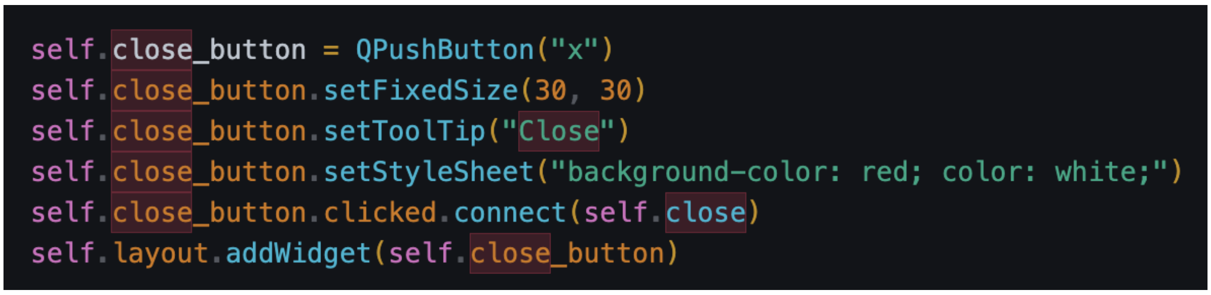
The signal was added to the top level of the class, outside of the constructor.



```python
class VideoPlayer(QMainWindow):
    video_closed = Signal()

    def __init__(
        self,
        video_source,
        archive_queue,
        model_path: str,
        use_stream: bool = False,
        queue_size=100,
        frame_skip=3,
    ):
```

Figure 40: Closure signal

The button itself was added via a QPushButton that calls the close method of the class.



```python
self.close_button = QPushButton("x")
self.close_button.setFixedSize(30, 30)
self.close_button.setToolTip("Close")
self.close_button.setStyleSheet("background-color: red; color: white;")
self.close_button.clicked.connect(self.close)
self.layout.addWidget(self.close_button)
```

Figure 41: Close button QPushButton

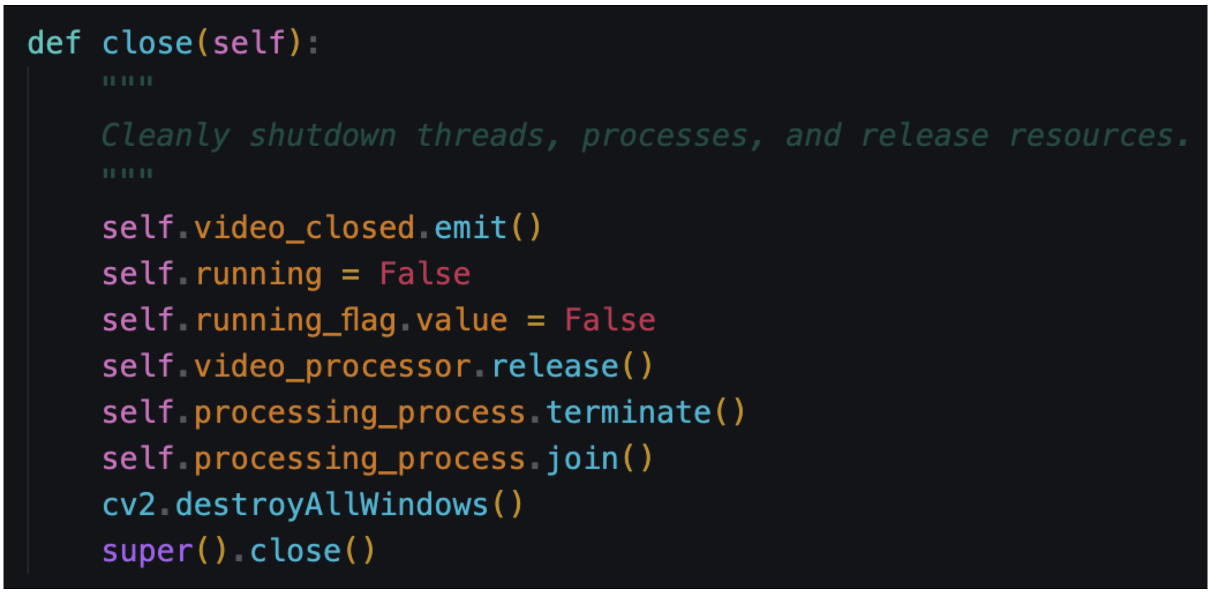

```python
def close(self):
    """
    Cleanly shutdown threads, processes, and release resources.
    """
    self.video_closed.emit()
    self.running = False
    self.running_flag.value = False
    self.video_processor.release()
    self.processing_process.terminate()
    self.processing_process.join()
    cv2.destroyAllWindows()
    super().close()
```

Figure 42: VideoPlayer close method

The super().close() calls the close function of the QMainWindow. This kills the child QWidget (VideoPlayer in this case.)

