CHALMERS
UNIVERSITY OF TECHNOLOGY

Drone Platform for Safety in Testing

Bachelor thesis in Electrical Engineering

Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Bachelor thesis 2025

Drone Platform for Safety in Testing

Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2025

Drone Platform for Safety in Testing
Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin

© Abdullah Arshad, 2025.
© Arvid Boisen, 2025.
© Jesper Eriksson, 2025.
© Viggo Forsell, 2025.
© Albin Hallander, 2025.
© Erik Rödin, 2025.

Supervisor:
Emmanuel Dean, Department of Electrical Engineering
Mikael Enelund, Department of Mechanics and Maritime Sciences
Robert Brenick, AstaZero
Examiner: Knut Åkesson, Department of Electrical Engineering

Bachelor Thesis 2025
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Picture shows a close-up shot of a DJI Mavic drone.
Photo by Pok Rie: Pexels.

Typeset in LaTeX
Gothenburg, Sweden 2025

Drone Platform for Safety in Testing
Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin
Department of Electrical Engineering
Chalmers University of Technology

Abstract

This thesis addresses the need for improved safety surveillance in autonomous vehicle testing by developing a drone platform for multiple drones. Extending single drone systems, this work focuses on coordinating multiple drones to provide wider area coverage at testing facilities. The backend of the developed system calculates drone positions based on test trajectories from ATOS, an open-source platform for automated vehicle testing, ensuring required sensor overlap. Real-time video is streamed from DJI drones via an Android app using Web Real Time Communication to a backend system. The backend processes the streams, performs image stitching to create a panoramic view, and applies object detection with estimated GPS coordinates for bikes, cars, and pedestrians. A web-based frontend allows users to monitor individual and merged video feeds, view detected objects, and control the drones. The system architecture is containerized using Docker. Experimental validations demonstrate the platform's ability to control drones, stream video in real-time, and provide extended surveillance with object detection. Key challenges and future improvements, including addressing hardware dependencies and automating configuration, are identified. The project represents a collaborative effort between Chalmers University of Technology and Pennsylvania State University.

Keywords: Drone Surveillance, Autonomous Vehicle Testing, Multi-Drone Systems, Object Detection, YOLO, WebRTC, Image Stitching, ATOS, AstaZero.

Drone Platform for Safety in Testing
Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin
Department of Electrical Engineering
Chalmers University of Technology

Sammandrag

Detta kandidatarbete behandlar behovet av förbättrad säkerhetsövervakning vid testning av autonoma fordon genom att utveckla en plattform med flera drönare. Som en vidareutveckling av tidigare system med endast en drönare, fokuserar detta arbete på att koordinera flera drönare för att ge täckning över större områden vid testanläggningar som AstaZero. Systemet beräknar optimala drönarpositioner baserat på testbanor från ATOS-systemet och säkerställer nödvändigt kameraöverlappning. Realtidsvideo strömmas från DJI-drönare till ett backend-system via en Android-applikation som använder WebRTC. Backend-systemet bearbetar videoströmmarna, utför bildsammansättning (image stitching) för att skapa en panoramavy, och tillämpar objektdetektering med skattade GPS-koordinater. Ett webbaserat gränssnitt (frontend) möjliggör för användare att övervaka individuella och sammanfogade videoflöden, se objektdetekteringar och styra drönarna. Systemarkitekturen är uppbyggd med Docker-containrar. Validering visade att plattformen kan styra drönare, strömma video i realtid och erbjuda utökad övervakning med objektdetektering. Centrala utmaningar och framtida förbättringsmöjligheter identifieras, såsom att hantera hårdvaruberoenden och automatisera konfiguration. Projektet är resultatet av ett samarbete mellan Chalmers tekniska högskola och Pennsylvania State University.

Nyckelord: Drönarövervakning, Testning av autonoma fordon, Flerdrönarsystem, Objektdetektering, YOLO, WebRTC, Bildsammansättning, ATOS, AstaZero.

Acknowledgements

We wish to express our sincere gratitude to everyone who supported us throughout this thesis project.

Firstly, we are very grateful to our supervisors at Chalmers, Emmanuel Dean and Mikael Enelund. Their invaluable guidance, insightful feedback, and unwavering support were instrumental to our progress.

Secondly, we thank Robert Brenick from RISE AstaZero, who was consistently available to answer technical questions and generously provided us the opportunity to visit AstaZero's proving ground at RISE.

Lastly, our work benefited greatly from connections at Penn State. Our peers and collaborators there - Suryansh Agrawal, Shashank Bommareddy, Aidan Brown, Rahique Mirza, Jaden Peacock, Eugene Sosa, and Jack Volgren - deserve sincere appreciation. Their impressive work is admirable, and their readiness to meet and collaborate, often at short notice, was especially valued. Further thanks are extended to Darryl Farber for the valued invitation to the "Strategic Foresight and Global Governance of Critical Technologies & Sociotechnical Systems: Implications for Peace and Security 2025-2050" conference.

Preface

The work presented in this thesis stems from a significant international collaboration between Chalmers University of Technology (Chalmers) in Gothenburg, Sweden, and The Pennsylvania State University (Penn State) in State College, United States. From January to May 2025, six students from Chalmers and seven students from Penn State collaborated to achieve a multi-drone safety surveillance system for autonomous vehicles and a frontend interface for drone control.

This partnership relied on continuous communication through weekly virtual meetings, fostering a regular exchange of ideas, code, and footage. The collaboration was further strengthened by a visit from the Chalmers team to Penn State, where they participated in a productive joint workshop with their Penn State teammates.

The project's interdisciplinary nature is reflected in the participants' fields of study: Automation and Mechatronics, Mechanical Engineering, Computer Science, Computer Engineering, and Computational Mathematics. This diverse expertise proved crucial to the outcomes presented herein.

This thesis covers the work performed in the spring of 2025 within this collaboration.

Abdullah Arshad
Arvid Boisen
Jesper Eriksson
Viggo Forsell
Albin Hallander
Erik Rödin
Gothenburg, May 2025

List of Acronyms

Below are the acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
API	Application Programming Interface
ATOS	Autonomous Vehicle Test Operating System
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CD	Continuous Deployment
CI	Continuous Integration
DJI	Da-Jiang Innovations, drone designer and manufacturer
GitHub	Developer platform that allows developers to create, store, manage, and share their code
FOV	Field Of View
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment, used in WebRTC
JSON	JavaScript Object Notation
MAVlink	Messaging protocol for communicating with drones
ML	Machine Learning
NAT	Network Address Translation
OBB	Oriented Bounding Box
Open-source	Source code that is made freely available for possible modification and redistribution
RISE	Research Institutes of Sweden
ROS	Robot Operating System
SDK	Software development kit
SDP	Session Description Protocol, used in WebRTC
SIFT	Scale-Invariant Feature Transform
STUN-server	Session Traversal Utilities for NAT server
TCP	Transmission Control Protocol
TURN	Traversal Using Relays around NAT, used in WebRTC
YOLO	You Only Look Once
WebRTC	Web Real-Time Communication

Contents

1 Introduction **1**
- 1.1 AstaZero . . . 2
  - 1.1.1 Autonomous Vehicle Test Operating System (ATOS) . . . 2
- 1.2 Previous work . . . 3
  - 1.2.1 Identified points of improvement . . . 3
- 1.3 Objectives . . . 4
  - 1.3.1 Key project milestones . . . 4
- 1.4 Demarcations . . . 5

2 Theory **7**
- 2.1 Docker . . . 7
- 2.2 Redis and WebSocket . . . 7
- 2.3 DJI SDK . . . 8
  - 2.3.1 Control using DJI SDK . . . 8
  - 2.3.2 Video extraction . . . 9
- 2.4 Android App . . . 9
- 2.5 MAVLink . . . 9
- 2.6 Robot Operating System . . . 9
- 2.7 ATOS . . . 10
- 2.8 Web Real-Time Communication . . . 10
  - 2.8.1 Videostreaming with WebRTC . . . 11
- 2.9 Object detection . . . 11
- 2.10 Merging multiple video streams . . . 11
- 2.11 Threading . . . 12
- 2.12 Frontend . . . 14
- 2.13 CI/CD . . . 14
- 2.14 Hardware . . . 14
- 2.15 CUDA . . . 14

3 Methodology **17**
- 3.1 Project management . . . 17
- 3.2 Testing & Verification . . . 18
- 3.3 Version control . . . 18
- 3.4 Software . . . 18
  - 3.4.1 Docker . . . 19
  - 3.4.2 Drone Positioning . . . 20

3.4.3 Real-time video streaming . . . 24
3.4.4 Extended video surveillance . . . 25
3.4.5 Android app . . . 26
3.4.6 Frontend . . . 27
3.4.6.1 Drone control using the frontend . . . 28

4 Experimental Validation and Results 31
4.1 Verifying the Docker Setup . . . 31
4.1.1 Container Startup and Communication . . . 31
4.1.2 Monitoring and Debugging with Logs . . . 32
4.1.3 Ensuring Consistent and Reproducible Experiments . . . 32
4.1.4 Continuous Integration and Delivery . . . 34
4.2 Drone Control . . . 34
4.3 Real-time Video Streaming Using DJI Drones . . . 35
4.3.1 Integrated Functionality For Testing and Debugging . . . 37
4.4 Extended Video Surveillance . . . 38
4.4.1 Testing the Subsystem . . . 39

5 Discussion 41
5.1 Drone Control . . . 41
5.2 Data Serialization . . . 41
5.3 Endpoint vs Message Type . . . 42
5.4 Real Time Video Streaming In the Developed System and Its Limitations . . . 42
5.5 Extended Video Surveillance . . . 42
5.6 Suggestions on Future Work . . . 44
5.6.1 Communication . . . 44
5.6.2 Extended Video Surveillance . . . 44
5.7 Social and Ethical Aspects . . . 45

6 Conclusion 47

Bibliography 49

A Code I
A.1 Message type in backend . . . I

1 Introduction

In today's market, technical solutions to everyday problems and comfort standards are key points to remain competitive and compliant with regulations. Examples of such markets are the automotive industry, where advanced automated driving assistance and road safety are important [1].

Outside of Borås, Sweden, the company AstaZero operates a vehicle testing facility. At this site, autonomous vehicles can be tested in a wide range of traffic scenarios, including urban environments with multilane traffic and isolated country roads [2]. To ensure safe and efficient vehicle tests, the facility requires constant monitoring. By leveraging applied machine learning, drones can be used to observe test vehicles and detect behaviors that deviate from normal patterns. This enables the identification of potential hazards during autonomous vehicle testing, issues that might otherwise be difficult for a human operator to detect on their own.

Drone surveillance systems facilitate object detection across a variety of simulated traffic scenarios. By enabling the simultaneous operation of multiple drones, the system is capable of covering an extended testing area. Through this approach, a greater area can be covered while maintaining sufficient video resolution in the footage.

This project is a collaboration between AstaZero and a team of students from both Chalmers University of Technology hereafter Chalmers, and Pennsylvania State University hereafter Penn State. The purpose of this thesis is to document the continued development of a drone surveillance system [3]. The project will involve cross-disciplinary work, including DevOps, front- and backend development, as well as data processing.

1.1 AstaZero

This project is a collaboration with AstaZero, an organization owned by *Research Institutes of Sweden* hereafter RISE. RISE is an independent, state-owned research institute to strengthen Sweden's competitiveness and contribute to sustainable growth [2]. AstaZero operates a test facility, illustrated in Fig. 1.1, specifically designed for the development and evaluation of autonomous vehicle technologies. The facility enables simulation of a wide range of traffic environments and testing of advanced safety systems under a wide variety of conditions [4].

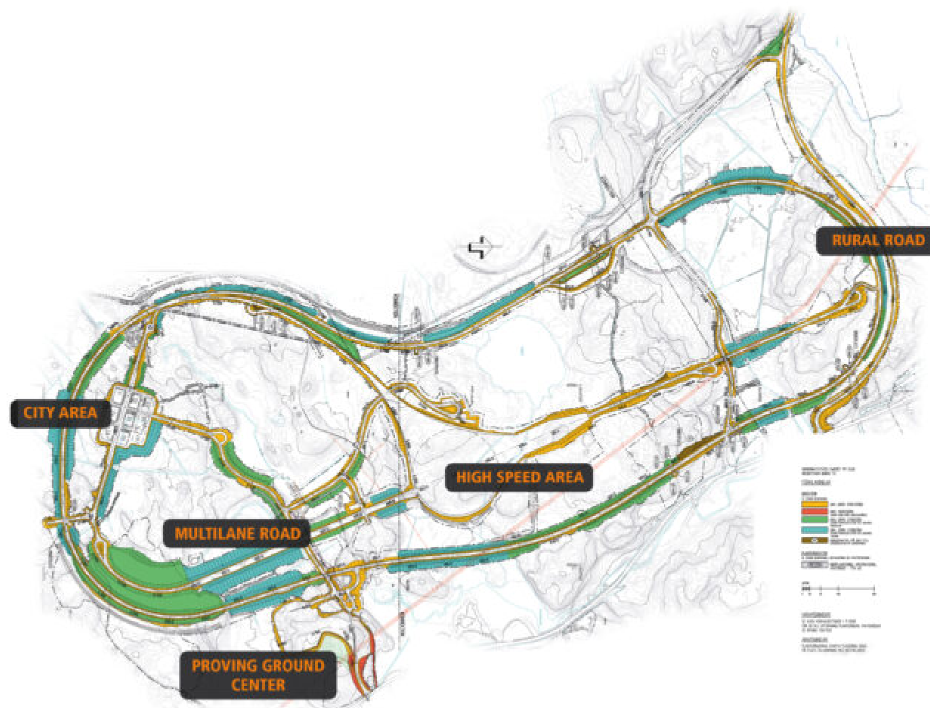

Figure 1.1: Map of AstaZero test facility [5].

1.1.1 Autonomous Vehicle Test Operating System (ATOS)

To operate the test facility, AstaZero has developed its own autonomous system, known as the *Autonomous Vehicle Test Operating System* (ATOS). This system functions as a central control hub and coordinator for scenario-based testing of autonomous vehicles.

ATOS determines the trajectories of the test participants, such as vehicles and dummy pedestrians, as well as the location of surrounding infrastructure. To ensure precise and repeatable test scenarios, ATOS uses an internal Cartesian coordinate system to position each participant accurately [6]. ATOS is built on the open-source software *Robot Operating System 2* (ROS 2), and the source code can be accessed on AstaZero's GitHub [7].

1.2 Previous work

The system previously developed by Mohi *et al.* [3] at Chalmers, illustrated in Fig. 1.2, comprises a single drone, a mobile app, a backend, and an ATOS module. The mobile app connects to the backend to enable comparison between the positions of objects detected by the drone camera and the corresponding positions provided by ATOS. If anomalies are detected, there is support for sending a command to ATOS to abort the test.

In addition, the communication software (backend) calculates the desired position of the drone. The mobile app facilitates communication between the drone and provides a *graphical user interface* (GUI) that allows users to operate the drone within system constraints by inputting coordinates and adjusting camera settings [3].

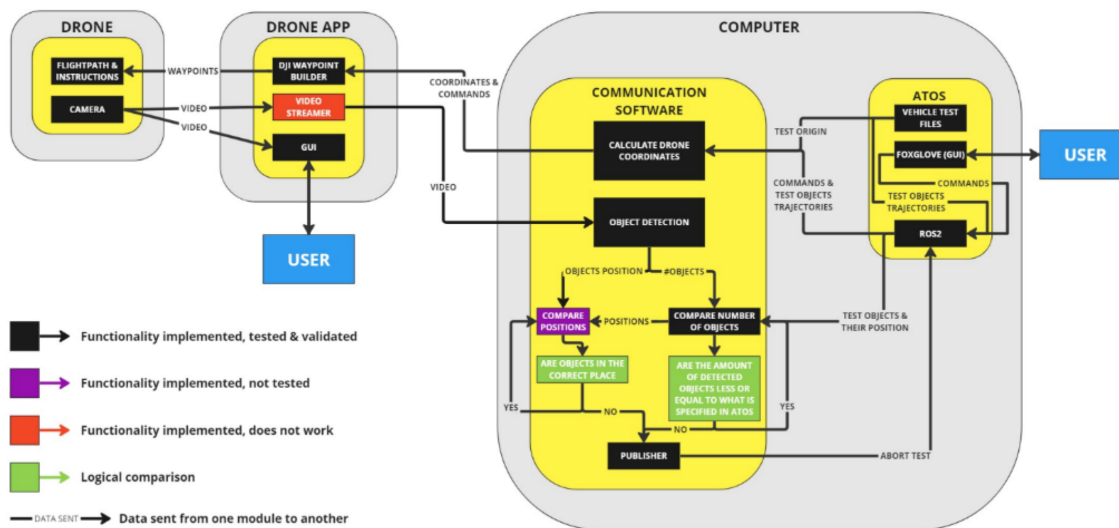

Figure 1.2: An overview of the system developed in previous work [3]. Some of the modules, *e.g.*, *Compare Positions* are not yet fully developed. Image taken from [3].

1.2.1 Identified points of improvement

The existing system's monitoring capabilities are constrained by the limited area that a single drone can cover. When operating at close range, the monitored area is insufficient for comprehensive data collection. However, operating at a greater distance results in inadequate image resolution for detailed analysis. A system that is capable of using multiple drones and thus multiple video streams during the test monitoring is required. This will allow a larger surveillance image of the test area in real time. Expanding the monitored area will enable the monitoring of larger test scenarios at AstaZero's facility, thereby increasing the system's overall usefulness.

Furthermore, surveillance with the current drone system is run on proprietary soft-

ware. This severely limits the applicability of the system since it requires a specific software to be developed for each specific hardware. Migrating the control system to an open-source communication layer is therefore also needed. This will significantly increase the availability of such a drone monitoring system. In addition, the system currently lacks support for real-time video streaming between the drone and the backend, which is essential for enabling real-time object detection. Hence, a low-latency video streaming solution needs to be implemented.

1.3 Objectives

The purpose of this project is to improve the testing capabilities of the AstaZero test track. This improvement involves the development of an autonomous system capable of controlling multiple drones, thereby enabling effective supervision of large areas through the integration of multiple video streams with different *Field Of View* (FOV). The drone control system, in conjunction with ATOS, will be utilized to efficiently plan drone waypoint missions and detect anomalies or unexpected behavior in test objects.

In previous work, students developed a system supporting a single drone, built on the existing software from *Da-Jiang Innovation* (DJI). This software allowed communication exclusively with DJI drones. To reduce reliance on foreign technology systems, which pose risks related to cybersecurity and information leaks, an open-source application will be developed to replicate the functionality of DJI's *Software Development Kit* (SDK). This will allow integration of drones from other manufacturers into a unified open-source platform. Additionally, a toggle switch will be implemented to enable seamless switching between between multiple control platforms.

1.3.1 Key project milestones

The key project milestones are outlined below, highlighting the major phases and deliverables critical to the project's success:

1. **Drone Formation:** Optimally position the drones using test trajectories from ATOS.
    (a) *Pose control of one drone using ATOS:* Implement and validate the positioning of one drone from test trajectories from ATOS.
    (b) *Pose control of multiple drones using ATOS:* Extend the control system to manage multiple drones simultaneously from the same ATOS instance.
    (c) *Positioning of drones for optimal test coverage:* Develop and test strategies or algorithms to determine the relative positions and orientations of the drones required to maximize the visual coverage of a designated area or target object, considering overlap, resolution, and potential occlusions.
2. **Combined Video Streams:** From two drones, create one single panoramic view.
    (a) *Merging of video streams:* Combine the frames of each stream into one seamless video.

   (b) *Object detection using merged video:* Apply a computer vision algorithm to the combined video feed to detect objects of interest within the wider field of view provided by the multi-drone system.

3. **Frontend Interface:** Developing a frontend interface where drone position and telemetry data are displayed. The individual drone feeds and the stitched video are also presented. Drone control is also possible from the frontend interface.
   (a) *Drone control:* Implement buttons which can arm, take off, and command the drone to return to its origin (home).
   (b) *Video streaming:* Receive individual video feeds from the drones.
   (c) *Drone location and telemetry:* Retrieve the position and telemetry from the drone, such as speed, height, and battery state. The data should be shown in the frontend interface.

4. **Toggle Switch:** Create an interface that can receive instructions and communicate with with different drone *application programming interface* (API).
   - *DJI Communication:* Successfully implement and verify bidirectional communication with DJI drones. This includes sending commands and receiving telemetry and video data.
   - *MAVLink Communication:* Implement and verify communication using the MAVLink protocol, enabling interaction with drones running compatible flight stacks.
   - *Control of one drone using simulation:* Develop and test the drone control algorithms within a simulated environment.
   - *Open Source Translation/Toggling Layer:* Design and implement an abstraction layer within the software architecture. This layer should allow the core application logic to interact with different drone types through a unified interface.
   - *Integrated control of multiple drones with merged video:* Control two drones of different types while processing and stitching their respective video streams.

1.4 Demarcations

- This project will focus on software development exclusively.
- The detection software will be able to detect bikes, pedestrians, and motor vehicles.
- Since testing and usage primarily occur during the daytime in the dry, the system will be tested only for such conditions.
- The drones will only be operated in an outdoor environment, and, in the testing phase, the system will only be evaluated with two drones due to limited access to drones.

2
Theory

This chapter presents the theoretical foundation necessary to carry out the project. It outlines key concepts, methods, and background knowledge about the project's design and implementation.

2.1 Docker

The software platform Docker can be used to build, test, and deploy applications quickly. Docker packages software into standardized units called containers that have all the dependencies the software needs to run, including libraries, system tools, code, and runtime [8]. A Docker image is a read-only template with instructions for creating a Docker container [9]. The image contains all of the files, binaries, libraries, and configurations to run the container [10].

A Docker container is a runnable instance of an image. By default, a container is relatively well isolated from other containers and its host machine [9]. This ensures that other containers cannot interact with each other, enhancing security. With isolation, the libraries and dependencies are the same between different hosts.

2.2 Redis and WebSocket

Redis is an in-memory key-value real-time database [11]. A key-value database uses a key-value pair to store data. The key is a unique string, and the value is an arbitrarily large data field that should be stored [12]. The Fig. 2.1 shows an example of a key-value pair.

Redis communicates over a *Transmission Control Protocol* (TCP) connection. This allows other clients across different machines to connect to the server. Redis supports a publish/subscribe paradigm, where publishers are not designed to send messages to specific receivers. Instead, messages are published on channels without awareness of whether there are subscribers. Subscribers, in turn, declare interest in one or more channels and receive only the messages relevant to those channels, without knowledge of the publishers' identities or activities [14]. This allows scalability across machines.

WebSocket is a communication protocol providing a full-duplex communication channel [15]. Both client and server can send requests using WebSockets, which is visualized in Fig. 2.2.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 2.1: Example of a key-value database [13] Clescop, CC BY-SA 4.0, via Wikimedia Commons.

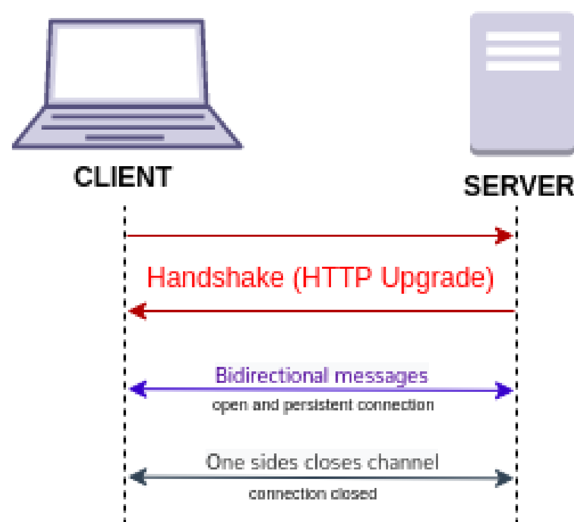

Figure 2.2: WebSocket connection [15] Brivadeneira, CC BY-SA 4.0, via Wikimedia Commons.

2.3 DJI SDK

The DJI Mobile SDK V4 is a library of classes that integrates with Android apps to control drones from a mobile device. [16]. The mobile SDK has a manager class that can be used to, for example, register the SDK and monitor nearby fly zones. The `Base` class contains a video feeder that manages the live feed from the DJI product to the mobile device. The `Product` and `Component` classes are used to get live telemetry and send commands to the products. The `Mission` class is used to construct custom missions for the drones, such as setting waypoints for the drones. The `Misc` class handles, for example, errors and diagnostics [16].

2.3.1 Control using DJI SDK

The drones utilize three primary flight modes: `Arm`, `Take Off`, and `Return to Home`. Arming the drone places it in a ready-for-takeoff state where the motors are enabled,

and safety checks are run to ensure the drone is ready to fly. The `Take Off` mode flies a preprogrammed waypoint mission. The waypoint coordinates are retrieved via a WebSocket connection from the backend. Return to Home mode flies the drone back to the position where it initially took off.

2.3.2 Video extraction

The `DJICodecManager` class is a part of the DJI Mobile SDK that provides methods for encoding and decoding video. By utilizing its functions `enabledYuvData` and `setYuvDataCallback`, YUV420 pixel matrices can be retrieved from the DJI drone's live video feed. The drone camera's video feed can be accessed using the VideoFeeder class provided by the DJI Mobile SDK. This is done in combination with VideoDataListener which is a callback interface that processes incoming video frames via its `onReceive` method [17, 18].

2.4 Android App

In the work presented by Mohi *et al.* [3], an Android app was developed for communicating between the backend and DJI Mobile SDK V4. The Android app uses a WebSocket to establish an asynchronous bidirectional connection. The Android app connects to the backend, which requires exposing ports on the host. Information exchange between the app and the backend is done in string format. A data interchange format that could be used in similar systems is *JavaScript Object Notation* (JSON), which is a data format that is used to store and transfer data using a "name-value" pair [19].

2.5 MAVLink

MAVLink is a very lightweight, header-only message library for communication between drones and/or ground control stations [20]. MAVLink is published under the LGPL license [21], which makes it suitable to use with an open source drone. Using an open-source drone offers multiple advantages, such as avoiding vendor lock-in, transparency, and customization. Compared to the DJI SDK described in Section 2.3, which is only compatible with drones made by DJI, MAVLink is compatible with multiple different drone manufacturers.

2.6 Robot Operating System

Robot Operating System 2 (ROS 2) is a further development of the original ROS 2 and is an open source software package specifically designed for robotics applications [22]. It offers a unified platform for robotic software development and is widely used today in the robotics industry. Because ROS 2 is open source, users have complete control over how it is implemented and can customize it to their specific needs [22]. It also works seamlessly with other software stacks and can be easily integrated into

existing systems. ROS 2 is fully open source, ensuring free access to an advanced and complete robotics SDK. [22].

2.7 ATOS

AstaZero has developed the *Autonomous Vehicle Test Operating System* (ATOS) to function as a control center and coordinator for scenario-based testing of autonomous vehicles. ATOS controls the test participants, such as vehicles and dummy pedestrians, as well as the surrounding infrastructure. To ensure precise and repeatable executions, ATOS utilizes GPS [6]. ATOS is built on ROS 2 and is open-source. The ATOS code is published on AstaZero's GitHub [7].

An ATOS-trajectory is defined as a dictionary of key-value pairs. The key is made up of a string describing the test participant, and the value consists of a list of objects in the *Coordinate* type, with attributes *lat*, *lng*, and *alt*. The list of coordinates describes the trajectory of the vehicle. These coordinates are in the ATOS internal Cartesian coordinate system and can be mapped onto latitude and longitude using the drone origin coordinate. The code implementation of this process can be found in Appendix B in [3]

2.8 Web Real-Time Communication

Web Real-Time Communication (WebRTC) is a low-latency streaming framework commonly used in applications for video calls, such as Google Meet. WebRTC provides a JavaScript API that enables direct integration [23]. In Python, several libraries have been developed to follow this API. One of them is `aiortc`, which enables Python-based integration of WebRTC [24]. Another benefit of using WebRTC is that it allows multiple peer-to-peer connections to be established on the same network. The connection between peers is established through what is commonly known as a "WebRTC handshake".

The first step in this process is for each peer to connect to a signaling server; for this, a WebSocket could be used. The first peer connects to a *Session Traversal Utilities for NAT* (STUN) server to obtain its *Network Address Translation* (NAT) information. Using the signaling server, the first peer then requests a *Traversal Using Relays around NAT* (TURN) server. After this, the first peer sends a *Session Description Protocol* (SDP) offer to the second peer, which responds with an SDP answer, both utilizing the signaling server to send these messages.

The first peer receives the SDP answer and sets up a remote description, which forms the basic network setup. This is followed by an *Interactive Connectivity Establishment* (ICE) candidate exchange between the peers. ICE candidates contain information about the peers and how they can discover one another on the network, enabling basic connectivity for video and audio transmission between the peers [23].

2.8.1 Videostreaming with WebRTC

Once the peer-to-peer connection is established, the WebRTC API provides support for streaming media between peers. The media stream is added asynchronously to the `RTCPeerConnection` by the first peer, attaching a media track. On the receiving side, the second peer retrieves the incoming stream using a listener event provided by the WebRTC API (ontrack event), which is triggered when a new media track is added to the connection. This enables the transmission of real-time video between peers [25]. WebRTC supports several video formats, including VP8, VP9, and H.264. Encoding in one of these formats can be managed using the `DefaultVideoEncoderFactory` provided by the WebRTC API [26]. For example, H.264 supports various pixel formats such as YUV420, YUV422, and YUV444 [27].

2.9 Object detection

Object detection is one of the most central and challenging tasks in computer vision and has undergone significant development over the past decades. Early approaches relied heavily on handcrafted visual descriptors such as the *Scale-Invariant Feature Transform* (SIFT) and the *Histogram of Oriented Gradients* (HOG), which provided a degree of robustness to scale and orientation variations [28, 29].

A particularly influential method in this domain is *You Only Look Once* (YOLO), which introduced a novel paradigm for object detection. In contrast to earlier architectures, such as R-CNN, that employ multiple sequential steps, including region proposal, classification, and post-processing, YOLO treats detection as a single regression problem. The model predicts both object classes and bounding box coordinates directly from raw image pixels in a single forward pass [30].

One of YOLO's key strengths lies in its ability to analyze the entire image during both training and inference. This holistic approach allows the model to implicitly incorporate contextual information, thereby reducing the likelihood of misclassifying background regions. This characteristic makes YOLO particularly well-suited for use in complex and dynamic environments where multiple objects may appear simultaneously [30].

2.10 Merging multiple video streams

To add an extended monitoring of the test area, the information from several cameras is combined, creating a panoramic view. This method aims to create a seamless video and is known as video stitching. As shown in Fig. 2.3 and Fig. 2.4, each camera captures a part of the scene from different angles. To successfully generate the stitched panorama see Fig. 2.5, an overlapping (FOV) between the two drone cameras is required. The overlap ensures that corresponding features can be aligned accurately to form a unified image.

To create a smooth transition between the video streams, the overlapping area in the cameras' FOV needs to be smoothed and blended. Fig. 2.6 and Fig. 2.7 shows

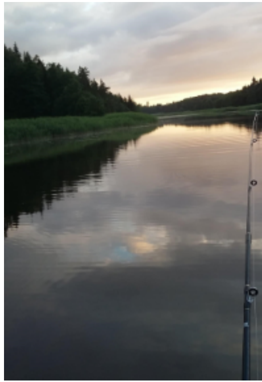

Figure 2.3: Footage left camera.

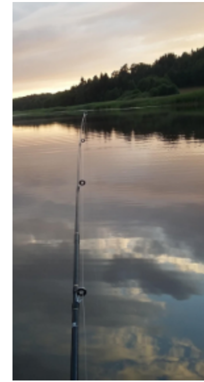

Figure 2.4: Footage right camera.

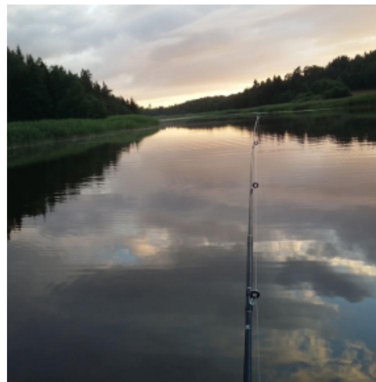

Figure 2.5: Stitched image.

the same area from two different perspectives and changes in angle. Fig. 2.8 presents the result after blending has been applied to combine them. Since the surveillance scenarios specifically involve the drones maintaining a static position, free from sharp changes in angle or perspective, a linear blending method can be used.