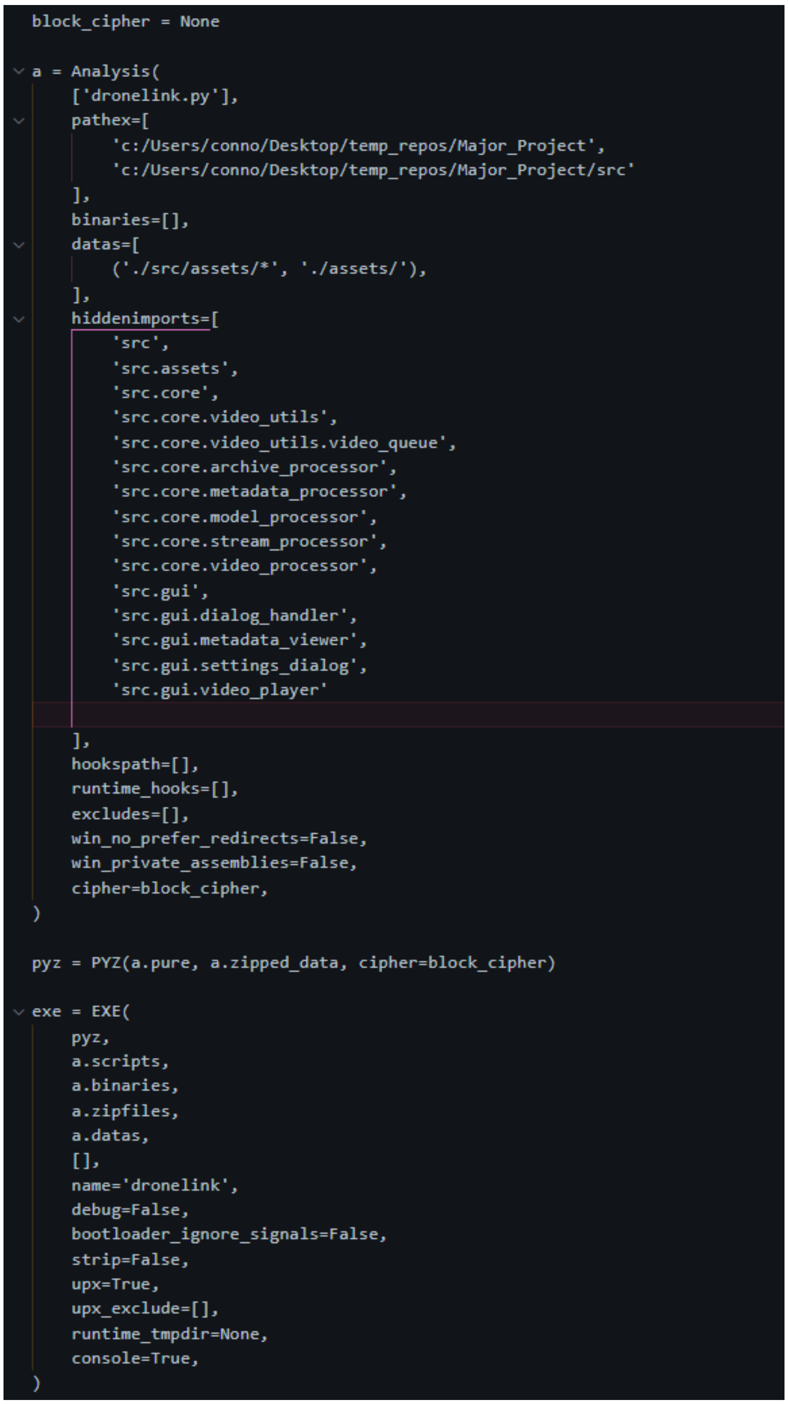
9.15 *Binary Compilation*

For a program to be effectively distributed, it should be compiled into a standalone executable. PyInstaller, a Python library, facilitates this process by packaging Python applications with all their dependencies into a single executable file. “PyInstaller bundles a Python application and all its dependencies into a single package. The user can run the packaged app without installing a Python interpreter or any modules.” (PyInstaller, n.d.)

Packaging the application into a single executable enhances user-friendliness significantly. Users are not required to install Python or manage dependencies manually, actions which typically demand technical expertise. Instead, most users are already comfortable with executing applications from a binary executable file.