A very computationally efficient linear blending method is Alpha blending [31]. Alpha blending consists of weighting each individual pixel gradually in the overlapping area, where the coefficient α can take a value in the range $[0, 1]$ and is determined by how close the pixel is to the left edge of the right image or the right edge of the left image. The extreme value 0 means that only the left image is used, and the extreme value 1 means that only the right image is used.

2.11 Threading

In video processing, especially when it comes to real-time requirements or processing large amounts of data, the use of multithreading becomes a key optimization technique. By dividing the workload among multiple threads, the CPU's resources can be utilized more efficiently, reducing wait times and increasing the overall performance of the system.

A common implementation strategy for multithreaded programming is the producer-consumer pattern [32]. This design model is based on two separate threads, a

Figure 2.6: Left image before blending.

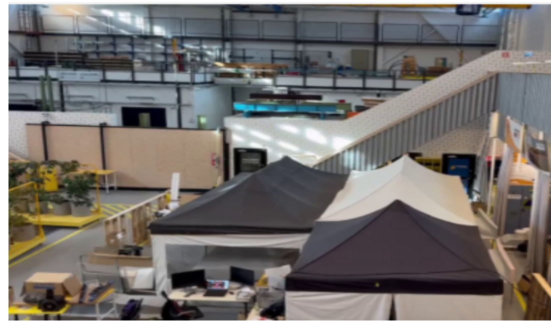

Figure 2.7: Right image before blending.

Figure 2.8: Image after blending.

producer and a consumer, that communicate via a shared data cache. The producer thread is responsible for generating or loading data, while the consumer thread processes this data. A wait/notify mechanism is used to manage synchronization: if the cache is full, the producer is blocked, and if it is empty, the consumer is blocked. This pattern allows for continuous work in at least one of the threads, which prevents the system from stalling [32].

In video-based synthesis, multiple frames or video streams are processed and combined to generate new visual data, such as reconstructed frames, interpolated motion, or enhanced video output. This type of processing often involves several computational stages, including decoding, transformation, and rendering. Multithreading plays a critical role here by enabling these tasks to run in parallel. For example, while one thread decodes incoming video frames and stores them in a buffer, another thread can simultaneously process previously decoded frames to perform synthesis operations. By breaking down the workflow into concurrent threads, the system can maintain high throughput and responsiveness [32].

Another key use case for multithreading in video processing is motion vector computation, a fundamental step in video compression. Techniques such as optical flow computation, which estimates the motion between pixels in sequential frames, are particularly computationally intensive. Since each framework in this process is independent of the others, these operations can be effectively parallelized across multiple cores [33].

2.12 Frontend

Modern web-based interfaces for drones are commonly designed using modular web technologies that allow for dynamic, real-time data display and interaction. A common architecture includes the use of structured HTML for layout, CSS frameworks like Bootstrap for responsive design, and client-side JavaScript libraries such as *Leaflet.js* for interactive geospatial visualization [34].

This architecture supports real-time telemetry visualization, responsive user control elements, and synchronized updates of drone state and geolocation. Such frontend designs align with modern principles of human-robot interaction, which emphasize low-latency responsiveness, usability, and system modularity. A key goal in human-robot interaction interfaces is to support real-time monitoring and control while minimizing cognitive load for the operator [35]. Additionally, modular frontend architectures allow integration with heterogeneous backends such as robotic controllers, simulation environments, or mission management platforms, thereby supporting flexible and extensible robotic systems [36].

2.13 CI/CD

Continuous Integration (CI), is a practice in which members of a team integrate and merge development work (*e.g.*, code) frequently, for example, multiple times per day. CI enables software companies to have shorter and more frequent release cycles, improve software quality, and increase their teams' productivity. This practice includes automated software building and testing [37].

Continuous Deployment (CD); the goal of this practice is to automatically and steadily deploy every change into the production environment [37].

2.14 Hardware

The hardware used in this project consists of a DJI Mavic 2 Enterprise, which is shown in Fig. 2.9, and the Holybro x500 v2 Development Kit. The DJI drone communicates with the remote controller over a USB connection and is compatible with the DJI SDK described in Section 2.3 whereas the Holybro drone, sometimes referred to as an open-source drone, communicates over 915 MHz radio frequency using MAVLink packets, see Section 2.5. The camera FOV for the DJI drone is 16:9 which is used for determining drone area coverage.

2.15 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on *Graphical Processing Units* (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs [38].

Figure 2.9: Figure shows DJI Mavic 2 drone, ZLEA CC BY-SA 4.0, via Wikimedia Commons.

3

Methodology

This project will build upon previous work on identification models, which can identify various objects in the test area, including trucks, bicycles, and pedestrians [3]. The control system for the drones will be implemented using MAVLink [39], described in Section 2.5 and using the DJI SDK described in Section 2.3. The drone controller developed in previous work, which utilizes an Android app will be extended. A frontend interface will serve as the main control center for the drones.

3.1 Project management

For this project, the team adopted the Scrum framework based on the Agile methodology. The Scrum framework was well suited for this project due to the ability to adapt to changing requirements and unforeseen change [40]. Other methodologies, such as the Waterfall methodology, were not suitable since the project requires continuous testing and being able to adapt to feedback during development.

The work in this framework was split into Epics, also referred to as Milestones. Milestones were then further broken down into a number of smaller Stories, or sometimes called Issues [41].

All the Stories were initially placed in the To-do section of the team's backlog. As a student began working on a Story, it was moved to the In progress section. Upon completion, the Story was moved to the Done section. This workflow is visually represented on the Scrum board, as illustrated in Fig. 3.1.

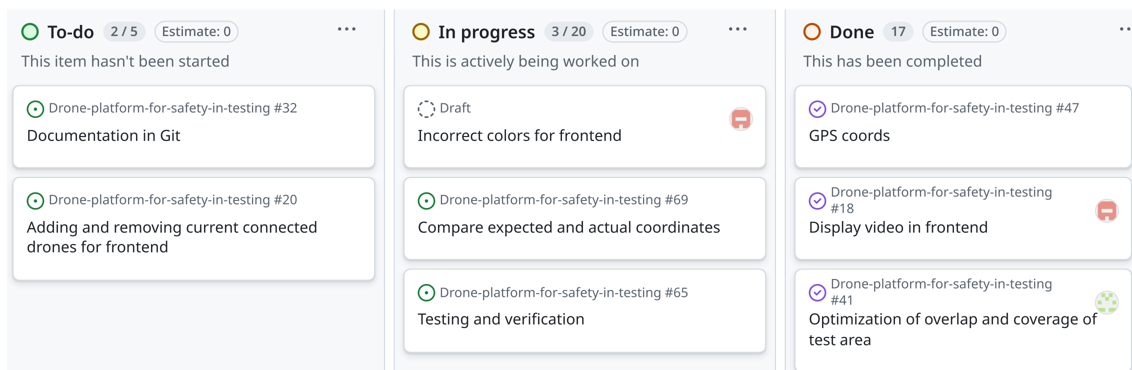

Figure 3.1: Scrum board hosted on GitHub.

3.2 Testing & Verification

For the test of code during development, multiple techniques were used, for example, the theory from Section 2.13, and a CI pipeline was configured within GitHub Actions. This pipeline is automatically triggered whenever a developer initiates a pull request to merge changes into the main branch. The pipeline executes a number of unit tests. If any test fails, the merge is blocked, preventing the introduction of faulty code into the main branch.

Upon successful completion of the unit tests. A second developer examined the proposed changes for code quality, style adherence, potential bugs, and overall maintainability. The pull request required approval from the reviewer before it could be merged. If the code review identifies issues or suggests improvements, the original developer addresses the feedback and updates the pull request.

Once code changes are successfully merged into the main branch, a CD pipeline is automatically initiated. The pipeline built and pushed a new Docker image to the container registry to incorporate the latest code changes. The updated Docker image was then deployed, either through a manual process or using a tool, for example, Argo CD Image Updater to automatically detect the new image and initiate deployment [42].

3.3 Version control

The project used Git for version control. The project's Git repository was hosted on GitHub, leveraging its features for code management, collaboration, and CI/CD. The repository is publicly available at `GitHub` [43]. For new features or bug fixes, developers created dedicated feature branches from the main branch. When development was completed, a pull request was initiated to merge the feature branch into main. This process follows the steps outlined in Section 3.2.

3.4 Software

The complete system, illustrated in Fig. 3.2, comprises various smaller subsystems, incorporating pre-existing components such as ATOS (discussed in Section 2.7) and the YOLO software (explored in Section 2.9). The system also builds upon previous work, as mentioned in Section 1.2. The development tasks were divided among the students illustarted in Fig. 3.2. The Chalmers team (green boxes) focused on the frontend interface, combining video streams (stitching), video transmission, and object detection, while the Penn State team (blue boxes) worked on making the system compatible with both the DJI SDK and the MAVLink platform. For the sake of clarity, detailed discussions about the *Toggle switch* and MAVLink platform are beyond the scope of this thesis. Interested readers are referred to the work by Penn State for further information [44].

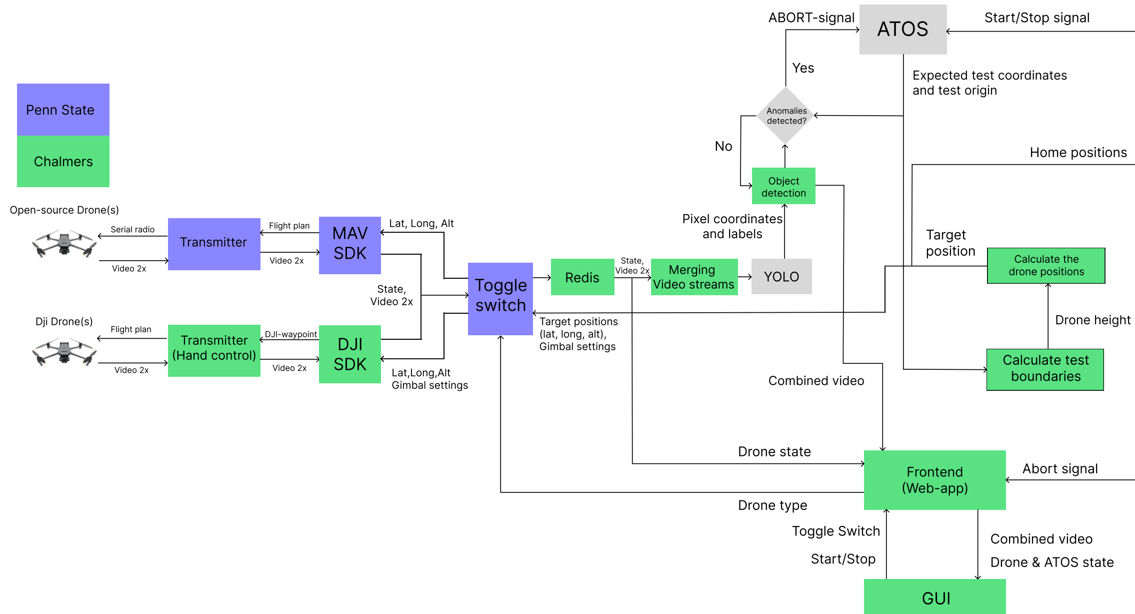

Figure 3.2: UML diagram for the complete system, the gray boxes refer to external existing software provided by [6, 30].

3.4.1 Docker

The system architecture leveraged Docker to enhance modularity and deployment flexibility, see Section 2.1. Three separate Docker images were defined, corresponding to the primary functional units: the backend, the image stitching, and the frontend interface. This containerization strategy significantly simplified deployment onto any Linux AMD64 machine by abstracting away underlying system configurations. The deployment workflow supported obtaining pre-built images from a container registry or building the images directly from source using the provided Dockerfiles. All containers communicated with each other using an internal Docker network.

The backend container utilized the ATOS image, detailed in Section 2.7, as its base. This ATOS image, in turn, was built upon a ROS 2 base image, described in Section 2.6. A Dockerfile orchestrated the build process for the backend image:

1. It began by specifying the ROS 2-based ATOS image as its base.
2. Necessary Python libraries were then installed.
3. Subsequently, the project's source files were copied into the image.
4. The ROS 2 packages within the project were then compiled.
5. Finally, the ROS 2 workspace setup was sourced in the container's `.bashrc`[1] file to ensure the environment was correctly configured on startup.

To manage this multi-container application, a Docker Compose file was created. This file serves as a blueprint, defining how all Docker services, networks, and volumes are configured and orchestrated. Specifically, it:

- Defined the start command for the backend container, initiating the main loop within its ROS 2 package.

[1] `.bashrc` files are configuration files that define the bash shell environment.

- Ensured all services communicated over a shared internal network named `atos-net`.
- Specified the necessary port mappings for external access:
  - Port **14500/tcp**: Mapped for communication between the Android app.
  - Port **8000/tcp**: Mapped for interaction with the frontend interface service.
  - Port **3478/udp**: Exposed for the STUN server functionality.
  - Port **8765/udp**: Exposed for WebRTC signaling.
- Facilitated the configuration of essential environment variables for the containers:
  - `ENV_LATITUDE`: Sets a custom latitude for the ATOS origin.
  - `ENV_LONGITUDE`: Sets a custom longitude for the ATOS origin.
  - `DEBUG_MODE` (True/False): Enables a debug mode, utilizing a predefined ATOS origin and outputting additional debug messages.
  - `N_DRONES`: Specifies the number of drones to be used.
  - `OVERLAP` (0.0-1.0): Defines the desired camera image overlap percentage between adjacent drones.

3.4.2 Drone Positioning

The drone positioning is controlled through a Python script in the backend. It is based on the drone positioning function used in the previous work [3]. In this project, we extended this function for multiple drones instead of one. The goal of the script is to evaluate the ATOS trajectories and produce the optimal coordinate placements for the drones. The efficiency of the placements is evaluated based on effective use of the drones FOV. It is required that the drones cover the entire test area, and that the placements maximize effective use of the drones FOV.

The drone-localization function extracts the ATOS-trajectories into an array of coordinates **C** with respect to the internal ATOS Cartesian plane. A nested function is used to compute the *Oriented Bounding Box* (OBB) of the convex hull of the points array, see Fig. 3.3.

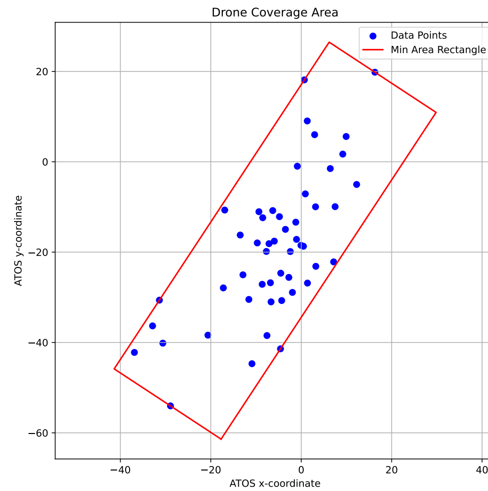

Figure 3.3: Oriented bounding box of sample trajectory.

The OBB calculating function that is applied is the one described in [45], Listing 1. The pseudocode was translated from C++ into Python using ChatGPT. The results were verified through testing the function with an arbitrary input trajectory dictionary and plotting the result using `Matplotlib` in Python.

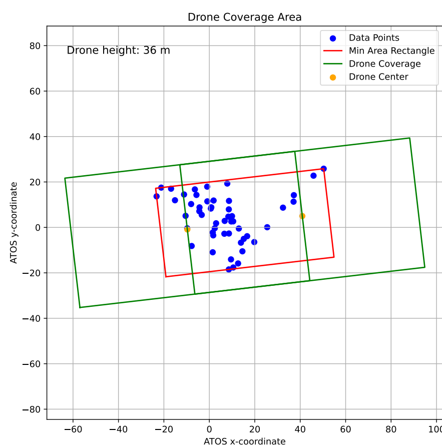

Figure 3.4: Drone coverage area 1.