9.15.1 *Creating a binary*

One of the methods that can be used to create a binary of a Python program is to create what is called a Spec file. A Spec file explicitly instructs PyInstaller on the compilation process, detailing the source files, necessary dependencies, included data files, and additional configurations.



```
block_cipher = None

a = Analysis(
    ['dronelink.py'],
    pathex=[
        'c:/Users/conno/Desktop/temp_repos/Major_Project',
        'c:/Users/conno/Desktop/temp_repos/Major_Project/src'
    ],
    binaries=[],
    datas=[
        ('./src/assets/*', './assets/'),
    ],
    hiddenimports=[
        'src',
        'src.assets',
        'src.core',
        'src.core.video_utils',
        'src.core.video_utils.video_queue',
        'src.core.archive_processor',
        'src.core.metadata_processor',
        'src.core.model_processor',
        'src.core.stream_processor',
        'src.core.video_processor',
        'src.gui',
        'src.gui.dialog_handler',
        'src.gui.metadata_viewer',
        'src.gui.settings_dialog',
        'src.gui.video_player'

    ],
    hookspath=[],
    runtime_hooks=[],
    excludes=[],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
)

pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)

exe = EXE(
    pyz,
    a.scripts,
    a.binaries,
    a.zipfiles,
    a.datas,
    [],
    name='dronelink',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=True,
)
```

Figure 43: DroneLink Spec File

As seen in figure 43, the spec file is structured into sections each with an important role.

- Analysis: This part identifies the script (dronelink.py) intended for compilation and outlines paths (pathex) for source files and dependencies. It specifies additional data files required at runtime under the datas section, like asset files (src/assets). The hidden imports list explicitly includes modules and submodules that PyInstaller may not automatically detect due to dynamic imports or indirect usage.

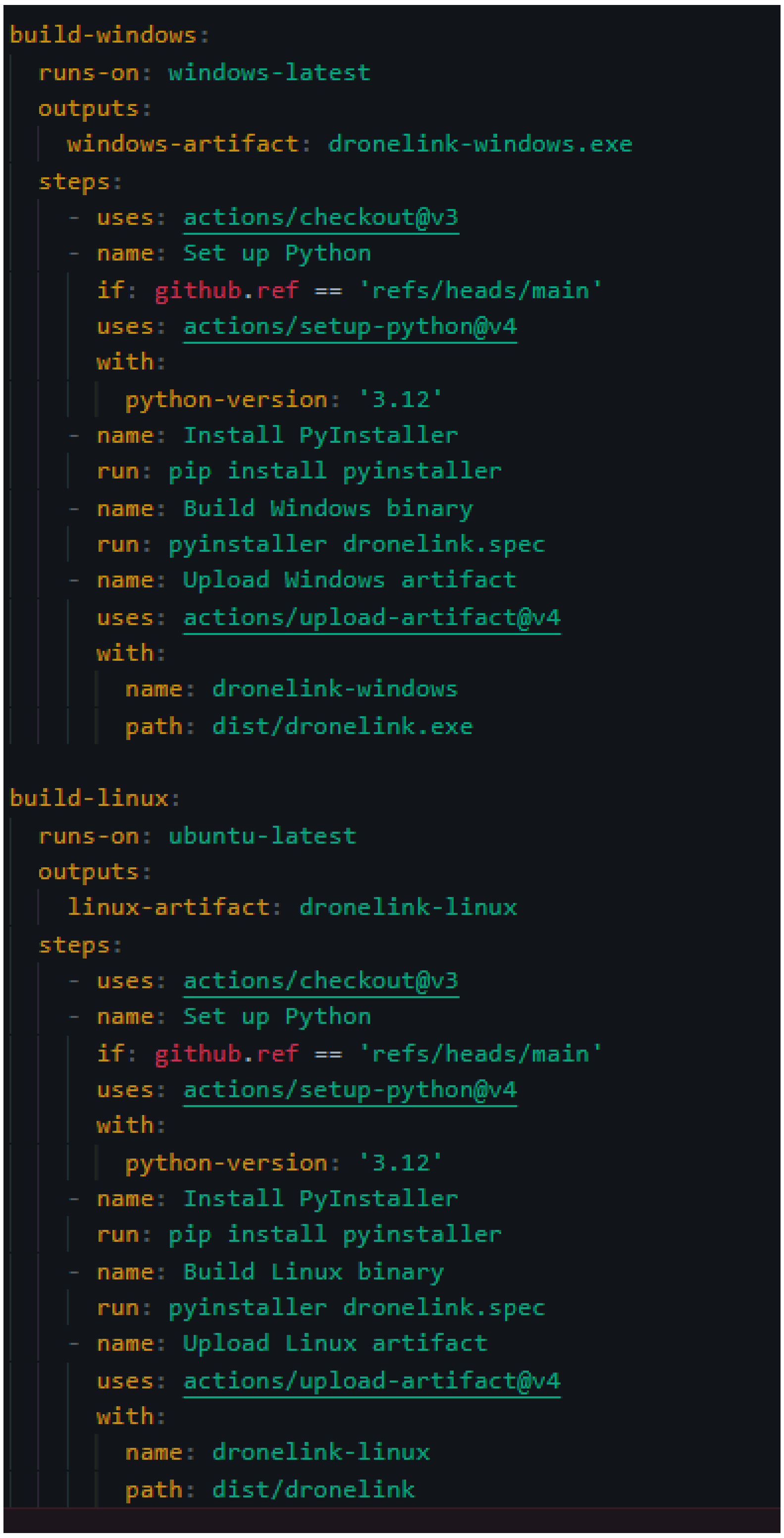
- PYZ: This section packages Python bytecode compiled from the scripts identified during analysis into a zipped archive. It encapsulates the logic and dependencies of the application, improving efficiency and load performance at runtime.
- EXE Section: This is the final step where PyInstaller generates the executable binary. It bundles scripts, binaries, zipped data, and additional metadata from previous sections. Configurations here include defining the executable's name (DroneLink), controlling debugging behaviour (debug=False), and determining whether a console window will appear upon execution (console=True). UPX (ultimate packer for executables) compression (upx=True) is also applied here to reduce the size of the executable, improving portability and download speed. UPX is a compression tool solely designed to reduce file size of executable files.

The created binary will only run on the operating system architecture that it was created on. For instance, a binary created on a Windows machine will only run on other Windows (Win32) machines, likewise with Ubuntu and MacOS. MacOS binaries require the binary to be created on the oldest version of the operating system you wish to support as new versions of the OS have backwards compatibility but in inverse may not be true.

9.15.2 *Automating Compilation*

Instead of manually having to create binaries for each new version of the tool, it is possible to automate the compilation of the code into binaries by using the CI pipeline. The pipeline can then use containerised instances to create the binaries for the desired operating systems when code is merged into the main branch.

This is achieved by defining jobs in the GitHub actions CI YAML file.



```
build-windows:
  runs-on: windows-latest
  outputs:
    windows-artifact: dronelink-windows.exe
  steps:
    - uses: actions/checkout@v3
    - name: Set up Python
      if: github.ref == 'refs/heads/main'
      uses: actions/setup-python@v4
      with:
        python-version: '3.12'
    - name: Install PyInstaller
      run: pip install pyinstaller
    - name: Build Windows binary
      run: pyinstaller dronelink.spec
    - name: Upload Windows artifact
      uses: actions/upload-artifact@v4
      with:
        name: dronelink-windows
        path: dist/dronelink.exe

build-linux:
  runs-on: ubuntu-latest
  outputs:
    linux-artifact: dronelink-linux
  steps:
    - uses: actions/checkout@v3
    - name: Set up Python
      if: github.ref == 'refs/heads/main'
      uses: actions/setup-python@v4
      with:
        python-version: '3.12'
    - name: Install PyInstaller
      run: pip install pyinstaller
    - name: Build Linux binary
      run: pyinstaller dronelink.spec
    - name: Upload Linux artifact
      uses: actions/upload-artifact@v4
      with:
        name: dronelink-linux
        path: dist/dronelink
```

Figure 44: Binary Compilation Jobs

As seen in Figure 44, the two CI jobs, build-windows, and build-linux will use the previously created spec file (Figure 43) in a docker instance. They achieve this by building an image from the windows/ubuntu-latest image and then installed the required Python version along with the needed packages. In this case that is

PyInstaller. After package installation, PyInstaller is run with the spec file provided in the repository. After compilation the binaries are available for download as a upload artifact.

9.15.3 ***Conclusion***

In summary, compiling a Python application into a standalone executable using PyInstaller significantly improves the ease of software distribution and user accessibility. By clearly defining the compilation process through a Spec file and automating the creation of executables via Continuous Integration pipelines, developers can efficiently produce consistent and reliable software binaries. This strategy not only simplifies deployment but also enhances the overall user experience by reducing complexity and dependency management.

9.16 ***Implementation Conclusion***

During the implementation phase, the design ideas were transformed into a fully working, modular application that could track and detect objects in drone footage in real time. Class-based design, multiprocessing pipelines, and a non-blocking signal-slot communication mechanism were used to prioritise performance, responsiveness, and maintainability. While threading and multiprocessing facilitated effective frame recording and model inference without affecting user interaction, PySide6 integration allowed for a responsive and user-friendly GUI.