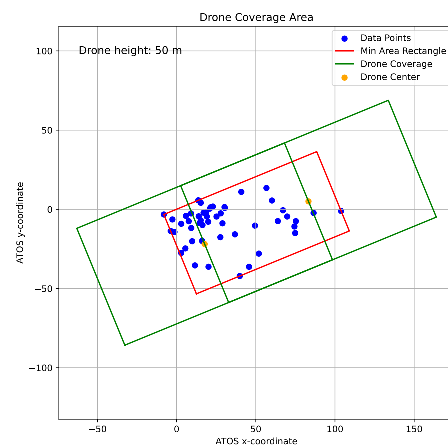

Figure 3.5: Drone coverage area 2.

The result was assessed and deemed to be satisfactory because the drone squares cover the entire area while also respecting the overlap parameter.

The attributes of the OBB are the axis $\mathbf{A}_{\text{rect}} \in \mathbb{R}^{2\times2}$, center $\mathbf{c}_{\text{center}} \in \mathbb{R}^{2\times1}$, the extent (half-lengths of the sides of the triangle, see Fig. 3.6) $\mathbf{e}_{\text{rect}} \in \mathbb{R}^{2\times1}$ and the area $a_{\text{rect}} \in \mathbb{R}$.

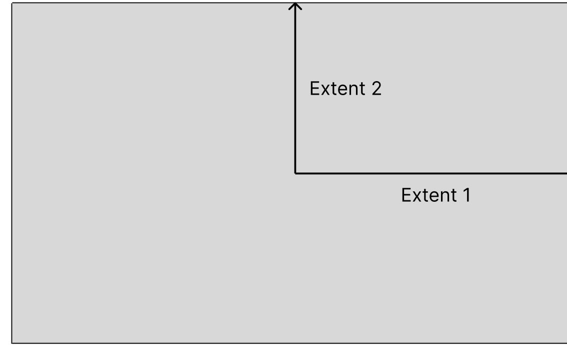

Figure 3.6: Extent of the OBB.

The axis stores the unit vectors in a matrix for the coordinate system on which the OBB is defined. The center variable $\mathbf{c}_{\text{center}}$ stores the center position of the rectangle, and the extent refers to the half-lengths of the rectangle sides in the directions of the unit vectors. An important assumption that is made is that the points are not collinear. These cases are handled by a function that returns a `bool` indicating if the points are collinear, and if they are, then the rectangle is manually constructed to fit the coordinates using the following calculation. First, the center is taken as the mean, and the list is sorted,

$$\mathbf{c}_{\text{center}} = \text{mean}(\mathbf{C}, \text{axis=0}) \text{ and} \tag{3.1}$$

$$\mathbf{C}_{\text{sorted}} = \text{sorted}(\mathbf{C}, \text{key} = p_0) \tag{3.2}$$

where p_0 refers to the first value in the coordinate pair and axis=0 refers to sorting the array on the first value of the coordinate pair. Then, the last point of the array is stored in a variable

$$\mathbf{c}_{\text{end}} = \mathbf{C}_{\text{n}}. \tag{3.3}$$

A vector $\mathbf{d}$ is constructed between the center and the last point

$$\mathbf{d} = \mathbf{c}_{\text{end}} - \mathbf{c}_{\text{center}}. \tag{3.4}$$

The vector $\mathbf{d}$ is normalized and stored in $\mathbf{U}_0$ as a unit vector. The perpendicular unit vector is then stored in $\mathbf{U}_1$

$$\mathbf{U}_0 = \text{norm}(\mathbf{d}), \tag{3.5}$$

$$\mathbf{U}_1 = \text{perp}(\mathbf{U}_0), \tag{3.6}$$

where $\text{perp}(\boldsymbol{\alpha})$ is a function that obtains the perpendicular vector of $\boldsymbol{\alpha}$, and $\text{norm}(\boldsymbol{\alpha})$ is a function that obtains the normalized vector of $\boldsymbol{\alpha}$. The max extent of the rectangle is defined as the absolute value of the vector $\mathbf{d}$, and the short extent is set to half of e_{max}. These values are stored in an array

$$e_{\text{max}} = |\mathbf{d}|, \tag{3.7}$$

$$e_{\text{rect}} = \begin{bmatrix} e_{\text{max}} \\ \frac{e_{\text{max}}}{2} \end{bmatrix}. \tag{3.8}$$

The axis $\mathbf{A}_{\text{rect}}$ is stored in an array, and the area for the rectangle a_{rect} is calculated

$$\mathbf{A}_{\text{rect}} = \begin{bmatrix} \mathbf{U}_0 \\ \mathbf{U}_1 \end{bmatrix}, \tag{3.9}$$

$$a_{\text{rect}} = 4\, e_{\text{rect},0}\, e_{\text{rect},1}. \tag{3.10}$$

When the OBB has been calculated, the drone boxes and positions $\mathbf{c}_{\text{drones}}$ are extracted. The split axis s_{axis} along which the drones are placed is set as the unit vector parallel to the side of the longest extent, and the axis for which the drone angles are calculated is defined as the unit vector perpendicular to the split axis. The measurements are visualized in Fig. 3.7.

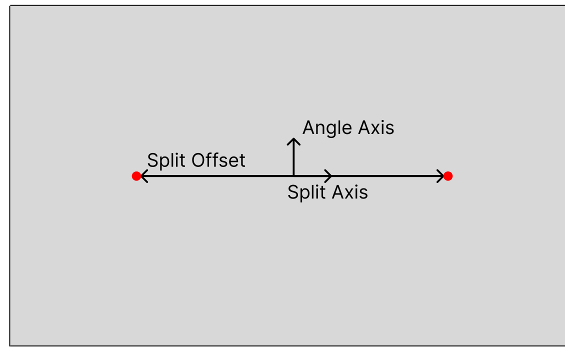

Figure 3.7: Visualization of axes and offset.

The calculation for the distance between the drones s_{offset} is as follows, for a certain overlap ratio ϕ. w and h_{rect} refers to the width and height of the rectangle respectively. The drone rectangles maintain a 16:9 aspect ratio, which is due to the aspect ratio of the drone's camera described in Section 2.14. The different parameters are calculated as

$$r = \frac{16}{9}, \tag{3.11}$$

$$h_{\text{rect}} = \sqrt{\frac{a_{\text{rect}}}{n_{\text{drones}}\, r}}, \tag{3.12}$$

$$w = h_{\text{rect}}\, r, \tag{3.13}$$

$$s_{\text{offset}} = w\,(1 + (1 - 2\,\phi)). \tag{3.14}$$

The drone centers are calculated using a for-loop in list comprehension shown in

Algorithm 1 Compute Drone Positions in a Line Formation

Require: Number of drones n_{drones}, center position $\mathbf{c}_{\text{rect}}$, offset scalar s_{offset}, axis vector s_{axis}

1: **for** $i = 0$ to $n_{\text{drones}} - 1$ **do**
2: $\quad \mathbf{c}_{\text{drones},i} \leftarrow \mathbf{c}_{\text{rect}} + \left(i - \frac{n_{\text{drones}}-1}{2}\right) s_{\text{offset}}\, s_{\text{axis}}$
3: **end for**

The height is then retrieved from a height calculating function from Section 3.2.3 in [3]. For this function, the drone FOV area $a_{\text{drone}} = w\, h_{\text{rect}}$ is used. If the height of the drones is below 30 meters, it is locked at 30 meters, as the object detection model is trained at this area. In this case, the calculation is done in reverse, starting by fixing the height and then calculating the optimal drone positions in order to obey the overlap.

The drone coordinates are then converted to latitude and longitude relative to the test origin. The test origin is a coordinate in latitude and longitude that represents the origo of the ATOS coordinate system. The purpose of this function is to position the drones in such a way that their FOV collectively cover the entire OBB calculated from the ATOS trajectory points, while always maintaining the required overlap between the FOVs, regardless of the drones altitude.

3.4.3 Real-time video streaming

Using the `DJICodecManager` class of DJI's Mobile SDK V4, as described in Section 2.3.2, video frames could be extracted from the DJI drone in real-time in YUV420 pixel format. As stated in Section 2.8.1, WebRTC supports H.264 encoded video, which is compatible with the YUV420 pixel format. Using this format, a WebRTC-based system for transmitting the drone's video stream from the application to the backend could be implemented as illustrated in Fig. 3.8.

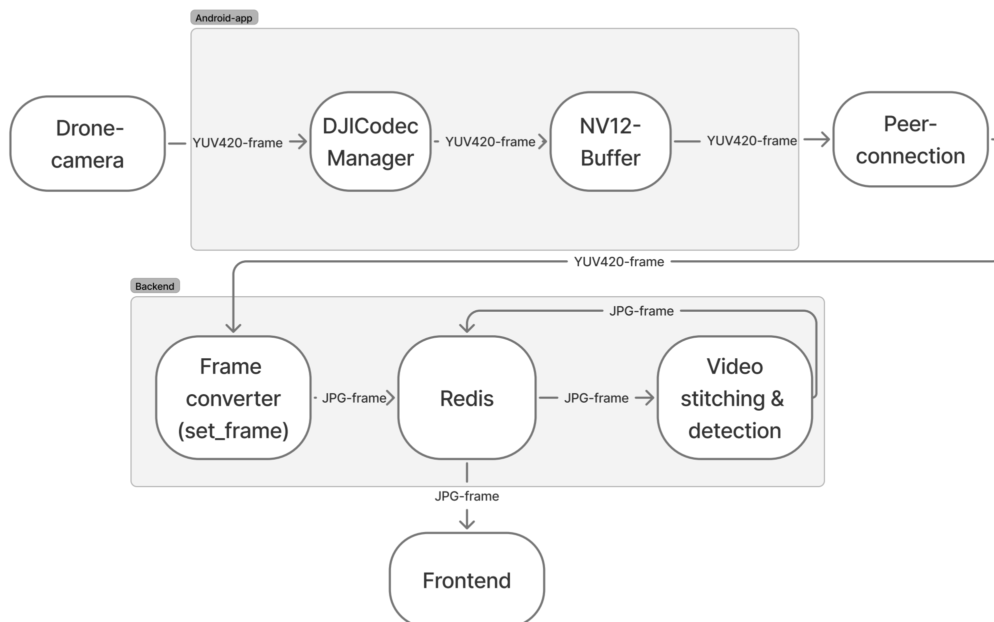

Figure 3.8: System architecture for the WebRTC-based video streaming system.

Since the Android app can connect to the backend via a WebSocket, as described in Section 2.2, this same WebSocket connection can also function as the signaling

server for the WebRTC handshake, explained in Section 2.2. Because all devices are located on the same local network, peer-to-peer connections should be achievable without the need for either a STUN or a TURN server, explained in Section 2.8, enabling direct communication between peers. When running in Docker described in Section 3.4.1, it is necessary to use port forwarding for this to work.

However, since the server-side client (backend) was implemented in Python using the `aiortc` library, which requires a STUN server address during ICE configuration, an ICE server had to be specified. For this purpose, Google's public STUN server was used. When the system is offline, this server remains unused, as peers can find each other directly when located on the same local network.

Once video frames are received by the backend, they are converted into JPG format and stored in Redis with a key indicating their origin drone, as described in Section 2.1. This makes the frames accessible throughout the backend and frontend. The formatting ensures compatibility with the video stitching and the frontend interface.

3.4.4 Extended video surveillance

Using the theory in Section 2.10, the live streams from both drones can be merged into one panoramic view. The two images $I_{\text{left}}(x)$ and $I_{\text{right}}(x)$ are those combined by the blend for each pixel x. The formula for the blending is obtained in Eq. (3.15),

$$I_{\text{blend}}(x) = (1 - \alpha(x))\, I_{\text{left}}(x) + \alpha(x) I_{\text{right}}(x), \tag{3.15}$$

where $I_{\text{blend}}(x)$ represents the blended pixel at position x, $I_{\text{left}}(x)$ and $I_{\text{right}}(x)$ are the pixel values in the left and right images at position x, respectively, while $\alpha(x)$ is the weight that controls how much each image contributes to the blended image.

Detecting objects from merged video streams requires both proper synchronization between drones and effective processing and analysis. Using the theory from Section 2.9 on object detection principles, it is evident that real-time detection requires efficient computational models. For real-time detection, models such as YOLO are essential. YOLO performs direct object detection, minimizing the computational load compared to traditional network-based approaches [46]. By using YOLO, both stationary and moving objects can be detected and tracked across each camera view.

To enable tracking of objects detected by YOLO, the coordinates of the objects need to be calculated. This method was originally developed in previous work described in Section 1.2 and focuses on saving the center pixel from each bounding box. Using the drone's altitude, FOV, and GPS position, an estimated position for all objects is calculated [3].

To allow positioning of multiple drones, a weighted GPS position needs to be calculated to locate the object between the two cameras in the merged image. By calculating a weighted alpha value α, the position of the object is interpolated between the GPS coordinates of the left and right cameras, depending on the location

of the object in the image. If the object is closer to the left camera, more weight is given to the left camera, and if it is closer to the right camera, more weight is given to the right camera. The methodology for transforming pixel coordinates into GPS coordinates was originally developed in previous work described in Section 1.2. As described in Eq. (3.16), this method enables the extraction of weighted GPS coordinates, including both latitude and longitude:

$$\text{GPS} = (1 - \alpha)\ \text{GPS}_{\text{left}} + \alpha\ \text{GPS}_{\text{right}}, \tag{3.16}$$

where α is the pixel's x-coordinate divided by the total width of the stitched image.

To enable simultaneous image processing from multiple video streams, an implementation of parallel computing through multi-threading is required [47]. The theory from Section 2.11 is using this architecture, where each video stream is assigned a separate execution thread, where frames are continuously fetched and placed in a queue structure. The pairs are then retrieved from these queues by the main thread, which then performs the merging and executes the object detection. The use of these queues and thread events ensures that the data flow remains stable. This ensures that data is processed in real time.

To accelerate image processing and offload computations from the CPU, the GPU was utilized. Using theory from Section 2.15, a special Docker Compose file was created to take advantage of the CUDA acceleration. This reduced the CPU load from around 95% to 20%. The subsystem for combining the video streams could also be run on a separate machines which was achieved via setting the Redis URL from the internal Redis hostname to the separate machine IP.

3.4.5 Android app

The communication between the two hosts was changed from string format, as it was implemented in previous work and described in Section 2.4, to JSON serialization. The name `msg_type` denotes the type of the incoming message and which function should be executed. The implementation in the backend is highlighted in Appendix A.1. All possible `msg_type`'s are listed in Table 3.1.

Table 3.1: Description of different. `msg_type`

msg_type	Purpose
Coordinate_request	Exchange drone waypoint coordinates
Position	Send telemetry and position to backend
Debug	Message type used for debugging prints JSON payload
candidate	Candidate for WebRTC
answer	Answer for WebRTC

The mobile phone receives the telemetry and position using the DJI SDK described in Section 2.3 and sends a message over the socket to the backend using the `msg_type` set to `Position`, see Fig. 3.2. The backend stores the incoming data in Redis. The

data is published in a Redis Pub/Sub channel where each drone ID is assigned its unique key.

3.4.6 Frontend

The frontend interface serves as the user-facing component of the system, allowing operators to control and monitor drones in real time. It builds upon the architecture described in Section 2.12. The frontend pipeline and it's integeration with the rest of the system is also visualized in the Fig. 3.2. The communication and data transfer between frontend and rest of the system is also visualized in that figure.
The interface includes a responsive layout comprising a video display section, telemetry dashboard, control panel, and a live map as shown in Fig. 3.9.