Following iterative training with Ultralytics Hub and Google Colab Pro, deep learning models were trained and reviewed, and optimised models suited for high-performance and resource limited scenarios were chosen. Important features including model selection, metadata extraction, live and pre-recorded video processing, and the ability to archive annotated material were created gradually and verified by both automated and manual testing.

Despite challenges such as platform-specific codec support and GUI limitations, the final implementation delivered a reliable and extensible foundation for drone-based SAR tools. The modular structure ensures that future enhancements, such as geolocation integration or cloud-based analytics, can be incorporated with minimal architectural disruption.

10 Testing

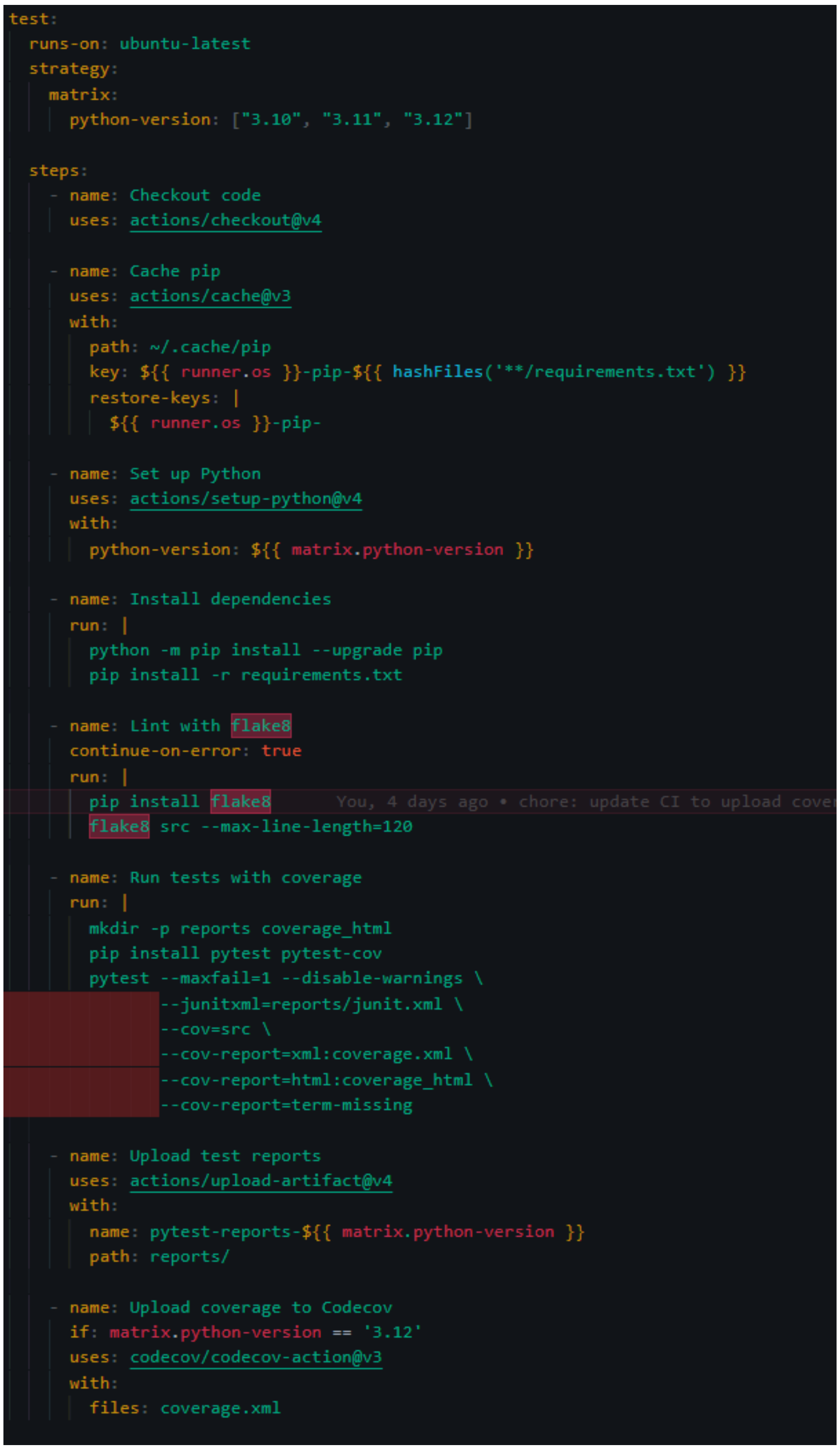
Throughout the project multiple methods were used to test and validate the usability, performance, and stability of the application across its various components. Given the program's complexity, ranging from GUI responsiveness and multi-threaded video handling to real-time object detection; testing was approached in an iterative and modular manner. Early testing focused on manual testing of key features, whilst later testing relied heavily on automated unit testing of methods, classes and functions for ensuring the functionalities of the program continued to work as expected.