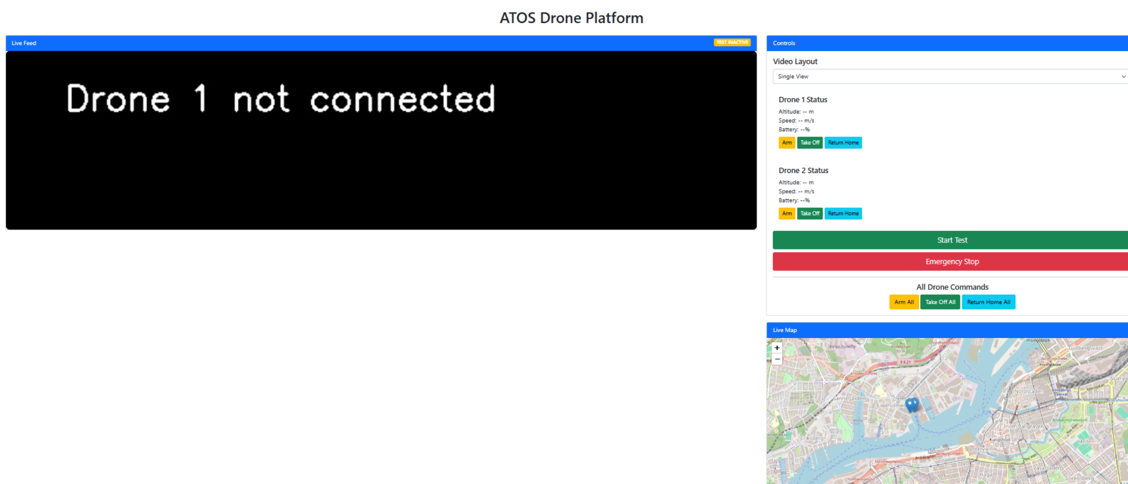

Figure 3.9: Frontend user interface with with video display, telemetry dashboard, control panel and live map.

Positioning and telemetry data for each drone is sent continuously from the backend and displayed in the browser. This data is obtained through a persistent WebSocket connection from the backend, which relays the live Redis-published telemetry to the frontend interface, where it is parsed and used to update each drone's position, altitude, speed, and battery status as described in Section 2.2.

To visualize the drones on a map, `Leaflet.js` markers are updated with the latest GPS coordinates of each drone, from Redis. In this way, operators receive an accurate and up-to-date spatial representation of drone positions relative to the test environment as shown in Fig. 3.10.

For each live video feed, the frames are retrieved from Redis, and served as MJPEG streams through *Hypertext Transfer Protocol* (HTTP) endpoints, and are displayed in the frontend using HTML `img` tags. Users can select different layout configurations (e.g., single stream, side-by-side, or stitched panoramic view) via a dropdown menu as shown in the Fig. 3.11. The stitched video feed is generated in the backend using the methodology explained in Section 3.4.4, and delivered to the frontend via a dedicated endpoint.

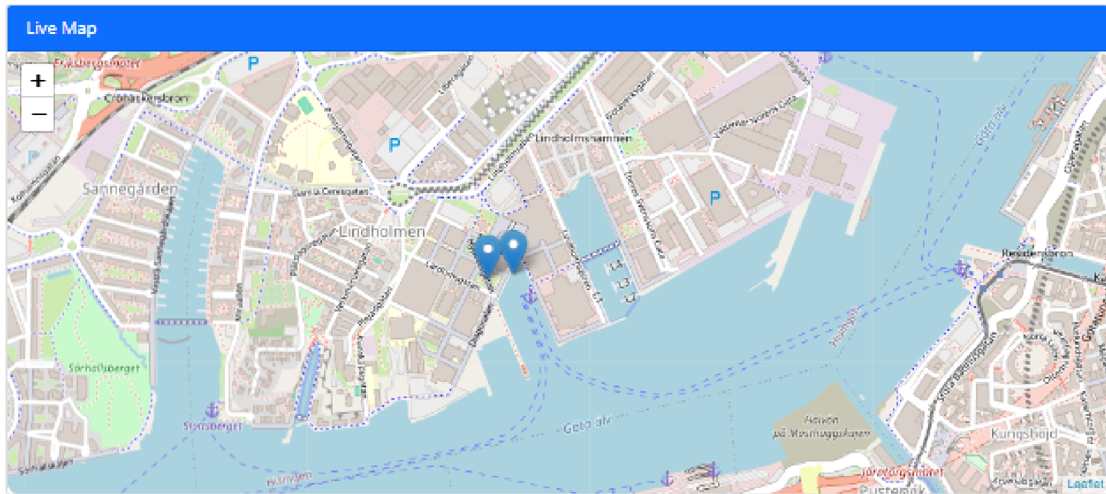

Figure 3.10: Interactive map displayed in the frontend user interface, visualizing live positioning of the drones.

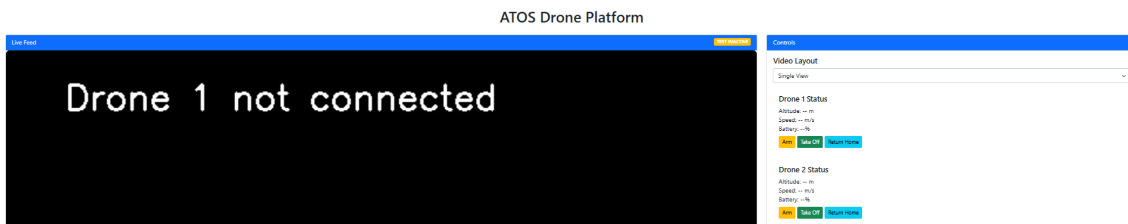

Figure 3.11: Video stream with single view layout configuration in the frontend user interface.

3.4.6.1 Drone control using the frontend

The control of the drones builds on the theory described in Section 2.3.1 is managed by the user from the frontend application. The interface includes command buttons for Arm, Take-Off, and Return to Home as shown in Fig. 3.12.
Each button dispatches a WebSocket message to the backend, which translates these into actionable commands for the drones and forwards it the mobile phone.

The Arm function sends a message from the frontend interface to the backend via WebSocket. The message is received by the Android app's client handler, which decodes it and triggers the arming function in the flight manager. This function performs initial checks, such as verifying the battery state (minimum 20 % required) and GPS lock. It then clears any leftover waypoints and defines the new mission. A take-off waypoint is generated 10 meters above the drone to prevent collision, and subsequent mission waypoints are appended in order. The flight characteristics are configured, and the waypoint mission is uploaded.

The Take-Off command, when triggered from the frontend, initiates the mission start by calling a function in the DJI SDK. Similarly, the Return to Home command sends a corresponding message to return the drone to its launch location. These controls are also available as global commands that apply to all connected drones simultaneously as shown in Fig. 3.12.

Additionally, operators can start or abort test sessions via the `Start Test` and `Emergency Stop` buttons as shown in Fig. 3.12. These trigger WebSocket messages directed at the ATOS controller, which maintains test state. The frontend interface visual feedback (*e.g.*, test active badge) is synchronized with the ATOS system response as it can be seen in Fig. 3.9 showing `TEST INACTIVE`.

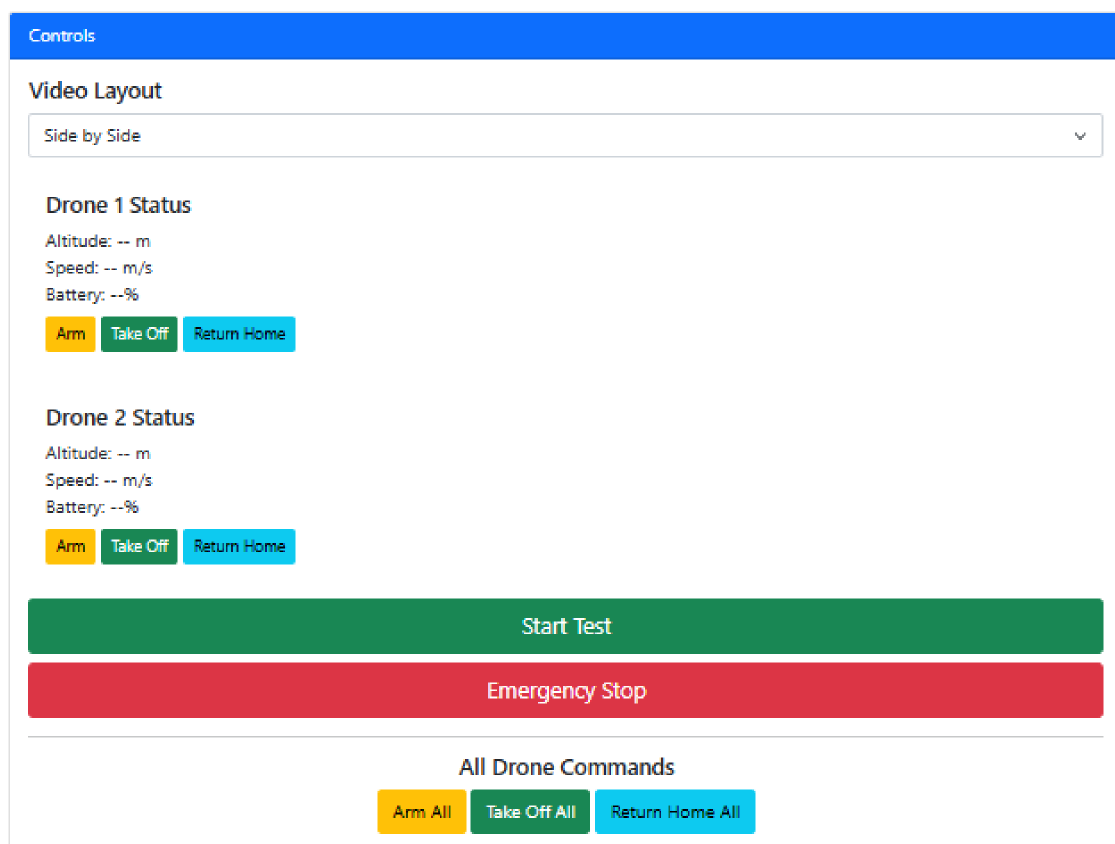

Figure 3.12: Different command buttons and video layout configuration in the frontend user interface.

4

Experimental Validation and Results

This chapter outlines the outcomes of the project and explains the methods used to test the system's functionality. The testing approach was aimed at assessing critical performance parameters and confirming that the developed components met the specified requirements. Both qualitative insights and quantitative data are provided to support the system's validation. The main results are:

- Positioning algorithm for two drones.
- Real time video streaming from drone to computer.
- Video merging of two drone-video streams.
- Web-based GUI with system monitoring and control capabilities.

These results are further discussed and validated bellow.

4.1 Verifying the Docker Setup

The experiments relied heavily on Docker containers. Before we could validate anything else, we first had to confirm that this containerized setup was working correctly. The containers and the communication between them is shown in Fig. 4.1.

4.1.1 Container Startup and Communication

The three Docker containers for the backend, image stitching module, and frontend interface all started up as expected. We used the `docker ps` command to check that they were running and that their ports were correctly mapped. The output from `docker ps` is shown in Fig. 4.2.

A key check was ensuring smooth communication. The containers needed to talk to each other, to internal services like ATOS and Redis, and also connect externally with the Android app and the frontend interface. This all happened over the dedicated `atos-net` Docker network. However, services intended for external clients were made accessible using Docker's port forwarding to expose the necessary ports. We confirmed this by observing successful data exchanges: for instance, the backend correctly queried ATOS, and the frontend interface received data from the backend containers, which originated from the Android app.

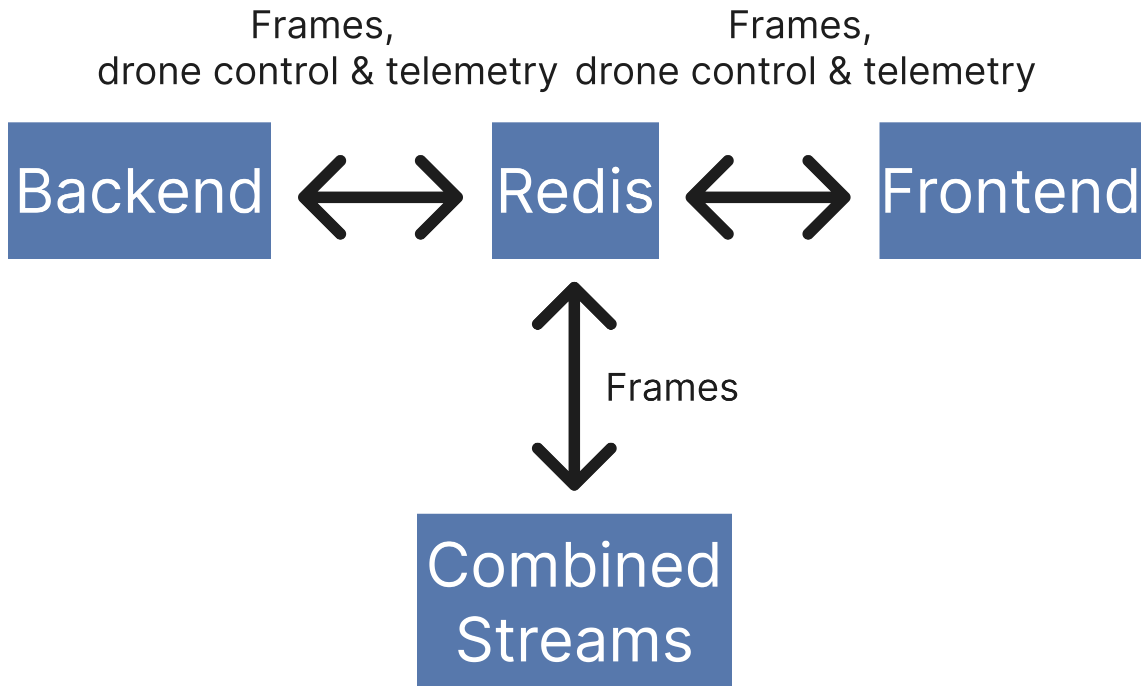

Figure 4.1: Communication between Docker containers.

4.1.2 Monitoring and Debugging with Logs

During experiments, container logs were essential for tracking real-time behavior and for any troubleshooting. We regularly used the command `docker logs <container_name>` to pull logs from each service. This helped us to:

- Confirm that each container initialized properly.
- Monitor key processes and check that data was flowing correctly between services.
- Quickly spot and fix any errors or unexpected issues that came up during specific test scenarios. For example, if the backend wasn't receiving a message, its logs were our first stop. An example of logs is shown in Fig. 4.3.

Sometimes, to dig deeper, the command `docker exec -it <container_name> /bin/bash` was used to open a shell inside a running container. This lets us inspect its internal state, files, and running processes directly, which was helpful for tricky debugging issues that logs alone couldn't solve.

4.1.3 Ensuring Consistent and Reproducible Experiments

One of Docker's biggest benefits for our experiments was environmental consistency. By packaging all software dependencies and configurations into Docker images, we made sure every experimental run used the exact same software environment. This cuts down on variations caused by different machine setups, making our results more reproducible and reliable. We controlled specific parameters for each run—like `ENV_LATITUDE`, `ENV_LONGITUDE`, or `N_DRONES`—through environment variables in the Docker Compose file, allowing us to systematically vary test conditions.

```
[inspirer@spectrex360 ~]$ docker ps --format json
Image: communication_software-frontend
Command: "/docker-entrypoint...."
Names: frontend
State: running
Status: Up 4 seconds
Ports: 0.0.0.0:8001->80/tcp, [::]:8001->80/tcp

Image: communication_software-backend
Command: "/ros_entrypoint.sh ..."
Names: backend
State: running
Status: Up 4 seconds
Ports: 5000, 8000, 3478, 8765, 14500 (tcp/udp mapped)

Image: communication_software-image_stitching
Command: "python3 main.py"
Names: image_stitching
State: running
Status: Up 5 seconds

Image: astazero/atos_docker_env:latest
Command: "/ros_entrypoint.sh ..."
Names: atos
State: running
Status: Up 4 seconds
Ports: 80, 443, 3000, 3443, 8080-8082, 9090, 55555 (tcp mapped)

Image: astazero/iso_object_demo:latest
Command: "/app/build/ISO_obje..."
Names: isoObject
State: running
Status: Up 5 seconds

Image: public.ecr.aws/docker/library/redis:latest
Command: "docker-entrypoint.s..."
Names: astazero-redis
State: running
Status: Up 5 seconds
Ports: 0.0.0.0:6379->6379/tcp, [::]:6379->6379/tcp
```

Figure 4.2: Status of running Docker containers. Some information for example, ID has been removed.

```
[inspirer@spectrex360 ~]$ docker compose logs backend
backend  | Client connected.
backend  | [DroneStream] Created RTCPeerConnection:
<aiortc.rtcpeerconnection.RTCPeerConnection object at 0x71c217fa8a60>
backend  | [DroneStream] WebRTC offer created: v=0
backend  | Assigned coordinate ('57.772956', '12.769956', '42',
'-164') to client 125078439888544
```

Figure 4.3: Example of logs retrieved from backend container. Some logs have been redacted.

4.1.4 Continuous Integration and Delivery

Our CI/CD pipeline, which automatically built and pushed Docker images for the backend and frontend interface to the GitHub Container Registry after every push to the `main` branch, was also key element for the correct deployment of our solution. Fig. 4.4 illustrates the build steps for the backend within this pipeline. This agile approach meant our validation always used the latest stable software version, keeping our tests aligned with ongoing development.

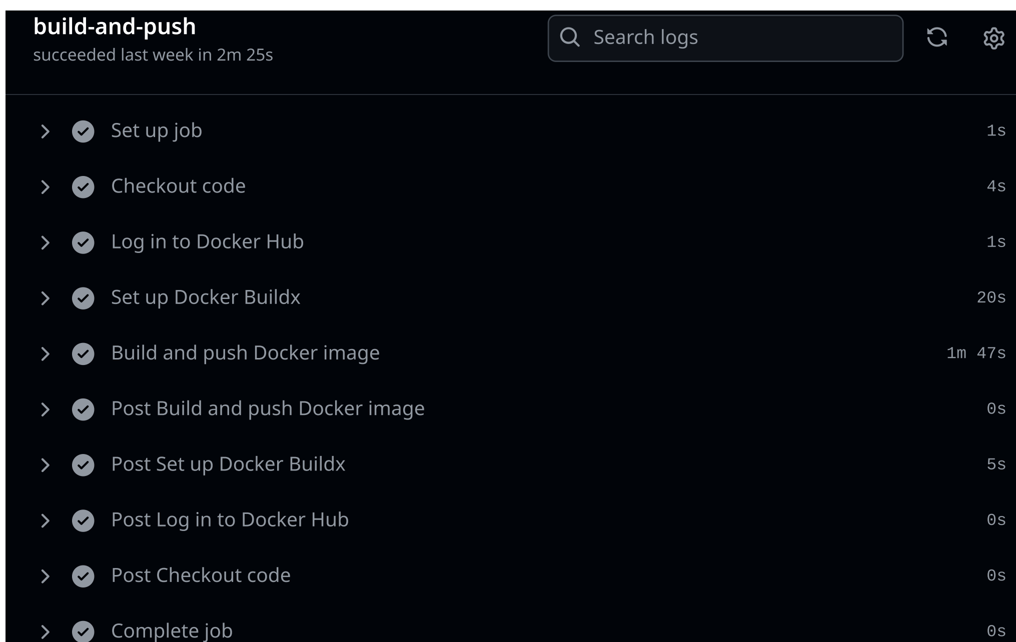

Figure 4.4: Pipeline for building the Docker image for the backend.

4.2 Drone Control

The drone control program, together with the drone positioning script, allows the drones to be controlled from the frontend interface and sent to their optimal positions. The drone positioning script maintains the desired amount of overlap at all heights below 120 meters, where the Swedish legal limit is according to [48]. In this

case, an exception in the code is raised. In the Section 3.4.2, the drone positioning system is presented.

The system covers the entire area in all cases with height below 120 meters, and respects the overlap that is used as argument in the function. In some edge cases bugs are present in the positioning system where it selects the wrong axis to plot the FOV-rectangles against, but this is deemed acceptable as in all tests the system has shown to work and still respects the overlap that is input to the system.

A test file was created to observe the coverage in the case of one drone and two drones. Ten simulated tests were performed using randomized ATOS trajectories. The drone positioning functions from this project and last year's project [3] were compared. The following graph was extracted. It can be observed that the height for the tests with two drones is lower in all cases. Thus, it can achieve sufficient coverage at a significantly lower height. This is valuable as the object detection model used currently is trained at a low altitude of 30 meters, and works best at that height.

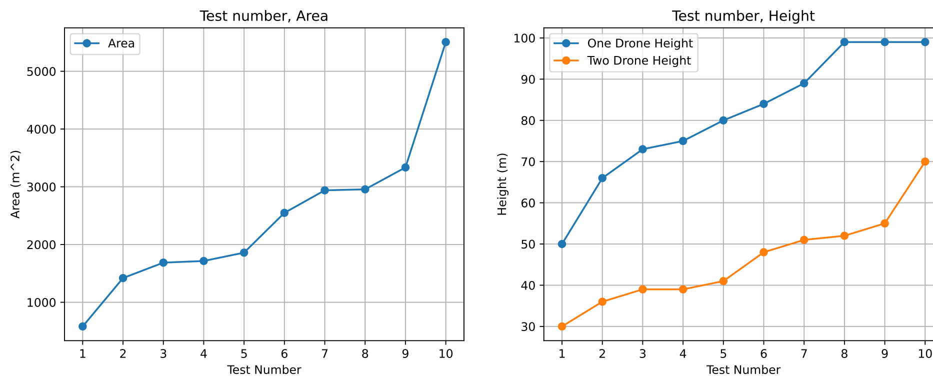

Figure 4.5: Covered area *versus* height.

The `FlightController` is effective for sending the drones to their desired locations. A Google Maps link was integrated into the code to test the positioning accuracy of the software, and it was deemed to be sufficient for the functioning of the system.

4.3 Real-time Video Streaming Using DJI Drones

The integrated WebRTC system connects the Android app to the client, as described in Section 3.4.3, once the WebSocket connection is established. Upon a successful peer connection, the Android client immediately begins streaming video captured from the drone to the backend, where the frames are written into Redis. These video streams can then be accessed and viewed separately by the user through the frontend interface, as illustrated in Fig. 4.6 and Fig. 4.7. If the drones are not connected or if a connection failure occurs, the frontend will receive and display black placeholder frames with an error message, as shown in Fig. 4.8 and Fig. 4.9.

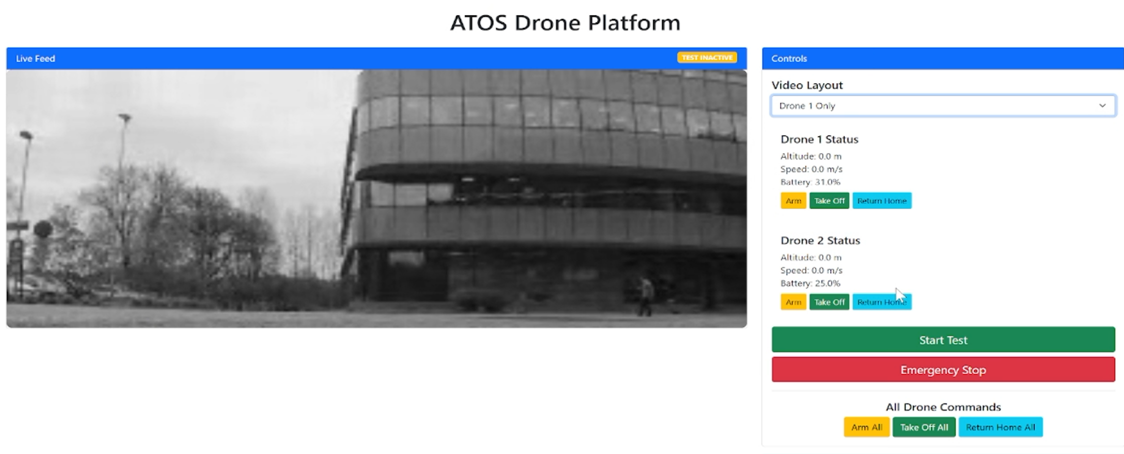

Figure 4.6: Camera view in the GUI for the first drone.

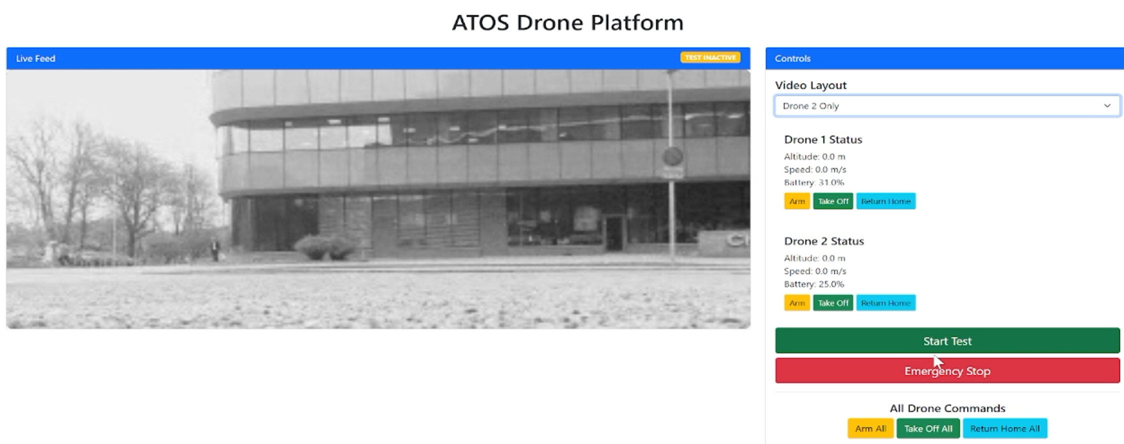

Figure 4.7: Camera view in the GUI for the second drone.

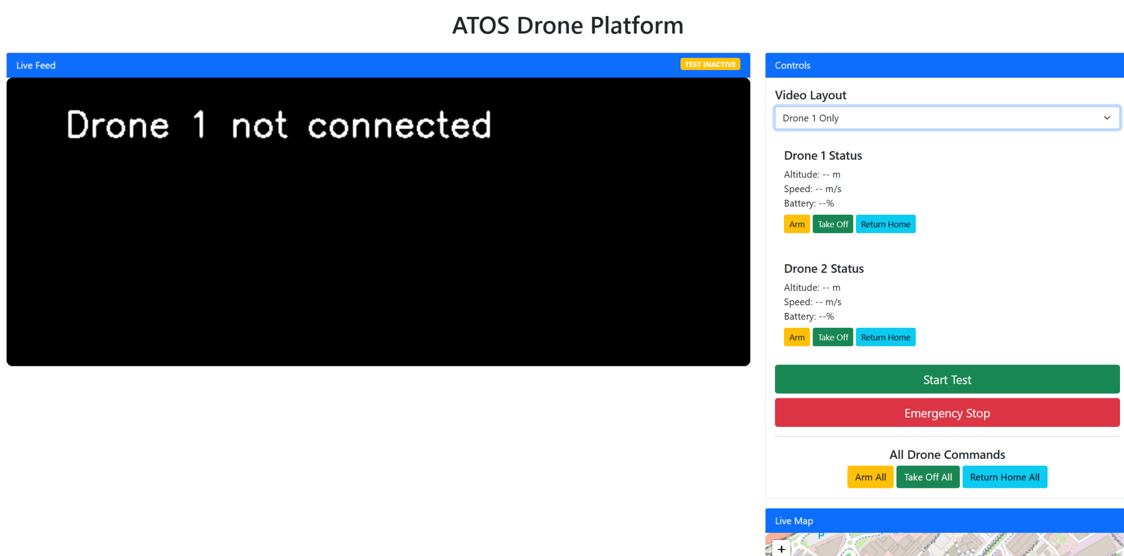

Figure 4.8: Filler frame with error message in the GUI for the first drone.

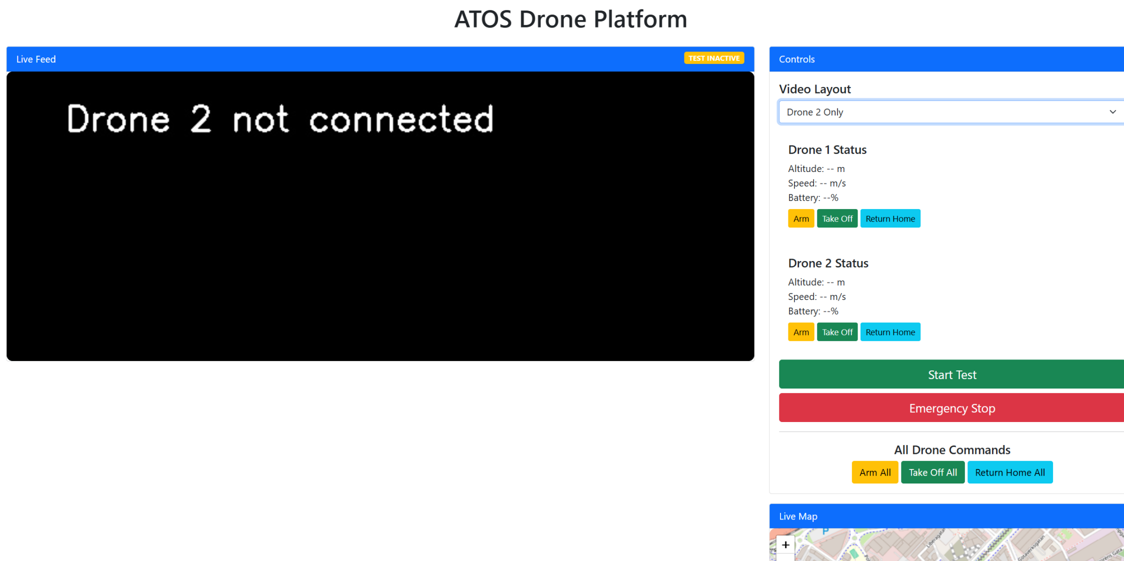

Figure 4.9: Filler frame with error message in the GUI for the second drone.

4.3.1 Integrated Functionality For Testing and Debugging

The peer connection state of each drone can be monitored through the terminal using the Docker command `docker compose logs comm_software` described in Section 3.4.1. This is possible because the WebRTC system is integrated into the backend with print statements to display information in the terminal. As discussed in Section 3.4.3, the server-side client is implemented in Python using the aiortc library. The library provides a connectionState attribute for each active peer connection. This attribute was used to implement drone stream state monitoring, described in Table 4.1.

An important observation from the system testing was that the Drone stream state will remain `Unknown` during the ICE candidate exchange before it becomes `Online` when the peer-to-peer connection is complete or `Error` if it fails.

To supervise the RTC handshake process, which is important during development and debugging, each step of the handshake is logged in the terminal. The logs include details about ICE candidates and SDP offers. These logs accessible via the same Docker command, provide visibility into the offer-answer negotiation and ICE candidate exchange between peers.

Table 4.1: Description of different drone stream states.

State:	**Meaning:**
Online	Peer connection was successful
Disconnecting	Client disconnected
Closed	Peer connection was removed successfully
Error	Peer-to-peer connection could not be established
Checking	Checking for peer connection
Unknown	State was not recognized by aiortc's `connectionState`

4.4 Extended Video Surveillance

In Fig. 4.10 the results of the extended video surveillance are displayed, where during system operation, a continuous video feed is displayed on the system frontend. Here, the live video streams are merged into one wide panoramic view. In the image, detected objects are marked with rectangles and text labels indicating each object's unique ID and its calculated GPS coordinates. The image is updated in real time and continuously reflects changes in the scene as detected objects move and are tracked, or new objects are detected. If the connection to a drone is lost, its part of the combined image is replaced with an information box indicating the connection status, as seen in Fig. 4.11.

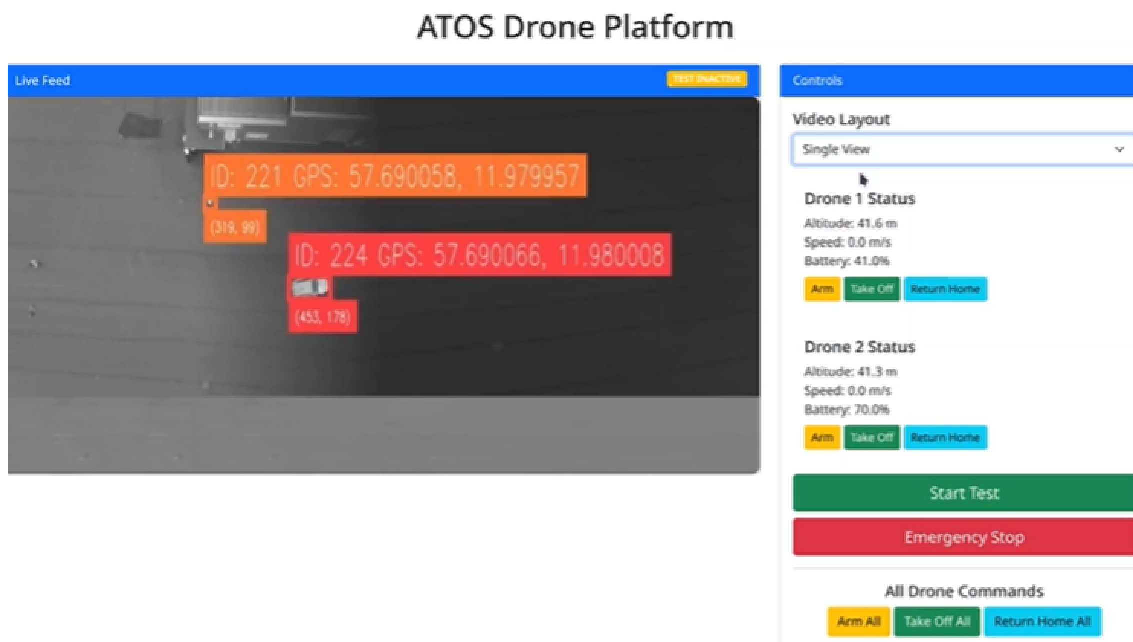

Figure 4.10: System testing at AstaZero.