In addition, user interface testing ensured that controls such as model selected, video playing and settings management operated as expected across different scenarios and edge cases. Continuous Integration pipelines were used to automate linting, test execution across multiple Python versions to catch platform specific errors. Manual testing was also conducted to attempt to simulate real-world SAR use cases and verify the reliability of object detection in varied environments.

10.1 ***Unit Testing***

Unit testing was used to verify the function of individual components of the application against a ground truth. In particular the elements that handled the processing side of the application in contrast to the GUI. The tests focused on ensuring that the processors in the application continued to output expected values and to guarantee that any changes would be durable.

Unit tests we implemented using PyTest, a testing framework for the Python language. To ensure the tests were always consistent across changes to the local Python and project environment, the tests were run as part of a containerised environment in a continuous integration pipeline provided by GitHub Actions. This container would be completely isolated from my local system as it runs in the cloud and thus separated from any changes or quirks of my local system. This meant that every time the tests were run it was in the exact same environment providing a solid foundation for the tests to be accurate.



```yaml
test:
  runs-on: ubuntu-latest
  strategy:
    matrix:
      python-version: ["3.10", "3.11", "3.12"]

  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Cache pip
      uses: actions/cache@v3
      with:
        path: ~/.cache/pip
        key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}
        restore-keys: |
          ${{ runner.os }}-pip-

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Lint with flake8
      continue-on-error: true
      run: |
        pip install flake8      You, 4 days ago • chore: update CI to upload cove
        flake8 src --max-line-length=120

    - name: Run tests with coverage
      run: |
        mkdir -p reports coverage_html
        pip install pytest pytest-cov
        pytest --maxfail=1 --disable-warnings \
               --junitxml=reports/junit.xml \
               --cov=src \
               --cov-report=xml:coverage.xml \
               --cov-report=html:coverage_html \
               --cov-report=term-missing

    - name: Upload test reports
      uses: actions/upload-artifact@v4
      with:
        name: pytest-reports-${{ matrix.python-version }}
        path: reports/

    - name: Upload coverage to Codecov
      if: matrix.python-version == '3.12'
      uses: codecov/codecov-action@v3
      with:
        files: coverage.xml
```

Figure 45: CI Test Job

The ‘test’ job runs in a ubuntu container and will run the tests a total of 3 times on different Python versions. This is to catch errors that may be related to the version of Python the user is running if building the application from source.

The CI ‘test’ job had a number of steps.

- Firstly, the code is checked out using the checkout@v4 GitHub action. This checks out the code to the container
- If the container has previously ran in its current configuration, then the pip packaged are loaded from the cache GitHub action. This prevents having to redownload the packages every time the job is run, reducing the time it takes to complete the job.
- Python is then installed to the container according to the matrix defined in the strategy key.
- The PIP packages are then installed into the Python environment.
- Linting is performed on the Python code to ensure that the code conforms to Pythonic code standards and that there is not redundant code. For example, it catches unused imports and variables that are defined but never used. This step ensures that any submitted code is of a high quality.
- The PyTest unit test suite is then run on the code. This step will also create a coverage report. This gives information on how much of the code is covered by tests and can help highlight areas that need more tests to special attention.
- The coverage report is then uploaded as a job artifact to GitHub, allowing developers to view the report after the job has been completed.

10.2 *Integration Testing*

Integration testing was considered to validate the interactions between the processing and GUI logic. Given the real time nature of the application, it was important that the UI correctly handled all the core functions.

The primary tool for integration testing was PyTest-Qt. PyTest-Qt is an extension to PyTest that facilitates GUI testing by simulating user interactions with Qt widgets. This would allow the test suite to mimic actual user behaviour such as clicking buttons, selecting models and beginning processing on a video.

However, due to the limitations on running Qt applications inside headless, dockerised CI environments, these integration tests were not implemented. Qt applications require a display server in order to function. There are some tools (such as xvfb) that simulate a display server, allowing a dockerised application. The real-world implementation of this, on the other hand, turned out to be vastly more complex than initially thought. Especially with the added complexity of handling multi-threading. Instead, the GUI would be manually tested

10.3 *Manual Testing*

During the course of development, manual testing was conducted on both the graphical user interface and the object detection model. This testing process involved verifying that each feature of the GUI operated correctly during regular usage, ensuring that expected user workflows, such as loading models, managing video playback, and adjusting settings, functioned without errors or unexpected behaviour. Throughout development, special attention was also paid to identifying and executing potential edge cases, such as providing invalid file inputs, abruptly terminating video streams, or adjusting settings during active detection, to observe how the application handled unexpected or abnormal user actions.

The manual testing approach for the GUI emphasized not only validating the correct behaviour under standard conditions but also challenging the system's robustness under unusual scenarios that a real-world user might encounter. Testing sessions frequently involved switching models mid-playback, rapidly changing settings, introducing corrupt media files, and forcing resolution changes to validate that the application could recover gracefully without requiring a restart or leading to a crash.

In addition to the GUI, the model's behaviour was manually tested by running it against a variety of video samples representing different environments and conditions. These tests verified the model's ability to maintain detection accuracy under varying lighting conditions, image qualities, and object densities. Real-world SAR video data were incorporated to ensure that object detection results were consistent, reliable, and free of critical failures such as frame freezes or detection hangs.

Manual testing cycles were performed iteratively throughout the project's development, with observations carefully noted and used to guide bug fixing and feature refinement. This manual validation process played a crucial role in ensuring the overall stability, usability, and performance of the final application

11 Further Development

There are many potential areas for further development of the DroneLink application. Whilst the current implementation focuses on real time object detection and tracking with the goal of aiding Search and Resue applications.

A clear path for growth in the project would be to tailor the system for different operational demands in sectors such as defence, policing, surveying and others. Each of these sectors has unique requirements that could be met though the publishing of modules for the base program, introducing features tailored to each industry.

For instance, in the defence sector there is a demand for autonomous systems capable of target recognition, tracking hostile assets and conducting semi or fully autonomous reconnaissance. DroneLink could be expanded to include model training on camouflaged personnel, military equipment or in structure detection to map out possible static defences. Combining this further with tools to autonomously control the drone based on its detections and classifications could be a desirable feature for this sector.

Similarly, for policing applications, the software could be optimised for urban environments, with software capabilities added to aid in crowd monitoring, traffic control or with facial recognition to aid in the tracking of persons of interest. Further expansion of the metadata extraction tool in the application would be desirable for this use case to aid in the building of a case against an individual.

Additionally, surveying features could be added to the software to allow for not only human detection but for infrastructure analysus, for example, the model could be trained to identify potential faults in buildings, bridges or other infrastructural buildings. Integration with Geographic Information Systems (GIS) would enable the automatic mapping of detected features would add a huge capability to the program.

12 Conclusion

The aim of this project was to develop a real-time item recognition and tracking system to support drone-based Search and Rescue (SAR) operations. Throughout the project, I had to learn and apply principles of modularity, performance optimisation, usability, and maintainability, recognising their importance for real-world SAR applications where dependability and efficiency are critical.

During the initial research phase, a gap was identified in the availability of open, flexible, and cost-effective SAR solutions. Addressing this gap required learning about recent advances in AI, computer vision, and object detection technologies. I designed and implemented a full system capable of processing both live and recorded video streams, incorporating object detection and multi-object tracking using DeepSORT. I developed independent and concurrent modules for video processing, model inference, and metadata extraction through multithreading and multiprocessing, gaining practical, professional grade experience in managing parallel systems

Model training and optimisation involved learning to use platforms such as Ultralytics Hub and Google Colab Pro. This allowed me to create models specifically adapted to drone footage, balancing detection accuracy against computational demands. For the user interface, I studied and implemented PySide6 to ensure that SAR personnel could operate the system intuitively, without requiring specialist technical knowledge.

Although learning to use these platforms and packages was essential to the development of the project, the complexity and depth of the required learning to correctly implement many of the features significantly slowed down the initial development of the features. Substantial time had to be dedicated to understanding new frameworks, tools and best practices before implementing the features. This overall lead to less features being in the program than initially desired.

A significant focus was placed on system validation. I learned to integrate unit testing into a containerised Continuous Integration (CI) pipeline to maintain backend reliability across different Python environments. Manual testing complemented this process by validating the graphical interface, user workflows, and overall system behaviour under realistic and extreme conditions.

The project presented several challenges, such as the difficulties associated with GUI testing within Docker containers and the limitations on performance during video archiving. I researched and applied practical solutions to address these issues and outlined opportunities for further enhancements.

The final outcome is a fully functional and extensible application designed to assist SAR teams by reducing search times, minimising human error, and has the capability to provide a ‘force multiplier’ to SAR teams. This project reflects the critical role of machine learning, computer vision, and software engineering practices in creating effective, real world technological solutions

12.1 ***Final Words***

Beyond the technical outcomes of the project, the process of researching, designing, planning and developing this system has been a valuable opportunity to independently build my skills and to critically identify areas of weakness in my professional capacity. It has provided clearer insight into the challenges involved in bringing a major software project from concept to delivery under tight time constraints.

Many of the skills developed throughout this project, particularly in technical planning, self-learning, and system integration, will support my ongoing professional growth. Recognising the areas where my knowledge is currently limited will also allow me to more effectively target future learning and development

.

13 Bibliography

Aermatica3D. (n.d.). *LOC8 by USR*. Retrieved from Aermatica3D: https://www.aermatica.com/en/loc8-image-processing-software-2/

Amjoud, A. B., & Amrouch, M. (2023). Object Detection Using Deep Learning, CNNs. *IEEE*.

Barla, N. (2021, November 16). *The Complete Guide to Object Tracking [+V7 Tutorial]*. Retrieved from V7Labs: https://www.v7labs.com/blog/object-tracking-guide

Brookes, R. (2008, September 04). *MRMap Manual Chapter 05 – Basic Guidelines for using the MRMap.* Retrieved from MRMap: http://mrmap.org.uk/forum/uploaded_files/mrmap_chap05_mrmap_appl_v1.04_04.09.2008.pdf

Carmatec. (2024, September 26). *Difference Between Machine Learning and Deep Learning: A Comprehensive Guide*. Retrieved from Carmatec: https://www.carmatec.com/blog/difference-between-machine-learning-and-deep-learning/

Dukowitz, Z. (2024, June 14). *Search and Rescue Drones: A Guide to How SAR Teams Use Drones in Their Work*. Retrieved from UAV Coach: https://uavcoach.com/search-and-rescue-drones/

GeeksForGeeks. (2024, November 15). *Introduction to Recurrent Neural Networks.* Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

Jiao, L., Zhang, F., Liu, F., Tang, S., Li, L., Feng, Z., & Qu, R. (2019). A Survey of Deep Learning-based Object Detection. *IEEE*, 1-5.

Klingler, N. (2023, December 3). *Object Tracking in Computer Vision (2024 Guide)*. Retrieved from Viso.ai: https://viso.ai/deep-learning/object-tracking/

Kouidri, A. (2023, October 11). *Mastering Deep Sort: The Future of Object Tracking Explained*. Retrieved from ikomia: https://www.ikomia.ai/blog/deep-sort-object-tracking-guide

MathWorks. (n.d.). *What Is a Convolutional Neural Network?* Retrieved from MathWorks: https://uk.mathworks.com/discovery/convolutional-neural-network.html

Mattless, C. (2024). *Investigating Object Detection, Tracking & its Applications in Search and Rescue.* IADT – Institute of Art, Design and Technology.

Mattless, C. (2024, December 11). *MP_video_processing.* Retrieved from Github: https://github.com/cmattless/MP_video_processing

McGee, J., Mathew, S. J., & Gonzalez, F. (2020). Unmanned Aerial Vehicle and Artificial Intelligence for Thermal. *2020 International Conference on Unmanned Aircraft Systems (ICUAS).* Athens: IEEE.

Modi, P. (2023, October 14). *Convolutional Neural Networks for Dummies*. Retrieved from Medium: https://medium.com/@prathammodi001/convolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f

MRMap. (n.d.). *Introduction*. Retrieved from MRMap: http://www.mrmap.org.uk/index.php/introduction

Murel, D. J., & Kavlakoglu, E. (2024, January 3). *What is object detection?* . Retrieved from IBM: https://www.ibm.com/think/topics/object-detection

Nehete, P., Dharrao, D., Pise, P., & Bongale, A. (2024). Object Detection and Classification in Human Rescue Operations: Deep Learning Strategies. *International Information and Engineering Technology Association*.

OpenCV. (n.d.). *About*. Retrieved from OpenCV: https://opencv.org/about/

PyInstaller. (n.d.). *PyInstaller Manual*. Retrieved from PyInstaller: https://pyinstaller.org/en/stable/

Shen, Y.-T., Kwon, H., Conover, D., Bhattacharyya, S., Vale, N., Gray, J., . . . Skirlo, F. (2023). Archangel: A Hybrid UAV-based Human Detection. *IEEE*.

Texsar. (2024, November 6). *Automated Drone Image Analysis Tool*. Retrieved from Texsar: https://www.texsar.org/automated-drone-image-analysis-tool/

USRI. (n.d.). *Home*. Retrieved from USRI: https://www.usri.ca/

Yeom, S. (2024). Thermal Image Tracking for Search and Rescue Missions with. *MDPU*.