The video streams are retrieved from a Redis database, where each individual drone uploads its current frame in JPEG format. Using asynchronous functions, these frames are loaded in parallel from each drone. These frames are then decoded into an OpenCV image in the program. The frames are then rescaled to a common size and stitched together into a merged panoramic image. Using alpha blending, the overlap zone from the two video streams is blended to create a seamless panorama with natural transition between the two frames.

On the extended panorama image, object detection is applied using a YOLO model developed in previous work [3]. For each detected object, a GPS coordinate is calculated based on its pixel position and the geographical position and altitude of the drones. In order to track a detected object between frames, weighted GPS coordinates are calculated.

Finally, the merged image is annotated, where each detected object is enclosed by a rectangle showing each individual ID and geographical GPS coordinates. Once the

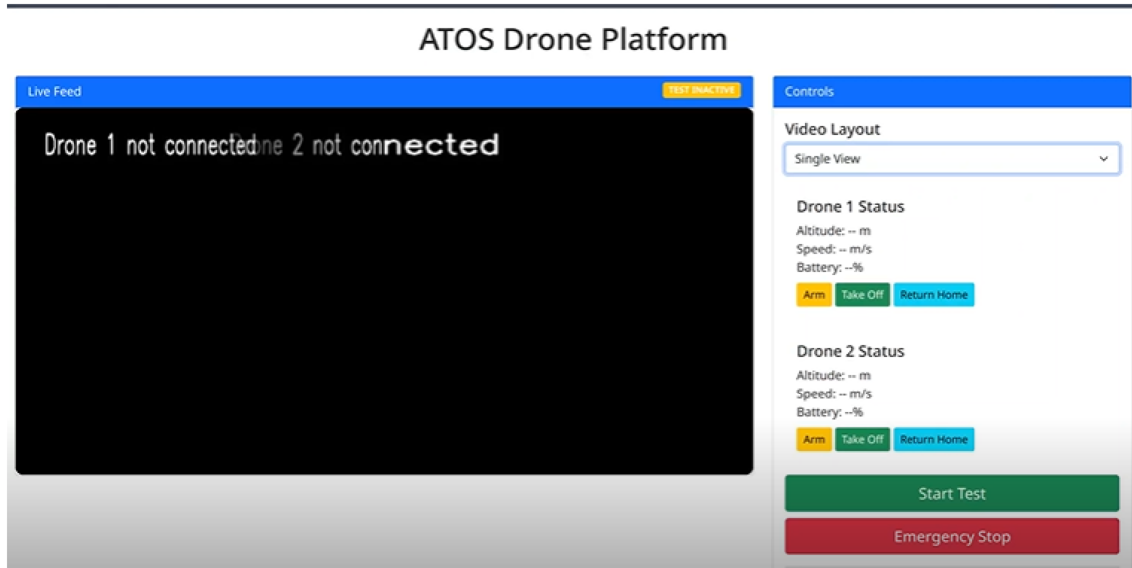

Figure 4.11: Drones not connected.

image is finalized, it is converted to JPEG format and sent to Redis to be accessed by the frontend interface.

4.4.1 Testing the Subsystem

To enable separate testing of the subsystem, locally stored video files were used. This approach was chosen because it enabled a more efficient development process by allowing the subsystem to be tested in parallel without being dependent on real-time streaming from drones. The use of local video files resulted in a significant increase in the number of possible tests, as the entire operational system did not need to be connected to obtain suitable footage. Development could thus proceed rapidly and iteratively, which is crucial in the early phases of system development. A fundamental prerequisite for this type of testing was that the methods implemented for the subsystem's purpose, such as video handling and image merging, were designed to function independently of the video source. This means that the functionality tested using local files could be assumed to be representative of the functionality during real-time streaming from drones.

The testing is considered valid because the same code base and processing methods were used regardless of whether the video originated from a local file or from a live stream. The critical functions to be validated concerned how the subsystem handles, processes, and merges video streams, and these operations are identical regardless of the source. Furthermore, local files allowed for systematic and reproducible testing: the same conditions could be reused to isolate and analyze potential errors, significantly improving the precision of fault analysis and systematic improvement.

Since the subsystem was based on previous work developed by Mohi *et al.* [3] , the initial phase of development focused on extending an existing object detection system that originally handled only a single video stream. The work then progressed to enable the handling of multiple streams simultaneously and to develop a method

for seamlessly merging these into a unified panoramic image. Validating the subsystem's ability to handle multiple streams and perform accurate merging could be reliably performed using local video files, as they provided controlled and repeatable test scenarios that captured the core aspects of the system's functionality.

After the tests with locally stored videos showed good results and a stable base system was established, the subsystem was adapted for integration with the other system components. This integration only required minor adjustments in the video handling logic.

Finally, the system's functionality was verified in a full drone system with real-time data, where further optimizations and calibrations of the model's key parameters could be performed.

5
Discussion

This chapter discusses the performance of the individual subsystems and the system as a whole, highlighting both strengths and limitations identified during development and testing. Reflections on the overall design process, lessons learned, and potential areas for improvement are also presented. In addition, the chapter addresses ethical aspects related to the project and proposes directions for future work.

5.1 Drone Control

For a surveillance area that is relatively square in shape the advantages of the system are not directly apparent, but for a test-scenario that follows a wider and narrower trajectory this system is capable of covering a substantially larger area than a system of only one drone. This is due to the 16:9 aspect ratio of the drones FOV.

5.2 Data Serialization

In previous work, the code utilized plain strings for communication between the Android app and the backend [3]. To send a coordinate over the socket the first ten digits represented the latitude, digits 11 to 20 denoted the longitude, digits 21 and 22 indicated the altitude, while the remaining digits specified the angle. However, one problem could arise if the altitude is single digit or three digits. Instead, a new serialization for data transfer was used which is JSON described in Section 2.4. One of the disadvantages of JSON is that it does not guarantee type safety and does not automatically enforce the schema.

It is also text based which is great for debugging, however, could lead to larger size, and serialization and deserialization could be slower compared to a binary format. An alternative is ProtoBuf developed by Google [49]. Messages transmitted with ProtoBuf are serialized into a binary wire which is compact, forward- and backward-compatible [50]. Since ProtoBuf uses a binary wire compared to JSON which is text-based ProtoBuf tends to be more performant. When transmitting large amount of data this could be a noticeable difference, for example, for video and telemetry data from the drones. Support for ProtoBuf in web browser are lower while JSON is more native. Therefore, it would make sense to use ProtoBuf between the mobile phone and backend and use JSON for the frontend webpage.

5.3 Endpoint vs Message Type

For communication between the backend and the mobile phone a message type was introduced as explained in Section 2.4 to distinguish between different types of methods. An alternative approach could be to use different endpoints for different messages *e.g.*, `/position` for drone position, `/flightmanager` for commands sent to the drones flight manager. This would eliminate the need for messages types since only drone position messages would be sent over the `/position` endpoint. This would make the code more structured, reducing the risks of bugs, for example, what if the mobile phone sends a message type that the backend does not recognize? The reason for current implementation is that to accommodate for the different endpoints the mobile phone would have to connect to multiple WebSockets. This would require changes to both the backend and the Android app.

5.4 Real Time Video Streaming In the Developed System and Its Limitations

The subsystem for real time video streaming, described in Section 2.8, is currently limited to DJI drones. This limitation arises because the Android app, described in Section 2.4, is currently specifically designed for DJI drones. To enable video streaming from other types of drones within the developed system, these drones or their control interfaces must support a WebRTC connection by implementing the logic to function as a WebRTC client, similar to the existing one in the current Android app.

Another limitation of the current video streaming setup is that the Android app supports video display either within the app or in the frontend, but not both simultaneously. The reason for keeping the function to display video in the app is for debugging purposes, such as verifying that the camera is properly connected before linking the drone to the backend system via the app.

5.5 Extended Video Surveillance

In the merged panoramic image, visual artifacts occasionally occur, particularly in connection with moving objects passing through the overlap zone between the two image frames. This phenomenon is well known in image stitching and is commonly referred to as “ghosting” [31], as it produces ghost-like duplicates of the moving objects. When such artifacts are not detected by the YOLO model, they primarily pose an aesthetic issue from the user’s perspective, without affecting the system’s core functionality or performance. However, problems may arise if these artifacts are falsely registered as valid objects by the model, which can lead to duplicate detections. This, in turn, may result in erroneous triggers, reactions, or mismatches in object recognition, where the system responds to objects that do not actually exist.

To minimize the occurrence of these artifacts and mitigate the effects of ghosting, it is crucial to ensure correct overlap between the image frames. In the current system, the overlap is manually defined in the code file. As long as these settings accurately reflect real-world conditions, a stable panoramic image without visible artifacts can be produced. However, this approach requires the ability to consistently reproduce test scenarios, enabling calibration of the model to specific conditions. This methodology was employed during the evaluation of the subsystem in Section 4.4.1.

Similar to the overlap configuration, this subsystem also requires manual specification of the coordinates for both drones. This is necessary because the transformation from pixel coordinates to geographic coordinates, required to assign GPS positions to detected and annotated objects, depends on accurate knowledge of the relative positions of the drones.

The YOLO model used in this subsystem was developed in previous work [3], and it performs sufficiently well to provide a foundation for object detection and the assignment of unique IDs that can be tracked across the merged image surface. However, the model exhibits certain limitations, particularly in frame-by-frame detection, where it occasionally fails to accurately identify objects. To compensate for this, the model is currently configured to operate at a low confidence threshold, which increases its sensitivity to potential objects. While this improves the likelihood of detecting true positives, it also raises the risk of false positives, that is, detecting objects that are not actually present. As a result, both false positives and false negatives may occur, leading to discrepancies between the expected number of objects and the actual detections recorded by the system.

To merge the image frames into a coherent panoramic image, alpha blending is used. This method has proven to be sufficiently effective for the objectives of the project, as all test scenarios are two-dimensional. However, if scenarios involving a third dimension (*i.e.*, incorporating depth) were introduced, more advanced image stitching techniques would be required. Such techniques include, for example, SIFT and homography-based methods, which aim to identify matching keypoints between image frames to enable robust image fusion. These methods were explored during the early development phase of the subsystem but could not be implemented in a usable way, most likely due to increased computational demands. This resulted in unstable panoramic images in which the frames exhibited temporal instability and misalignment between frames, which did not meet the system's requirements for stability and precision. Given that the test scenarios are limited to 2D use cases, alpha blending has proven to be an adequate solution for achieving the necessary object traceability across the panorama required for the drone system's full functionality.

5.6 Suggestions on Future Work

This subsection outlines potential directions for future work within the project, highlighting areas for improvement and proposing strategies to enhance the full system.

5.6.1 Communication

During testing at AstaZero, it was discovered that ATOS communication relies on an Ethernet cable connection. In the current system setup, the key functionality depends on all devices being connected to the same local wireless network. Therefore, a potential improvement is to implement an adapter to bridge Ethernet to the wireless network or transition to a fully Ethernet based solution. However, since the Android app requires a wired connection to the hand controller, this could present challenges in a fully Ethernet based setup.

An inefficiency observed during testing was the time required to manually enter IP address and port into the Android app to connect all devices. Therefore, another identified point of improvement is to eliminate this step. This would be a crucial step to undertake if the system were to be implemented for a large number of drones, which has been identified as another key area for improvement. One solution is that the Android app pings every possible private IP address on the network. If a server is running on that IP the server responds with a message the Android app recognizes. However, for IPv4 since a private IP can range from 10.0.0.0 to 10.255.255.255, 10/8 in CIDR prefix [51]. This could lead to 2^{24} IPv4 addresses which doesn't scale efficiently. An alternative is that running servers publish their IP to a list the Android app can access. Then the Android app checks the list and the user can easily connect to the servers in the list.

5.6.2 Extended Video Surveillance

Previous work [3] has identified several areas for improvement in the object detection model, particularly regarding its performance at varying flight altitudes and its ability to detect a broader range of object classes. Since the model has primarily been trained on image data captured from an altitude of approximately 40 meters, its accuracy decreases when applied at other heights [3]. This limitation is primarily attributed to insufficient training data, which could be addressed through extended data collection and additional training efforts [3]. Furthermore, efforts should focus on improving the model's robustness and reliability, as its current configuration fails to consistently deliver satisfactory detection results.

To extend the system's functionality to include three-dimensional interpretation of the environment, the implementation of depth estimation techniques is recommended. Algorithms such as SIFT and homography-based computations could be utilized to reconstruct depth information from multiple camera perspectives. This would notably improve the precision of image stitching, especially when handling multiple simultaneous video streams, where accurate spatial understanding is critical for achieving a coherent composite. Also, the current system is designed to operate

with two drones, which limits the area that can be covered by their camera views. Expanding the system to support a larger number of drones would overcome this limitation. However, integrating such a system introduces new challenges, including a more complex image stitching process.

Another proposed area for future development is the automation of parameter handling within the subsystem responsible for image stitching. In the present system, the user is required to manually specify the drone coordinates and image overlap settings directly in the source code. To enhance usability and reduce the risk of human error, these parameters should instead be retrieved automatically from other system modules. Such automation would increase the workflow and contribute to a more integrated and scalable system architecture.

In the current implementation, the coordinates of detected objects are not compared with the trajectories provided by ATOS. These trajectories typically include detailed information such as object type, position, and the expected time of arrival (*e.g.*, "Volvo at coordinates X, Y after T seconds"). A potential improvement would involve extracting and classifying objects from these trajectories so that the YOLO model adopts the same classification scheme. This would enable the system to compare expected object positions and classes with those estimated by the detection algorithm. If a significant mismatch is detected, an abort signal to ATOS via a ROS topic is triggered, terminating the scenario.

5.7 Social and Ethical Aspects

The relevant ethical concerns to consider during the project are mostly in the scope of personal integrity. As this project contains the use of automated camera surveillance from autonomous drones, therefore a risk of an invasion of privacy exists. However, during the use of the system, the video stream that is provided from the drones will not have the sufficient resolution to identify one particular identity. An important point of view from the social aspect is that drone operations regarding flight will be limited to operators with an approved license, as this is a legal requirement [52].

The ethical benefits if this project is to be used on a regular basis, is of course an improvement of safety at AstaZero's testing facility. The integrated multiple video streams will provide them with a larger surveillance coverage of the test range.

Social aspects can also be related to privacy, where the collection of data by drones can pose a problem for the security of vital installations, for example, installations requiring extra protection for sabotage, terrorism, espionage, and aggravated robbery. According to the Protection Act, a decision on a vital installation means that unauthorized persons do not have access to the object of protection. The prohibition of access also covers access by means of an unmanned aircraft system. A special decision may be issued prohibiting the taking of photographs, descriptions or measurements of or within the vital installation [53]. If the system developed during the project is misused or falls into the hands of someone with hostile intentions, it could pose a significant risk to these installations.

A social aspect that benefits from the use of drones is the rescue service. With the help of a drone, both endangered people and fires can be located faster. Drones contribute important information throughout the course of the rescue operation through their information retrieval [54]. If the project were to be successful, it could contribute to the rescue service in the future, as drone surveillance and detection of dangers are at the forefront in this sector as well.

During a discussion following the Strategic Foresight Workshop at Penn State, James E. Cartwright, a retired four-star general of the United States Marine Corps, emphasized his preference for using high-precision drones to eliminate specific targets rather than deploying ground troops, who might unintentionally inflict significant harm on bystanders. He argued that such an approach could preserve the lives of more soldiers while still achieving operational objectives and reducing collateral damage to civilians [55].

6 Conclusion

In this thesis, a platform for autonomous drone surveillance was developed. The system enables the positioning of multiple drones simultaneously, significantly expanding the effective surveillance area compared to previous solutions. As laid out in Section 3.4.2, the positioning algorithm optimizes the drone locations to ensure that both altitude and FOV are optimally tuned.

The system has fundamental functionality implemented of combining video streams for two drones. However, the subsystem has to be further developed if it is to handle more than two drones. As mentioned in Section 5.5 there is a need for tuning the overlap model. Additionally, this will help minimize the effect of ghosting, preventing the risk of misidentifying objects.

The developed frontend interface is capable of handling two video streams, both independently and stitched in real time. In the frontend interface, two drones can be seamlessly monitored and the system as a whole can be operated entirely autonomously. In conclusion, the frontend interface successfully meets all the desired functionalities outlined in the milestones Section 1.3.1. Nevertheless, the system is constrained by its substantial computational burden. Meaning, that running the system on a CPU only machine will affect performance. A solution is to run the different Docker services on different machines or run the entire system on a machine with a Nvidia GPU capable of CUDA computing Section 2.15.

The communication steps between each part of the system work as intended, with good performance as mentioned in Section 5.3 and Section 5.4. However, the communication can be improved between the backend and ATOS, ensuring that AstaZero's system can be run on an Ethernet connection. This will allow AstaZero to set up the drone surveillance system to their physical cable-based local network. Another related point of improvement is streamlining the user experience with the Android app or migrating the software into the backend completely.

Conclusively, the result of this thesis has contributed to AstaZero's continued work of improving autonomous test surveillance.

Bibliography

[1] TÜV SÜD. *What is Automotive Testing?* https://www.tuvsud.com/en/industries/mobility-and-automotive/automotive-and-oem/automotive-testing-solutions. [Accessed 2025-02-14].

[2] AstaZero. *About us - AstaZero.* Retrieved 2025-01-30. 2025. URL: https://astazero.ri.se/en/about-us/.

[3] Eddin Bilal Mohi et al. *Drone Platform for Safety in Autonomous Vehicle Testing.* Chalmers Open Digital Repository. 2024. URL: https://odr.chalmers.se/items/2ad5cf10-3358-41ac-9b99-38b23697e5e6.

[4] RISE Research Institutes of Sweden. *Annual and Sustainability Report 2024.* Swedish. Annual and Sustainability Report. Accessed 2024-05-16. Original title: Års- och hållbarhetsredovisning 2024. Gothenburg, Sweden: RISE Research Institutes of Sweden, Mar. 2025. URL: https://www.ri.se/sites/default/files/2025-03/RISE_ars-och-hallbarhetsredovisning-2024_web_0.pdf.

[5] AstaZero. *Test Site.* https://astazero.ri.se/en/tracks-facilities/test-site/. [Accessed 11-02-2025].

[6] AstaZero. *ATOS - Read the docs.* https://atos.readthedocs.io/en/latest/. [Accessed 11-02-2025]. 2024.

[7] AstaZero. *GitHub - RI-SE/ATOS: ROS2 based platform for coordinating tests of automated vehicles and their surrounding systems. — github.com.* https://github.com/RI-SE/ATOS. [Accessed 30-01-2025]. 2025.

[8] Amazon Web Services. *What is Docker?* https://aws.amazon.com/docker/. [Accsessed 11-04-2025]. 2025.

[9] Docker Inc. *What is Docker?* https://docs.docker.com/get-started/docker-overview/. Accessed: 2025-04-24. 2025. URL: https://docs.docker.com/get-started/docker-overview/.

[10] Docker Inc. *What is an image?* https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Accessed: 2025-04-24. 2025. URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/.

[11] Wikipedia. *Redis — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Redis&oldid=1282325057. [Online; accessed 17-April-2025]. 2025.

[12] Redis. *What is a Key-Value Database?* https://redis.io/nosql/key-value-databases/. [Online; accessed 18-April-2025]. 2025.

[13] Wikipedia. *Key-value database — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Key%E2%80%93value%20database&oldid=1258737249. [Online; accessed 05-May-2025]. 2025.

[14] Redis. *Redis Pub/Sub.* https://redis.io/docs/latest/develop/interact/pubsub/. Accessed: 2025-04-25. 2025. URL: https://redis.io/docs/latest/develop/interact/pubsub/.

[15] Wikipedia. *WebSocket — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=WebSocket&oldid=1285790539. [Online; accessed 25-April-2025]. 2025.

[16] DJI. *DJI DJISDKManager API Reference.* Accessed: 2025-04-28. 2025. URL: https://developer.dji.com/api-reference/android-api/Components/SDKManager/DJISDKManager.html.

[17] DJI. *DJI CodecManager API Reference.* Accessed: 2025-04-27. 2025. URL: https://developer.dji.com/api-reference/android-api/Components/CodecManager/DJICodecManager.html#djicodecmanager_enabledyuvdata_inline.

[18] DJI. *DJI VideoFeeder API Reference.* Accessed: 2025-04-27. 2025. URL: https://developer.dji.com/api-reference/android-api/BaseClasses/DJIVideoFeeder.html.

[19] Wikipedia. *JSON — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=JSON&oldid=1285495361. [Online; accessed 25-April-2025]. 2025.

[20] Mavlink. *MAVLink Developer Guide.* Apr. 2025. URL: https://mavlink.io/en/.

[21] Mavlink. *Mavlink.* https://github.com/mavlink/mavlink. 2025.

[22] Open Source Robotics Foundation. *Why ROS?* https://www.ros.org/blog/why-ros/. Accessed: 2025-04-21. 2024. URL: https://www.ros.org/blog/why-ros/.

[23] Adhiksha Thorat and Avinash Bhute. “SquashCord: Video Conferencing Application Using WebRTC.” In: *Lecture Notes in Electrical Engineering* 853 (2022). Cited by: 3, pp. 425–436. DOI: 10.1007/978-981-16-9885-9_35. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85127799947&doi=10.1007%2f978-981-16-9885-9_35&partnerID=40&md5=f5e617b2798c90f20e3d1322c6025222.

[24] AIORTC Contributors. *AIORTC Documentation.* Accessed: 2025-04-25. 2025. URL: https://aiortc.readthedocs.io/en/latest/.

[25] Google. *WebRTC: Getting Started with Remote Streams.* Accessed: 2025-04-27. URL: https://webrtc.org/getting-started/remote-streams.

[26] Chromium WebRTC Team. *DefaultVideoEncoderFactory.java Source Code.* Accessed: April 27, 2025. 2025. URL: https://chromium.googlesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/DefaultVideoEncoderFactory.java.

[27] Andrew D. Thompson et al. “YUV chrominance sub-sampling comparison using H.264.” In: vol. 9464. Cited by: 0. 2015. DOI: 10.1117/12.2181941. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84954069854&doi=10.1117%2f12.2181941&partnerID=40&md5=4d908840c8207445418c6b76b26d8362.

[28] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[29] Zheng Zou et al. "Object Detection in 20 Years: A Survey." In: *Proceedings of the IEEE* 111.3 (2023), pp. 257–276. DOI: 10.1109/JPROC.2023.3238524.

[30] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[31] Stanford University. *CS231M: Stitching and Blending*. Accessed: 2025-04-21. 2023. URL: https://web.stanford.edu/class/cs231m/lectures/lecture-5-stitching-blending.pdf.

[32] Jian Wang et al. "Synthesis of multiple video streams through multi-thread programming." In: *Proceedings of the 32nd Chinese Control and Decision Conference (CCDC)*. Hefei, China: IEEE, Aug. 2020, pp. 1152–1156. ISBN: 978-1-7281-5854-9. DOI: 10.1109/CCDC49329.2020.9164266.

[33] Aditya Ranjan and Mohamed Asan Basiri M. "High Performance Multicore Implementation of Motion Picture Estimation." In: *2023 IEEE 7th Conference on Information and Communication Technology (CICT)*. Jabalpur, India: IEEE, Dec. 2023. ISBN: 979-8-3503-0517-3. DOI: 10.1109/CICT59886.2023.10455400.

[34] Leaflet.js. *Leaflet JavaScript Mapping Library*. https://leafletjs.com/. Accessed: 2025-04-28. 2025. URL: https://leafletjs.com/.

[35] Michael A. Goodrich and Alan C. Schultz. "Human-Robot Interaction: A Survey." In: *Foundations and Trends in Human–Computer Interaction* 1.3 (2007), pp. 203–275. DOI: 10.1561/1100000005.

[36] Robin R. Murphy. *Introduction to AI Robotics*. 2nd. MIT Press, 2019. ISBN: 9780262038709.

[37] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices." In: *IEEE Access* PP (Mar. 2017). DOI: 10.1109/ACCESS.2017.2685629.

[38] NVIDIA. *CUDA Zone*. https://developer.nvidia.com/cuda-zone. [Online; accessed 6-May-2025]. 2025.

[39] MAVLink. *MAVLink Developer Guide*. https://mavlink.io/en/. [Accessed 13-02-2025].

[40] Ken Schwaber and Jeff Sutherland. *The Scrum Guide™*. Scrum.org and Scrum Alliance, 2020. URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf.

[41] Atlassian. *What is an Epic? Agile Project Management*. Accessed on January 31, 2025; publication date unknown. Atlassian. URL: https://www.atlassian.com/agile/project-management/epics (visited on 01/31/2025).

[42] Argo CD. *Argo CD Image Updater*. https://argocd-image-updater.readthedocs.io/en/stable/. [Accessed 13-02-2025].

[43] Adbullah Arshad, Arvid Boisen, Albin Hallander, Erik Rödin, Jesper Eriksson, Viggo Forsell. *GitHub - Drone platform for saftey in testing. — github.com.*

2025. URL: https://github.com/inspirers/Drone-platform-for-safety-in-testing.

[44] Suryansh Agrawal, Shashank Bommareddy, Aidan Brown, Rahique Mirza, Jaden Peacock, Eugene Sosa, Jack Volgren. *Drone Platform for Safety in Testing*. State Collage PA, USA: Pennsylvania State University, Apr. 2025. URL: https://sites.psu.edu/lfshowcasesp25/2025/04/29/drone-platform-for-safety-in-testing/.

[45] David Eberly. *Minimum-Area Rectangle Containing a Set of Points*. https://www.geometrictools.com/Documentation/MinimumAreaRectangle.pdf. [Accessed 25-04-2025]. 2015.

[46] Melanie. *You Only Look Once (YOLO): What is it?* Accessed on February 5, 2025; publication date unknown. DataScientest.com. URL: https://datascientest.com/en/you-only-look-once-yolo-what-is-it (visited on 02/05/2025).

[47] Adrian Rosebrock. *Increasing Webcam FPS with Python and OpenCV*. Accessed: 2025-04-21. 2015. URL: https://pyimagesearch.com/2015/12/21/increasing-webcam-fps-with-python-and-opencv/.

[48] Transportstyrelsen. *Guide to UAS operations in Sweden*. Accessed: 2025-05-05. URL: https://www.transportstyrelsen.se/en/aviation/aircraft/drones-unmanned-aircraft/guide-to-uas-operations-in-sweden/.

[49] Protocolbuffers. *Protocol buffers*. Apr. 2025. URL: https://protobuf.dev/.

[50] Wikipedia. *Protocol Buffers — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Protocol%20Buffers&oldid=1284635386. [Online; accessed 28-April-2025]. 2025.

[51] Robert Moskowitz et al. *Address Allocation for Private Internets*. RFC 1918. Feb. 1996. DOI: 10.17487/RFC1918. URL: https://www.rfc-editor.org/info/rfc1918.

[52] David Daly. *When Do You Need a Drone License?* https://www.flyingmag.com/when-do-you-need-a-drone-license/. [Accessed 30-01-2025]. 2024.

[53] Regeringskansliet. *Skyddslag (2010:305)*. Swedish. 2010. URL: https://www.riksdagen.se/sv/dokument-och-lagar/dokument/svensk-forfattningssamling/skyddslag-2010305_sfs-2010-305/.

[54] Myndigheten för samhällsskydd och beredskap. *UAS*. Swedish. 2023. URL: https://www.msb.se/sv/amnesomraden/skydd-mot-olyckor-och-farliga-amnen/raddningstjanst-och-raddningsinsatser/metod-och-teknikutveckling-for-raddningstjansten/uas/.

[55] James E. Cartwright. *Discussion at the Strategic Foresight Workshop at Penn State*. Personal communication, March 2025. 2025.

A

Code

A.1 Message type in backend

```
1       async def on_message(self, frame: str,
          connection_id: str) -> None:
2           """Processes incoming messages."""
3           try:
4               data = json.loads(frame)
5               # print(f"Received message: {data}")
6
7               msg_type = data.get("msg_type")
8               if not msg_type:
9                   raise ValueError(f"Missing␣‘msg_type‘␣in␣
                      message:␣{data}")
10
11              # Route messages based on ‘msg_type‘
12              if msg_type == "Coordinate_request":
13                  await self.send_coords(connection_id)
14              elif msg_type == "Position":
15                  self.incoming_position_handler(data,
                      connection_id)
16              elif msg_type == "Debug":
17                  msg = data.get("msg", "")
18                  print(f"Debug␣message:␣{msg}")
19              elif msg_type == "candidate":
20                  candidate_sdp = data.get("candidate")
21                  sdp_mid = data.get("sdpMid",
                      data.get("id", "0"))
22                  mline_index =
                      int(data.get("sdpMLineIndex",
                      data.get("label", 0)))
23
24                  if not candidate_sdp:
25                      print(f"[RTC␣WARNING]␣Missing␣
                          ’candidate’␣field:␣{data}")
26                      return
27
```

```
28                  # Parse the candidate line properly
29                  rtc_candidate =
                     candidate_from_sdp(candidate_sdp)
30
31                  rtc_candidate = RTCIceCandidate(
32                      foundation=rtc_candidate.foundation,
33                      component=rtc_candidate.component,
34                      priority=rtc_candidate.priority,
35                      ip=rtc_candidate.ip,
36                      port=rtc_candidate.port,
37                      protocol=rtc_candidate.protocol,
38                      type=rtc_candidate.type,
39                      sdpMid=sdp_mid,
40                      sdpMLineIndex=mline_index,
41                      relatedAddress=rtc_candidate.relatedAddress,
42                      relatedPort=rtc_candidate.relatedPort,
43                      tcpType=rtc_candidate.tcpType,
44                  )
45
46                  await
                     self.peer_connections[connection_id].
47                      addIceCandidate(
48                      rtc_candidate
49                  )
50                  print(
51                      f"[RTC]␣Added␣ICE␣candidate␣from␣
                         {connection_id}:␣{candidate_sdp}"
52                  )
53
54              elif msg_type == "answer":
55                  # Todo: Handle SDP answer
56                  if connection_id in
                     self.peer_connections:
57                      sdp = data.get("sdp")
58                      sdp_type = data.get("type") or
                         data.get("msg_type")
59
60                      if sdp_type not in ("answer",
                         "offer"):
61                          print(f"[RTC␣ERROR]␣Unexpected␣
                             SDP␣type:␣{sdp_type}")
62                          return
63
64                      await
                         self.peer_connections[connection_id].
65                          setRemoteDescription(
```

```
66                              RTCSessionDescription(sdp=sdp,
                                  type=sdp_type)
67                          )
68
69                  else:
70                      print(
71                          f"[DroneStream]␣ERROR:␣Peer␣
                              connection␣for␣{connection_id}␣
                              not␣found."
72                      )
73                  print(f"Received␣SDP␣answer␣from␣
                      {connection_id}:␣{data}")
74              else:
75                  print(f"Unhandled␣‘msg_type‘:␣
                      {msg_type}")
76          except json.JSONDecodeError:
77              print(f"Malformed␣JSON␣received␣from␣
                  {connection_id}:␣{frame}")
78          except Exception as e:
79              print(f"Error␣processing␣message␣from␣
                  {connection_id}:␣{e}")
```

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY