University of
South-Eastern Norway

www.usn.no

TS3000 Bacheloroppgave IRI

Image Processing on the Edge

Aerial Edge
SOFTWARE DEVELOPMENT

Group 6 - Aerial Edge

Faculty of Technology, Natural Sciences and Maritime Sciences

Campus Kongsberg

University of South-Eastern Norway

www.usn.no

Course: *TS3000 Bacheloroppgave IRI*
Title: *Image Processing on the Edge*

Project group: *Group 6 - Aerial Edge*

Group participants: *SINDRE NES,*
EVEN JØRGENSEN,
ABDUL MAJEED ALIZAI,
ÅDNE KVÅLE,
MARTIN BØRTE LIESTØL,
JON JAHREN

Supervisor: *JAN DYRE BJERKNES*
Project partner: *Kongsberg Defence & Aerospace*

Summary:
The Local Hawk drone project, using Single Board Computers (SBCs), is struggling with tasks like motor control and object detection due to the limitations of the SBCs. To address these issues, this project examined four different hardware and software setups for better object detection in lightweight drones. We evaluated each setup based on how accurately they could detect objects, how quickly they could process images, their power usage, weight, and how complex they were to set up and use. The study highlighted the need to carefully choose the right setup to get the best performance in object detection. This research provides valuable insights into image processing for lightweight drones, which could benefit the Local Hawk project and similar efforts.

1 Acknowledgements

We'd like to say a big thank you to our supervisors, Jan Dyre Bjerknes and Henning Gundersen. Their advice and support really helped us get through this project.

Additionally we would like to thank our university and our department for providing us with the resources and environment to learn and carry out this project. The opportunities and support we've received here have been invaluable.

2 Abstract

The Local Hawk drone project, currently using Single Board Computers (SBCs), faces difficulties in simultaneous task execution such as motor control and object detection due to the limitations of the SBCs. The necessity to improve frame rates and efficiency led to an exploration of various hardware and software configurations.

The primary aim of the research was to evaluate and compare four different configurations to optimize object detection capabilities in lightweight Unmanned Aerial Vehicles (UAVs). The key areas of focus were processing power, accuracy, and energy efficiency.

The study conducted an exhaustive benchmarking of four hardware and software configurations, evaluating them on various parameters such as detection accuracy (precision, recall, and F1-score), frame rate, power consumption, weight efficiency, and complexity of setup and operation.

The analysis demonstrated the importance of carefully selecting hardware and software configurations to achieve optimal object detection performance within the constraints of lightweight UAVs. It was found that configurations varied greatly in their precision, recall, and F1-scores, with different trade-offs between frame rate and power consumption. Additionally, weight efficiency and complexity of setup and operation played crucial roles in determining the overall suitability of each configuration.

This research significantly contributes to the understanding of edge image processing for lightweight UAVs, serving as a foundation for future investigations in this area. The results hold practical relevance for the Local Hawk project and similar endeavors aiming to enhance the capabilities of lightweight UAVs in applications ranging from surveillance to search-and-rescue operations.

Keywords: Single Board Computers (SBC), Local Hawk, Unmanned Aerial Vehicles (UAV), object detection, frames-per-second (FPS), edge image processing, hardware configuration, software configuration, lightweight UAV technologies.

Kongsberg, 22nd May 2023

Contents

1 Acknowledgements **4**

2 Abstract **5**

Contents **8**
List of Figures 10

3 Introduction **11**
3.1 Introduction 11
3.2 Problem Domain 11
3.3 Research Perspective 12

4 Process **15**
4.1 Project Tools 17
4.1.1 Github 17
4.1.2 Taiga 17
4.1.3 Overleaf 18
4.1.4 ChatGPT 19
4.1.5 Microsoft Suite 19
4.2 Risk Analysis 20

5 Configurations **23**
5.1 Hardware Selection Process 24
5.1.1 nVidia Jetson Nano 24
5.1.2 Coral Edge TPU 25
5.1.3 Raspberry Pi 4B & Zero 2 26
5.1.4 Pi Camera modules 27
5.2 Operating System and System Software Decision Process 28
5.2.1 Raspberry Pi 4 and Zero 2 35
5.2.2 Jetson Nano System Software Configuration 35
5.2.3 32-bit vs. 64-bit 38
5.3 Proposed architectures 38
5.3.1 Initial Proposal 38
5.3.2 Second proposal 40
5.3.3 Accepted architectures 41

5.4 Object detection software . . . 44
5.5 Distance Measurement . . . 45
5.6 Proposed Operating System and System Configuration . . . 46
5.7 Image processing modules . . . 47
5.7.1 Configuration 1, Jetson Nano . . . 48
5.7.2 Configuration 2, Pi 4 w/ Coral Edge TPU . . . 56
5.7.3 Configuration 3, Pi Zero w/ Coral Edge TPU . . . 60
5.7.4 Configuration 4, Pi 4 . . . 60
5.8 Configurations, full context . . . 66
5.8.1 ROS2 integration . . . 66
5.8.2 UAV side project . . . 67
5.9 Drone architecture . . . 68
5.9.1 Flight controller firmware setup . . . 68
5.9.2 Communication software setup . . . 69
5.10 Exploring Use cases . . . 71
5.10.1 Qualisys and drone tracking . . . 71

6 Measurements **76**
6.1 Config1 . . . 80
6.2 Config2 & Config3 . . . 80
6.2.1 EfficientDet Lite 0 . . . 81
6.2.2 EfficientDet Lite 1 . . . 82
6.2.3 USB 3.0 vs. USB 2.0 . . . 84
6.3 Config4 . . . 86

7 Conclusion **87**

References **88**

A Hardware **94**
A.1 Specification tables for configs . . . 94
A.1.1 Config 1 . . . 94
A.1.2 Config 2 . . . 94
A.1.3 Config 3 . . . 95
A.1.4 Config 4 . . . 95
A.1.5 Comparison . . . 96
A.2 Single-board computer (SBC) . . . 96
A.3 Camera . . . 98
A.3.1 Rolling vs. Global shutter . . . 98
A.3.2 Camera drivers . . . 99
A.4 Hardware Acceleration for ML Inference . . . 100
A.4.1 Google Coral TPU . . . 100
A.5 Flight controller . . . 100

B UAV **102**
B.1 Flight controller 102
B.1.1 Firmware 102
B.1.2 MAVLink 103
B.1.3 Drone simulation 105
B.2 Drone implementation 106

C Config4 Code Explanation and Analysis **113**
C.1 Blob detection.v1 113
C.2 Blob detection.v2 115
C.3 Blob detection.v3 118
C.4 Blob Detection.v3 Evaluation 121

D Config4 Source code **126**
D.1 Blob Detection.v1 126
D.2 Blob Detection.v2 127
D.3 Blob Detection.v3 129

E ROS1 to ROS2 **132**
E.1 Translating from ROS1 to ROS2 132
E.1.1 Syntax Changes 132
E.1.2 Architecture 133

F Yolo training Tutorial **135**

G Source code, ROS2 nodes **141**
G.1 Source code, ROS2 nodes 141
G.1.1 Image processing, configuration 1 141
G.1.2 Image processing, configuration 2/3 143
G.1.3 Object follower 145
G.1.4 Image Processing, configuration 4 149
G.1.5 Drone Position Estimation 153
G.1.6 Image Processing, configuration 4 (Code Only) 156

H Tensorflow Lite Model Maker script **160**
H.1 Tensorflow Lite Model Maker script 160

I Team **161**
I.1 Team Members 161

List of Figures

3.1 Hermenutic Spiral [5] 14

4.1 Sprint organization [6] 16
4.2 Taiga Interface 18
4.3 Risk table 21
4.4 Risk Matrix [10] 22

5.1 Jetson nano [12] 25
5.2 Coral USB accelerator [13] 25
5.3 Proposed Operating System Configuration 28
5.4 Example ROS2 graph 29
5.5 Docker architecture 31
5.6 Decision Process OS 32
5.7 Decision Process Distribution 33
5.8 Jetson Nano Decision Map 36
5.9 End Result Jetson Nano Architecture 37
5.10 Initial architecture design 39
5.11 Image processing modules 40
5.12 Four distributed and one centralized architecture 41
5.13 Hardware architectures accepted by client 42
5.14 Initial software architecture design 43
5.15 Finished Overview System Configuration 46
5.16 Hardware architecture, config 1 48
5.17 Visual representation of the journey 49
5.18 Deepstream app config file, source0 51
5.19 config_infer_primary_YoloV5.txt, custom model 51
5.20 YoloDET before 53
5.21 YoloDET after 54
5.22 Selecting our model 54
5.23 extracting metadata original 54
5.24 extracting metadata modified 54
5.25 Hardware architecture, config 2 56
5.26 Pascal VOC .xml contents (left), corresponding image (right) 58
5.27 Software, config 2 59
5.28 Hardware architecture, config 3 60

5.29 Hardware architecture, config 4 60
5.30 Config model, config 4 62
5.31 ColorFinder object, config 4 63
5.32 BlobDetection.v2, config 4 64
5.33 Trying different HSV combinations 65
5.34 ROS2 node description 67
5.35 Hardware architecture including the flight controller and ESCs 68
5.36 Dataflow during test scenario 70
5.37 Estimated drone position vs Qualisys tracking data 73

6.1 Results represented in spider diagrams 79
6.2 Yolov5 nano versus small model FPS 80
6.3 EfficientDet Lite 0 on various Raspberry Pies 81
6.4 EfficientDet Lite 1 on various Raspberry Pies 82
6.5 EfficientDet Lite 0 compared to EfficientDet Lite 1 on Raspberry Pi 4B . . 83
6.6 EfficientDet Lite 0, USB 3.0 vs 2.0 on RPi 4B 84
6.7 EfficientDet Lite 1, USB 3.0 vs 2.0 on RPi 4B 85
6.8 FPS, Configuration 4 86

A.1 libcamera-vid piping video to gstreamer 99
A.2 v4l2src (RaspiCam) feeding video to gstreamer 99
A.3 Coral USB accelerator [13] 100
A.4 Showcase of MATEKSYS H743-SLIM, from mateksys.com [1] 101

B.1 ArduPilot logo, from ardupilot.org [2] 102
B.2 SITL with Gazebo 106
B.3 Photo of drone from top 107
B.4 Photo of drone from left 108
B.5 Photo of drone from right 109
B.6 Photo of drone from front 110
B.7 Photo of drone hovering in the air 111
B.8 Photo taken from drone's perspective doing object detection 112

3 Introduction

3.1 Introduction

We are a team of six computer engineering students from the University of South-Eastern Norway, Campus Kongsberg. Our bachelor assignment was given to us by Kongsberg Defence & Aerospace (KDA), a Norwegian technology company headquartered in Kongsberg. KDA specializes in manufacturing equipment for defense, space exploration, and aviation.[3]

Our client conducts a student-centered initiative known as 'Local Hawk', which is operational during the summer. The main focus of this initiative is to investigate a variety of methods for fostering the development of autonomous drones. The client has expressed an interest in our project with the aim of garnering insightful data that could be applied to future deliberations concerning the architectural design of these unmanned aerial vehicles, with special emphasis on Frames-Per-Second (FPS), as this is the most crucial and important aspect.

The drone systems traditionally used in the Local Hawk project has limited computing power and restrictions on weight. KDA expressed an interest in doing a research project for our assignment, where we would examine any potential performance gains by moving the processing closer to the sensor hardware, meaning we will be using dedicated hardware for image processing.

3.2 Problem Domain

The drones being used in the Local Hawk project today use what we call Single Board Computers (SBC). These devices are usually created in a small form factor, and due to their limited size, they are also limited in processing capabilities. This means that it has challenges performing several tasks simultaneously, e.g., controlling the drone motors and executing object detection at the same time.

In earlier iterations of Local Hawk they attempted to run object detection while flying the drone at the same time on a single SBC. This resulted in very low framerate which in turn meant that it could not be used for any meaningful purpose. The reason for this is that processing camera images can be computationally expensive. For instance, if we need to process a 24-bit color image with a 600x600 resolution pixel-by-pixel, we would have to handle 360,000 pixels, each with 3 color channels, resulting in an input data size of 1,080,000 bytes per image. This poses a challenge for the limited hardware available for a lightweight drone.

The primary objective of this study is to meticulously evaluate various software and hardware architectures for a compact UAV equipped with object detection capabilities. We will place a particular emphasis on identifying the potential frames-per-second (FPS) improvements attainable on our systems, as this is our main focus. Each architecture will be critically analyzed and compared based on parameters such as performance, cost, complexity, and weight.

While the aspiration of implementing these architectures in an actual UAV or drone system is considered as a secondary or stretch goal, the core of our research will provide valuable insights and recommendations. These recommendations will cater to the technology selection process when designing lightweight UAVs for diverse applications requiring object detection capabilities. By doing so, our findings are expected to significantly contribute to the design of efficient and effective UAV systems.

In the context of the Local Hawk project, image processing is particularly essential when considering the use case of ”Autonomous Landing”. UAVs operate in diverse environments with various terrains and conditions, making precise and safe landings a challenge. Through image processing, the drone can significantly enhance its autonomous landing capabilities. One such improvement is the incorporation of visual markers to indicate the designated landing spot. By identifying these markers, the drone can ascertain the designated landing spot, ensuring a more precise and controlled landing.

3.3 Research Perspective

In our initial discussions with the client, it was clearly communicated that they desired a research report containing actionable information for decision-making, provided our findings indicate the potential for a successful endeavor.

This presented challenges for us, as we lacked prior experience with research-focused projects. Consequently, we needed to adapt our approach in order to comprehend how best to organize and execute our work, ensuring optimal delivery of results.

During the initial stages of the project, we deliberately refrained from immediately engaging with specific technologies or embarking on the development of a product in a domain where our knowledge was limited. Instead, we adopted a hermeneutic process, which is not commonly employed in engineering or systems engineering. However, this methodology proves valuable when dealing exclusively with knowledge and information, rather than specific implementations.[4]

The diagram of the hermeneutic spiral presented below illustrates the practical application of this concept. As our understanding of the problem domain deepens, we gained the ability to formulate more refined inquiries regarding the information that is relevant to us and the client. This enhanced understanding guided us in approaching the problem domain with the aim of achieving the desired results.

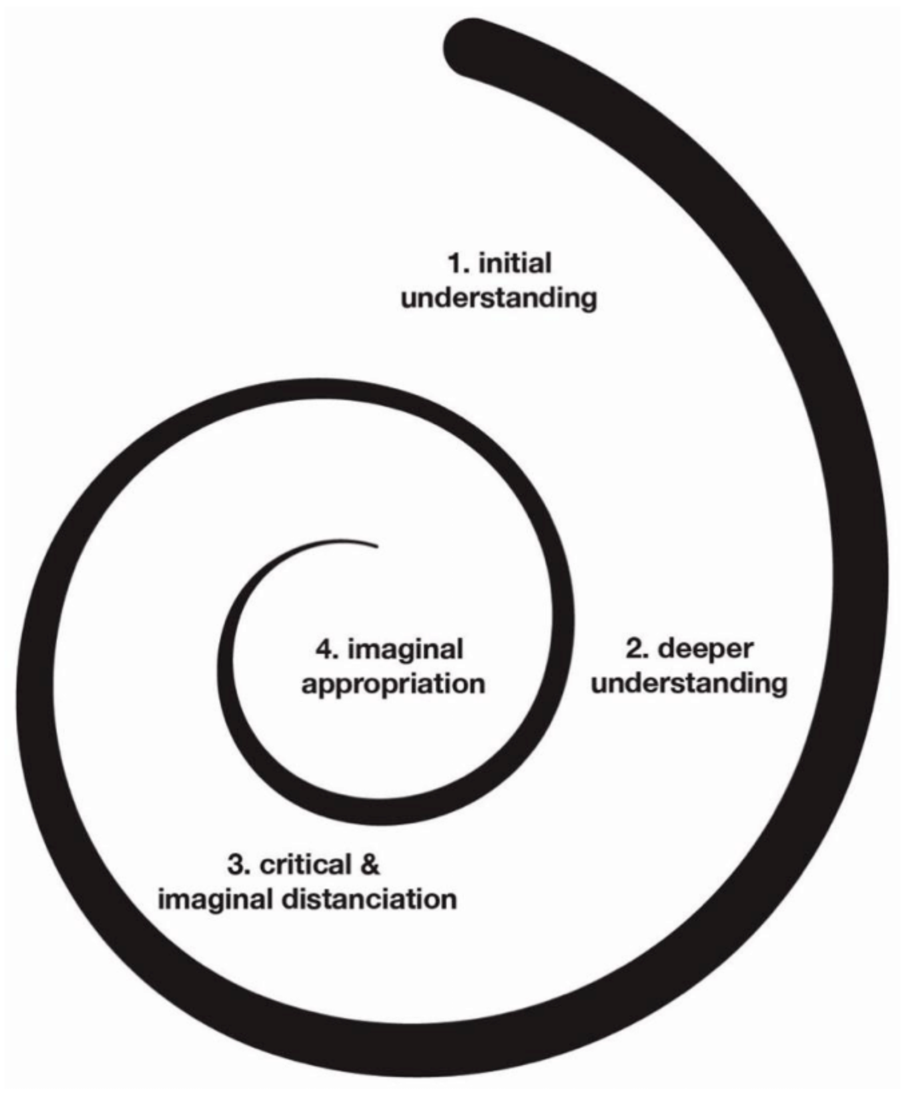

1. initial understanding
2. deeper understanding
3. critical & imaginal distanciation
4. imaginal appropriation

Figure 3.1: Hermenutic Spiral [5]

4 Process

In this section, we will outline and explain how we transitioned from our initial phase and organized our project in order to facilitate work on practical implementation. We will explain which tools we utilized and methodology we used to achieve our results.

While the hermeneutic methodology facilitated invaluable insights during the exploratory stage of our project, its emphasis on interpretation and understanding proved less applicable when transitioning from knowledge acquisition to knowledge application. The hermeneutic approach, with its focus on understanding texts or phenomena in depth, provides a foundation for comprehensive understanding rather than prioritizing explicit task completion. Consequently, as the project shifted to a phase demanding direct action and the tangible execution of tasks, the inherent characteristics of the hermeneutic approach were less aligned with these new requirements. Therefore, to maintain project efficiency during the implementation phase, it became necessary to consider alternative methodologies more attuned to the objectives of this new stage.

We decided that our project demands were best met using an agile methodology. This methodology is an iterative approach to software development and project management that emphasizes flexibility, collaboration, and client satisfaction. It advocates adaptive planning, evolutionary development, early delivery, continual improvement, and encouraging rapid and flexible response to change. Agile methods break tasks into smaller increments with minimal planning and do not directly involve long-term planning. This methodology prioritizes direct communication over extensive documentation, producing working software that evolves through a collaborative effort between self-organizing cross-functional teams.

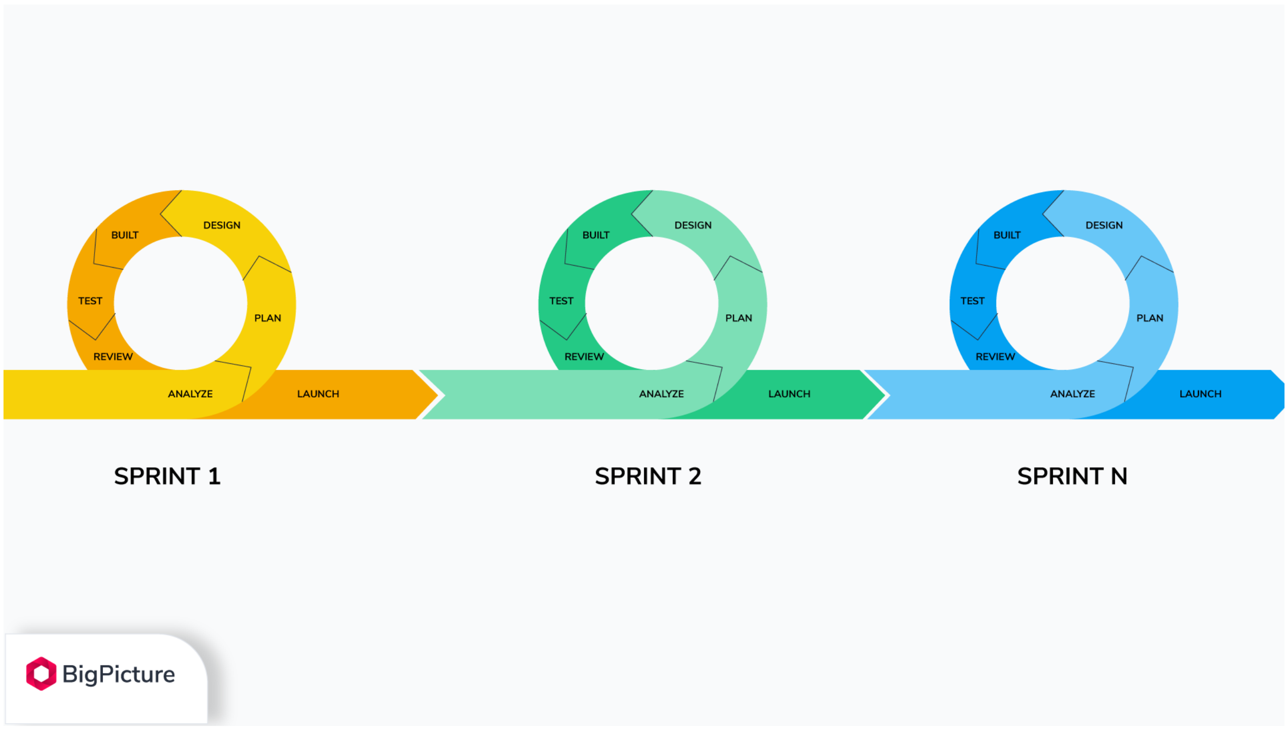


Figure 4.1: Sprint organization [6]

An agile workflow has a number of key terms and features, and there are several different ways to conduct an agile project. Common for all of these, however, is that these projects are divided into short work cycles known as iterations or sprints, typically lasting between one and four weeks. Each sprint has a defined goal and a set of tasks to be completed. We decided that for our project we would not adhere to a strict definition, e.g., scrum. Instead, we opted to pick and choose between different features that made the most sense for our project, and to make sure that we utilized the most prominent features that are essential to an agile project. These features were: sprint, sprint backlog, daily stand-up meetings, and retrospective.

Following the agile methodology, we decided to have daily stand-up meetings where we would update the group on our individual progress. We organized our sprints to last one week at a time, where we had an initial meeting on Monday to set up our tasks for the week, and a meeting at the end of the week where we updated our client on where we were in terms of progress. In the end-of-week meetings, the client was invited to give feedback on which tasks they wanted us to prioritize going forward into the next sprint.

4.1 Project Tools

4.1.1 Github

We needed a way to organize and collaborate on code, and essential to this is Source Code Management (SCM), and for this purpose, our choice fell on git. Git is a distributed SCM, originally developed for use on the Linux kernel, but has since gained a significant market share and is now the dominant SCM. Git's primary function is to enable developers to create different versions of their projects and switch between these versions seamlessly. This means that developers can experiment with different features and code changes without impacting the main, stable codebase. If a change works well, it can be integrated (or "merged") into the main codebase; if not, it can be discarded without having caused any disruption.

In a team environment, Git is essential for managing contributions from multiple developers. Each developer can work on their own copy of the project (a "branch"), without interfering with others' work. When their work is complete, it can be merged into the main codebase.

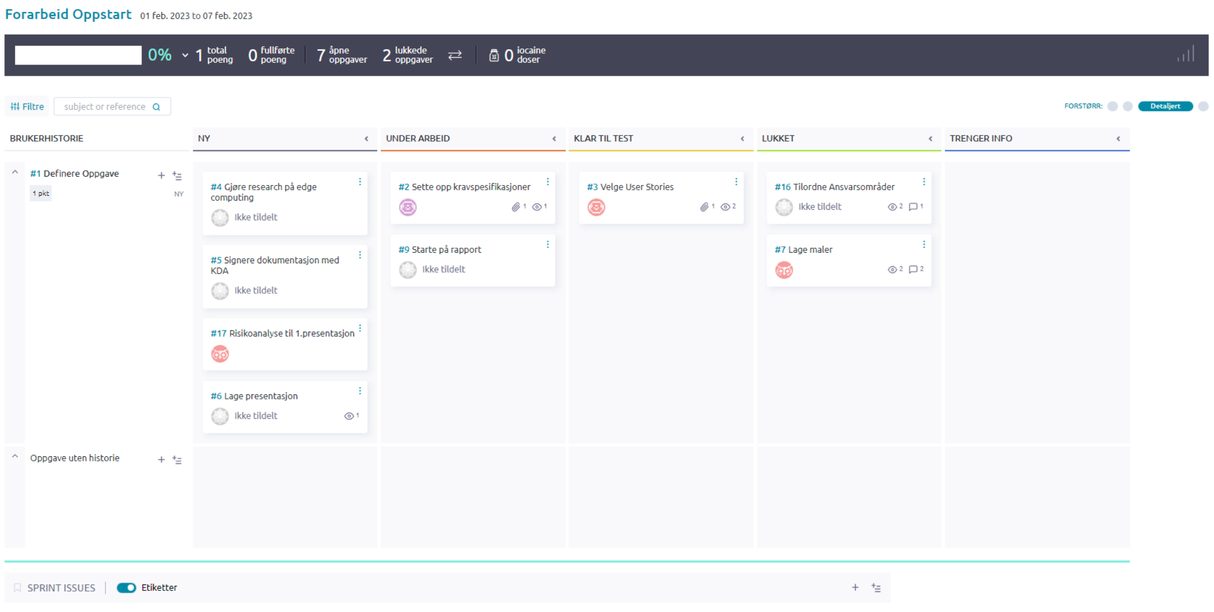
Git is also distributed, meaning every developer has a complete copy of the project's history on their local machine. This not only allows developers to work offline but also provides an inherent backup. If any repository is lost, it can be restored from any developer's local copy.

Instead of organizing our own server with git, we opted for using GitHub, which allows students and educators access to their professional option at no cost, allowing us to organize our code in a more structured manner.

4.1.2 Taiga

In order to organize our agile workflow we elected to use Taiga. It is a web-based tool that allows us to track sprints, tasks within sprints, and assign aforementioned tasks to specific members of the group, and additionally allow members to follow updates on a specific task. In addition to this, Taiga has integration with GitHub, allowing us to modify tasks from GitHub whenever we commit code in our repository there.

As the project progressed, we found that we were not generating a substantial amount of code, and therefore, the GitHub integration was not as crucial as we initially assumed.



Forarbeid Oppstart 01 feb. 2023 to 07 feb. 2023

0% 1 total poeng 0 fullførte poeng 7 åpne oppgaver 2 lukkede oppgaver 0 iocaine doser

Filtre | subject or reference | FORSTØRR: Detaljert

BRUKERHISTORIE	NY	UNDER ARBEID	KLAR TIL TEST	LUKKET	TRENGER INFO
#1 Definere Oppgave (1 pkt, NY)	#4 Gjøre research på edge computing — Ikke tildelt	#2 Sette opp kravspesifikasjoner	#3 Velge User Stories	#16 Tilordne Ansvarsområder — Ikke tildelt	
	#5 Signere dokumentasjon med KDA — Ikke tildelt	#9 Starte på rapport — Ikke tildelt		#7 Lage maler	
	#17 Risikoanalyse til 1.presentasjon				
	#6 Lage presentasjon — Ikke tildelt				
Oppgave uten historie					

SPRINT ISSUES | Etiketter

Figure 4.2: Taiga Interface

Moreover, our interaction with the Taiga platform was proving to be more of a diversion from our core tasks rather than a facilitator of our work. The platform's organization and functionality did not meet our needs and expectations, which caused further dissatisfaction. Consequently, we decided to transition away from this web-based project management tool. Instead, we opted for a more streamlined approach of crafting summaries for each sprint. This method proved to be less demanding in terms of resources and was more suitable and beneficial for the progression of our project.

4.1.3 Overleaf

Overleaf is a collaborative, cloud-based LaTeX editor used for the creation of scientific documents. LaTeX is a typesetting system favored in academia for its ability to handle technical and scientific documents with ease, especially those with complex mathematical equations or structures.

Overleaf provides a platform where multiple users can view, edit, and compile LaTeX documents in real time. This collaboration aspect makes it an excellent tool for group projects, theses, papers, or any document requiring input from several contributors.

4.1.4 ChatGPT

ChatGPT is an artificial intelligence language model, released to the public recently. It is designed to understand natural language and generate human-like responses to a wide range of prompts and questions.

ChatGPT is most useful for tasks that require natural language processing, such as language translation, sentiment analysis, text summarization, and conversational interfaces. It can also be used for a variety of other applications, such as content generation, language modeling, and knowledge extraction.

We have on occasion used ChatGPT for cleaning up and helping us formulate language in a more academic and formal fashion.

4.1.5 Microsoft Suite

Microsoft Office is a suite of productivity applications that have become a standard tool in most professional and academic environments. It includes software like Word for document creation, Excel for data management and analysis, PowerPoint for presentations, Outlook for email and calendar management, and more recently Teams for collaborative communication. Each of these applications serves distinct purposes and can be instrumental in managing and executing a project efficiently.

For our purposes, we organized a lot of our work through the university-provided teams solution, where we organized documents that were impractical to use LaTeX for. This is where we kept time sheets, presentation material, and meeting notes.

4.2 Risk Analysis

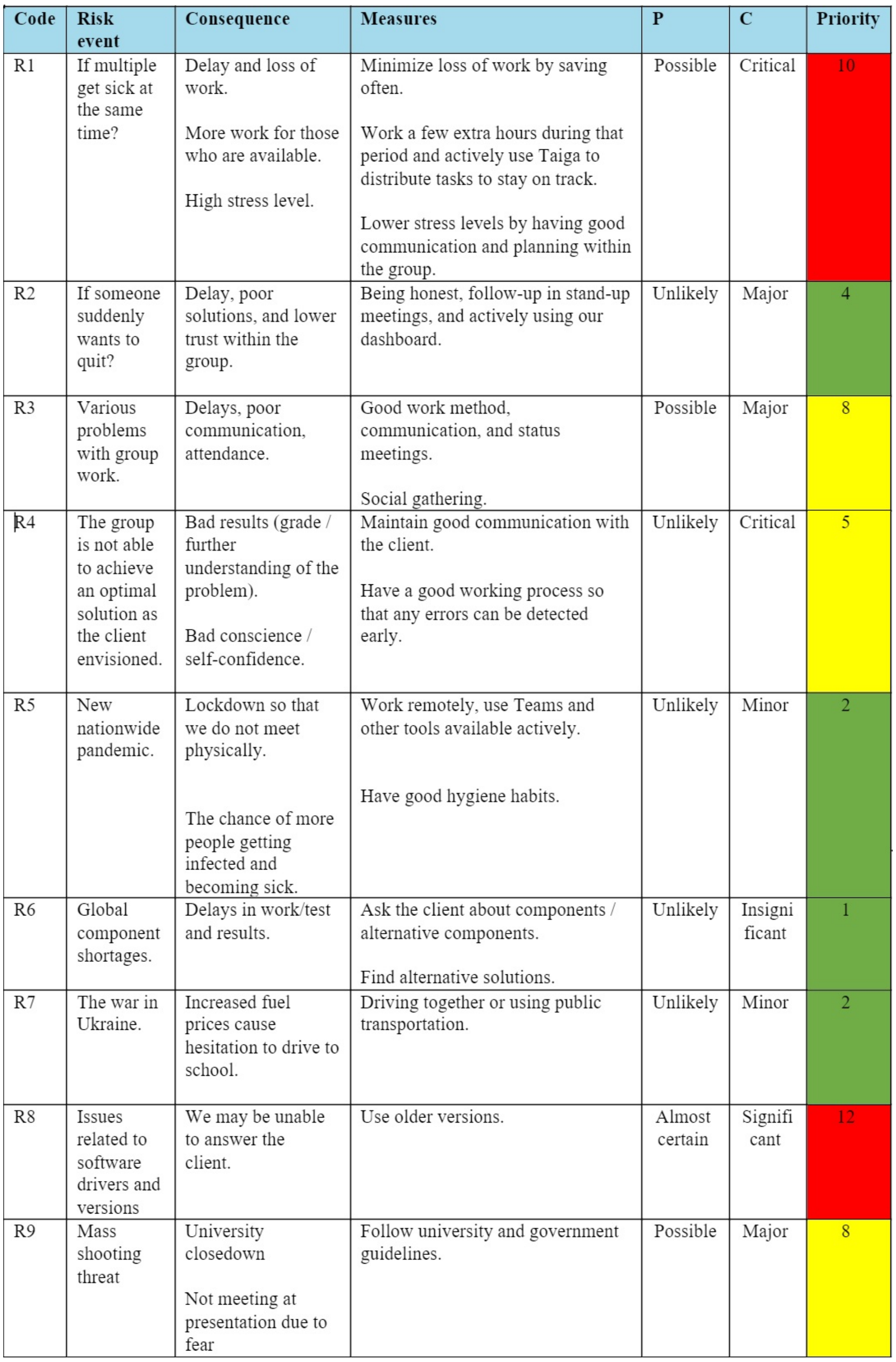
Risk analysis is an ongoing process that continuously evaluates risk throughout the project's lifecycle. It is the responsibility of the risk manager to ensure that this process takes place regularly and consistently during the project. This is crucial because it raises awareness among us and our client about potential risks and vulnerabilities, encourages necessary improvements, and facilitates necessary changes. Such analysis can help the risk manager to identify new risks and changes that require attention along the way. [7]

After identifying the risks, it is essential to prioritize them based on their probability and consequence. Moreover, measures should be put in place to manage them effectively if they occur. While everyone on the team should participate in assessing the project's risks, the risk manager will be primarily responsible for ensuring quality assurance. [8]

Identifying the risks in our project

A risk analysis was conducted for our project, wherein we identified both internal and external risks. Internal risks are linked with factors that are under our control, whereas external risks are associated with factors that lie beyond our control.[9]

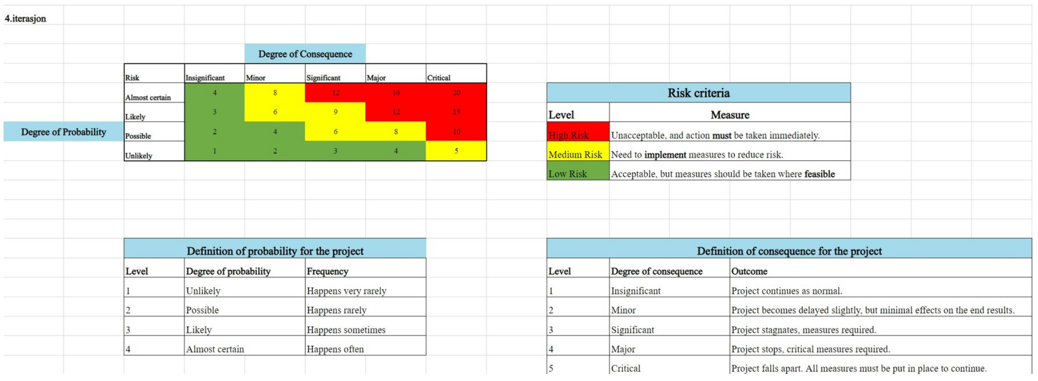
After identifying the risks at this stage, we evaluate their potential consequences and determine the appropriate measures that can be taken if they occur. To gain a better overview, we record the risks in a table that includes a unique code, a description of the risk event, recommended measures, probability (P), consequence (C), and priority. Below is a comprehensive overview of both internal and external risks:



Code	Risk event	Consequence	Measures	P	C	Priority
R1	If multiple get sick at the same time?	Delay and loss of work. More work for those who are available. High stress level.	Minimize loss of work by saving often. Work a few extra hours during that period and actively use Taiga to distribute tasks to stay on track. Lower stress levels by having good communication and planning within the group.	Possible	Critical	10
R2	If someone suddenly wants to quit?	Delay, poor solutions, and lower trust within the group.	Being honest, follow-up in stand-up meetings, and actively using our dashboard.	Unlikely	Major	4
R3	Various problems with group work.	Delays, poor communication, attendance.	Good work method, communication, and status meetings. Social gathering.	Possible	Major	8
R4	The group is not able to achieve an optimal solution as the client envisioned.	Bad results (grade / further understanding of the problem). Bad conscience / self-confidence.	Maintain good communication with the client. Have a good working process so that any errors can be detected early.	Unlikely	Critical	5
R5	New nationwide pandemic.	Lockdown so that we do not meet physically. The chance of more people getting infected and becoming sick.	Work remotely, use Teams and other tools available actively. Have good hygiene habits.	Unlikely	Minor	2
R6	Global component shortages.	Delays in work/test and results.	Ask the client about components / alternative components. Find alternative solutions.	Unlikely	Insigni ficant	1
R7	The war in Ukraine.	Increased fuel prices cause hesitation to drive to school.	Driving together or using public transportation.	Unlikely	Minor	2
R8	Issues related to software drivers and versions	We may be unable to answer the client.	Use older versions.	Almost certain	Signifi cant	12
R9	Mass shooting threat	University closedown Not meeting at presentation due to fear	Follow university and government guidelines.	Possible	Major	8

Figure 4.3: Risk table

The prioritization of these risks was determined by evaluating their consequence and probability using the risk matrix:



4.iterasjon

		Degree of Consequence				
	Risk	Insignificant	Minor	Significant	Major	Critical
	Almost certain	4	8	12	16	20
	Likely	3	6	9	12	15
Degree of Probability	Possible	2	4	6	8	10
	Unlikely	1	2	3	4	5

Risk criteria	
Level	Measure
High Risk	Unacceptable, and action **must** be taken immediately.
Medium Risk	Need to **implement** measures to reduce risk.
Low Risk	Acceptable, but measures should be taken where **feasible**

Definition of probability for the project		
Level	**Degree of probability**	**Frequency**
1	Unlikely	Happens very rarely
2	Possible	Happens rarely
3	Likely	Happens sometimes
4	Almost certain	Happens often

Definition of consequence for the project		
Level	**Degree of consequence**	**Outcome**
1	Insignificant	Project continues as normal.
2	Minor	Project becomes delayed slightly, but minimal effects on the end results.
3	Significant	Project stagnates, measures required.
4	Major	Project stops, critical measures required.
5	Critical	Project falls apart. All measures must be put in place to continue.

Figure 4.4: Risk Matrix [10]

5 Configurations

In this chapter we will shed light on the path we took, the decisions we made, and the progression of our project, with a primary focus on the architectures that underpinned our work.

Before proceeding to suggest potential architectures to our client, our initial task involved the selection of appropriate hardware and software, which we then integrated into a range of architectural designs and presented to the client, allowing them to further specify which configurations were the most interesting to their intended usage. This section aims to give an overview of the breadth of options we considered before narrowing down our focus.

After thorough discussions and iterations with our client, we arrived at a consensus on the architectures that held the most potential for our client's interests. This agreement marked a significant turning point in our project, as it enabled us to channel our efforts in a focused direction.

After deciding which architectures we would proceed with, we will outline and detail the decisions that went into each architectural design. This will give a clear picture of how and why we selected the hardware and software that comprises our system.

5.1 Hardware Selection Process

In the initial phase of the project, we presented multiple architectural designs to our client from which they chose three different versions, each with slight modifications. KDA expressed interest in an in-depth exploration of sensor readings conducted at the edge, prompting our decision to design a more decentralized architecture. This setup separates functionality between the edge configuration while maintaining a central configuration that remains consistent across all versions.

Earlier versions of the Local Hawk drone could fly independently using GPS, an internal measurement unit, and a barometer. These features were enabled by a specific accessory for the Raspberry Pi 4, called a NAVIO2 hat, which provided all the necessary sensors. The drone's flight was controlled by Ardupilot, a software suite running directly on the Raspberry Pi 4. However, our client wanted to avoid overloading the Raspberry Pi 4's processor with the heavy computational tasks associated with object detection. As a result, we were asked to focus on solutions where image processing is carried out on separate, dedicated hardware. Early in the project, we identified and chose various hardware parts that would be appropriate for our drone designs.

5.1.1 nVidia Jetson Nano

The Jetson Nano is a single-board computer with a built-in NVIDIA GPU with 128 CUDA cores. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model that utilizes NVIDIA's graphics processing units (GPUs) for general purpose computing. By allowing developers to leverage the power of GPU beyond graphics, CUDA has vastly increased the efficiency and speed of computationally intensive applications. It provides a suite of software tools and libraries that allow developers to create software that can perform complex computational tasks on GPUs, thus significantly improving performance for applications involving mathematical and scientific computations, image processing, and more.

We chose the Jetson because the GPU can be leveraged for high-performance deep learning-based object detection. Performing image processing on a GPU has the added benefit of freeing up the CPU for other tasks.[11]

Figure 5.1: Jetson nano [12]

5.1.2 Coral Edge TPU

Google's Coral Edge Tensor Processing Unit (TPU) is a type of custom-built circuit known as an "application-specific integrated circuit" or ASIC. This TPU is specifically designed for deep learning inference, which involves making predictions from new data using a trained machine learning model. Similar to the GPU embedded in the Jetson Nano, this hardware is believed to execute object detection based on deep learning more rapidly than a CPU in a typical single-board computer. The Coral Edge TPU is a USB device, meaning it needs a computer to function.

Our interest in the Coral Edge was driven by its light weight and compact size. Coupled with a Raspberry Pi single-board computer, it would create a system that is cost-effective and light, ideal for our project requirements. We were keen to explore the performance potential of this hardware combination.

Figure 5.2: Coral USB accelerator [13]

5.1.3 Raspberry Pi 4B & Zero 2

The Raspberry Pi 4B and Raspberry Pi Zero 2 were chosen as integral components for the lightweight drone system due to their blend of power, flexibility, and cost-effectiveness.

The Raspberry Pi 4B's quad-core processor offers impressive computational power in a compact package, making it capable of handling complex tasks like navigation algorithms. On the other hand, while the Raspberry Pi Zero 2 is less powerful, it remains a competent device for managing lighter tasks within the system, making the two boards a complimentary pair.

An important consideration for drone applications is the weight and size of components. Both the Raspberry Pi 4B and Pi Zero 2 excel in this regard with their lightweight and compact form factors that do not compromise the flight time or maneuverability of the drone.

Cost-effectiveness is another compelling factor that makes these boards an attractive choice. They are significantly more affordable than other single-board computers of similar capabilities, fitting well within projects constrained by budget without sacrificing functionality.

Raspberry Pi's flexibility is also notable due to the broad range of available peripherals, such as cameras and sensors. These can be seamlessly integrated into the system, providing great versatility when constructing the drone setup. Additionally, the Raspberry Pi benefits from extensive software support and a large, active community. With the ability to run various Linux distributions, software development and troubleshooting become much easier tasks.

Finally, the Raspberry Pi offers scalability for future expansions or modifications. With numerous Raspberry Pi models available, it is feasible to scale the system up or down according to the changing requirements. For instance, upgrading to a more powerful Raspberry Pi model to accommodate increased computational demand or additional features can be done without substantially altering the system's architecture.

All these reasons underscore why the Raspberry Pi 4B and Raspberry Pi Zero 2 are optimal choices for a lightweight, efficient, and robust drone system.

5.1.4 Pi Camera modules

The Raspberry Pi Camera Module v2 and v3 were chosen as key components for our lightweight drone system due to their high-resolution capabilities, light weight, compatibility, and affordability. In addition to this, several group members already owned the version 2 of the camera, and this was available on the Jetson Nano.

The Raspberry Pi Camera Module v2 is equipped with an 8-megapixel sensor, capable of capturing high-resolution images and video, which is vital for effective image processing and object detection tasks. It provides impressive image quality, ensuring the data collected by the drone is clear and detailed.

On the other hand, the Raspberry Pi Camera Module v3 offers an even higher resolution with its 11.9-megapixel Sony IMX708 sensor. This module delivers crisp, high-quality images and better low-light performance, crucial for varied and unpredictable flight environments.

Both camera modules are exceptionally lightweight and compact, which is an essential aspect for drone-based applications. By using these camera modules, we avoid adding substantial weight to the drone, thereby maintaining optimal flight time and maneuverability.

In addition, both camera modules are fully compatible with the Raspberry Pi ecosystem, which assures seamless integration into our drone setup. The ability to directly connect these modules to the CSI camera port on Raspberry Pi boards enables efficient data transmission and simplifies the overall system design.

Lastly, similar to the Raspberry Pi boards, these camera modules are cost-effective. They offer high-quality imaging capabilities at a fraction of the cost of many other comparable camera modules. This affordability makes them an ideal choice for projects operating on tighter budgets.

In summary, the Raspberry Pi Camera Module v2 and v3 provide the perfect blend of performance, compatibility, and affordability for our drone system. They offer high-quality visual data, which is essential for tasks such as object detection and navigation.[14][15][16]

5.2 Operating System and System Software Decision Process

In the pursuit of rigorous research and robust information, the significance of a controlled environment and reproducible results cannot be overstated. Absent these, the applicability and value of our findings risk being limited, particularly under rigorous scrutiny. A crucial component of establishing this controlled environment involves ensuring uniformity in our software across all configurations to the greatest feasible extent. This is to prevent the introduction of performance artifacts that could significantly impact performance, such as inconsistencies in software libraries or versions. In figure 5.3 you can see how we envisioned our platform to function.

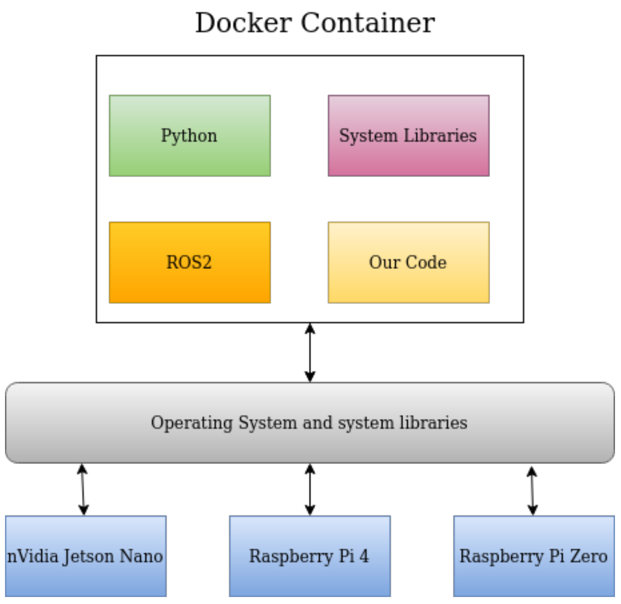


Figure 5.3: Proposed Operating System Configuration

In order to efficiently run the necessary software, we required a system configuration with robust hardware support. This system had to be compatible with a variety of Raspberry Pi hardware and the nVidia Jetson Nano. We decided to utilize the Python interpreted language for rapid prototyping. For the middleware and communication layer, we chose ROS2.

ROS2 Introduction and Rationale

Our first four proposals outlined in Section 5.10 each offer a certain degree of distribution. Distributing data processing across multiple hardware units necessitates machine-to-machine communication, a task for which the client specifically requested the use of

ROS2. ROS2 is a comprehensive software suite designed for robotics, with its core feature being inter-process communication. By creating and building our software as ROS2 nodes, we can avoid dealing with networking complexities, as ROS2 abstracts them. In ROS2, nodes constitute a graph where edges represent topics. Nodes can publish and subscribe to these topics.

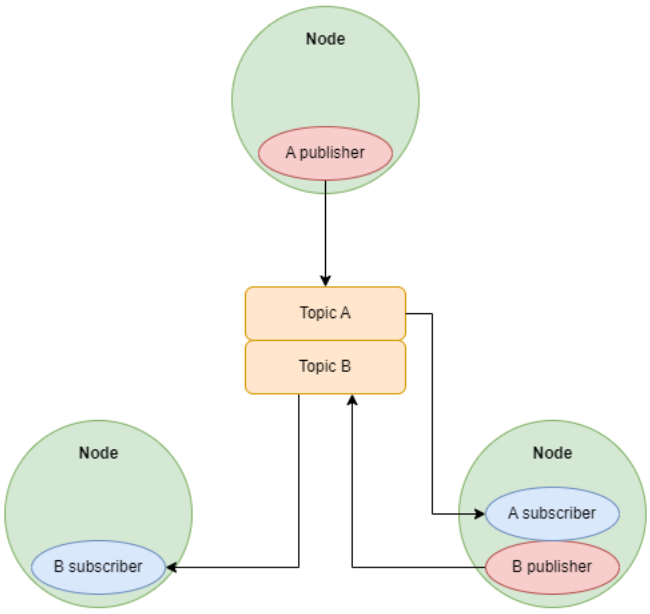


Figure 5.4: Example ROS2 graph

Figure 5.4 presents a potential configuration of a ROS2 graph featuring three nodes. Beyond managing low-level communication between hardware units, integrating ROS2 contributes to high modularity, fostering the possibility for significant scalability. Upon reviewing our initial configurations, our client voiced interest in incorporating an additional single-board computer to further distribute the workload. This hardware expansion could also be implemented at subsequent stages without necessitating code rewrites. Essentially, nodes can be easily transferred from one hardware unit to another.

Our objective was to make an informed decision, ensuring we did not default to conventional software without considering potential alternatives that might offer performance benefits or simplified software deployment. Both the Raspberry Pi variants and the Jetson Nano are compatible with the Linux operating system. Linux, a free and open-source operating system, is available in several versions, commonly referred to as distributions or "distros". These distributions can greatly vary in terms of software availability and hardware support, contingent on the installed version. Consequently, we aimed to investigate the potential differences between these versions to determine which one best met our needs.

However, with over 600 distinct Linux distributions available, it was not feasible to evaluate all of them within our limited timeframe. To streamline our focus, we decided to sample the most commonly used distributions that also maintain an enterprise presence. Ultimately, we chose to evaluate Raspberry Pi OS, the version bundled with Raspberry Pi variants, along with Fedora Linux, openSUSE Linux, and Ubuntu.[17]

After finalizing our decision, we turned our attention to designing our architecture, considering how the software would function on different hardware configurations and how we could ensure a uniform configuration across all systems. To this end, we explored the feasibility of running the software within what is commonly known as a container. A container is a compact, standalone, and executable software package that encompasses everything required to execute a piece of software, including the code, runtime, system tools, system libraries, and settings. Containers are isolated from each other and bundle their own software, libraries, and configuration files, communicating through well-defined channels. Implementing this technology would guarantee identical deployment configurations and could help mitigate any potential performance artifacts. The goal of this configuration was to encapsulate the necessary environment and libraries for ROS2 within these containers, as illustrated in Figure 5.5.

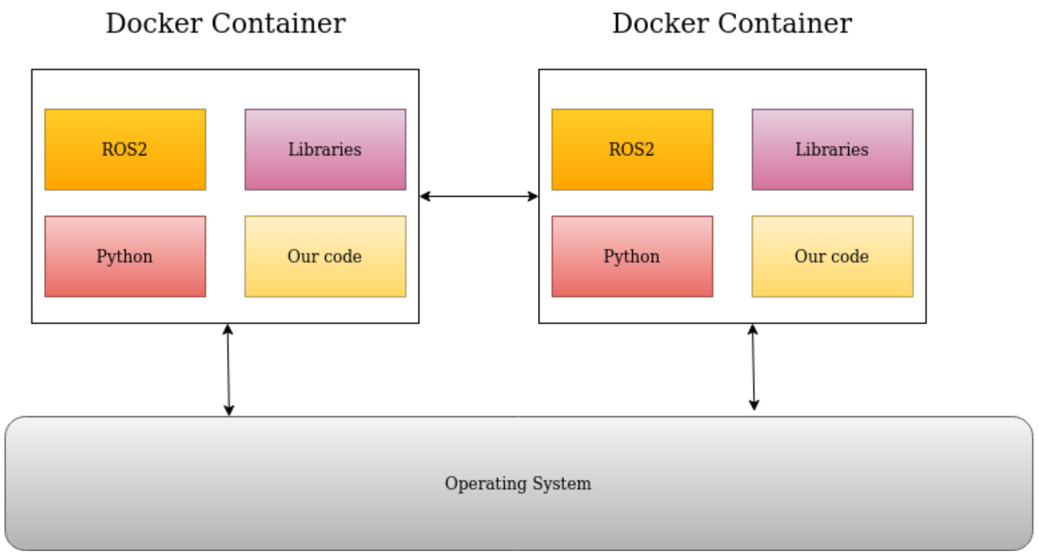


Figure 5.5: Docker architecture

Upon deciding which configurations to investigate, we initiated our examination by methodically installing the different distributions on physical hardware and assessing their functional capabilities and any shortcomings. As illustrated in Figure 5.6, you can observe the different outcomes and ascertain which functions were successful and which were not.

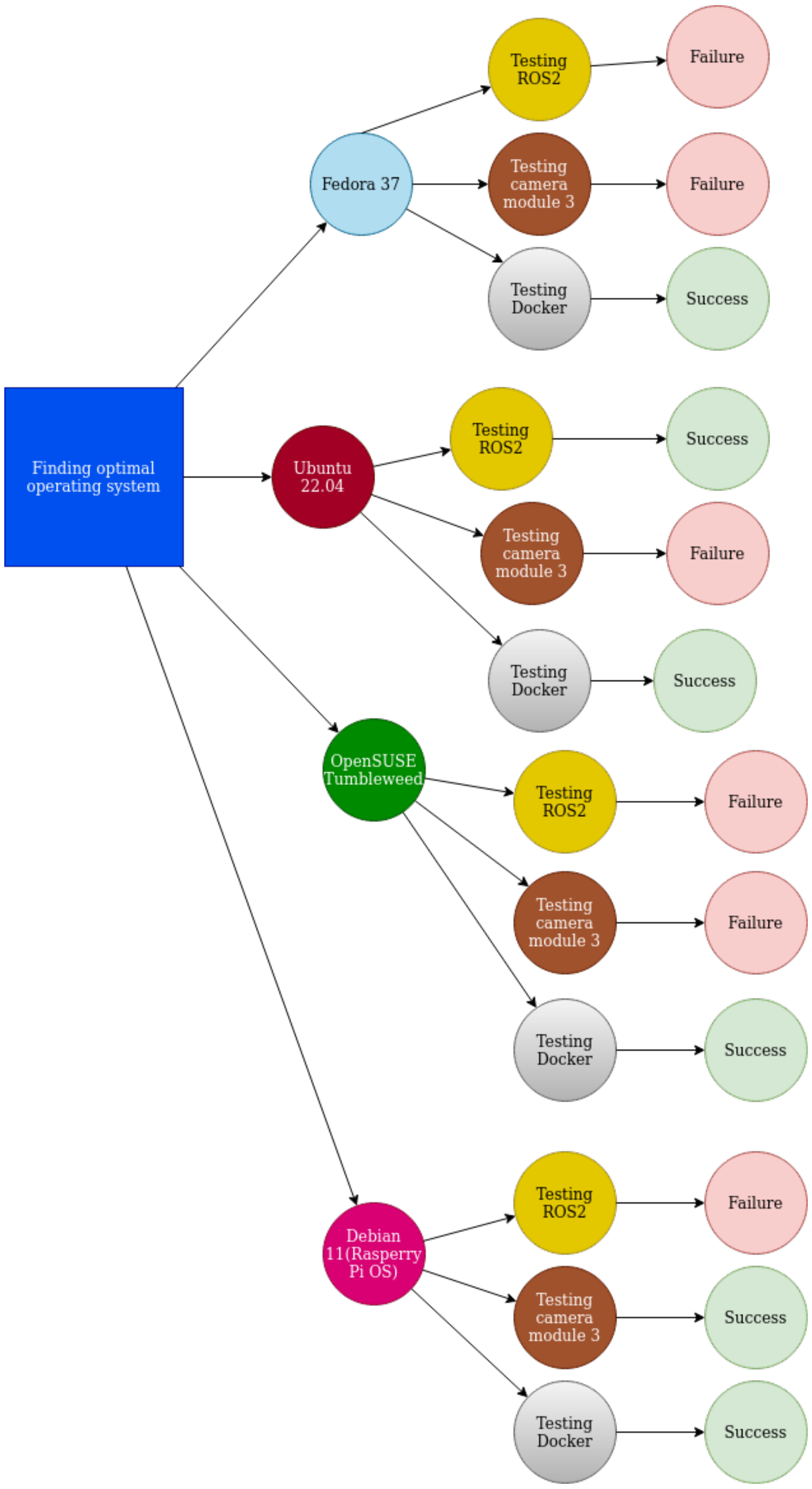


Figure 5.6: Decision Process OS

As indicated in figure 5.6, there were no systems that met all the requirements we had for our software platform, this caused a need to do further research in order to be able to decide which platform we would use going forward.

We further refined our choices to Ubuntu and Debian, as these platforms had only a single missing dependency. However, we needed to ascertain which option would offer the best outcomes relative to the efforts required to rectify each system.

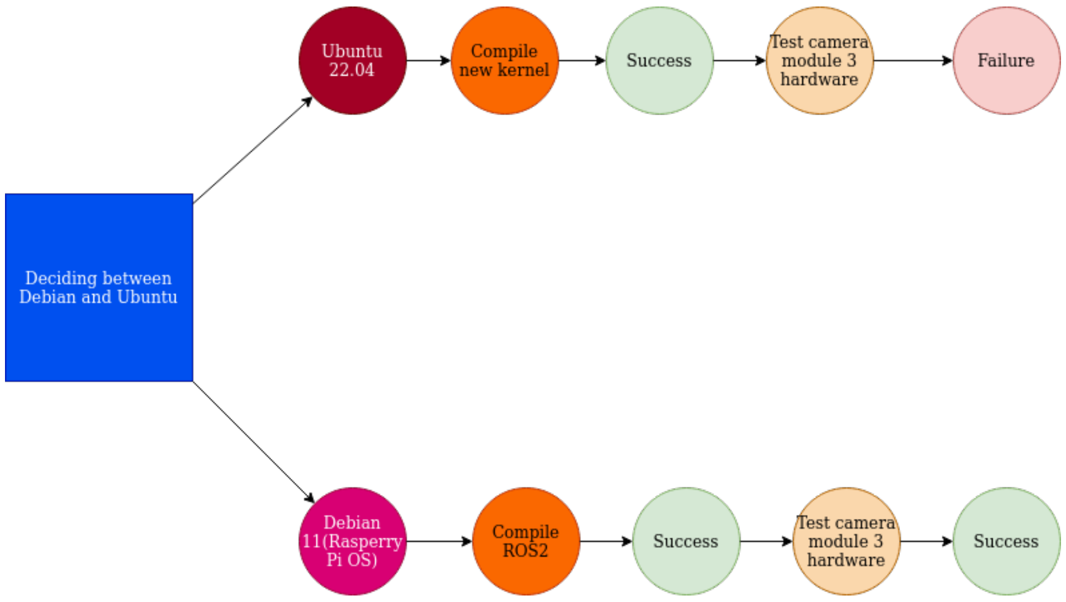

Deciding between Debian and Ubuntu
Ubuntu 22.04
Compile new kernel
Success
Test camera module 3 hardware
Failure
Debian 11(Rasperry Pi OS)
Compile ROS2
Success
Test camera module 3 hardware
Success

Figure 5.7: Decision Process Distribution

Debian faced the disadvantage of having difficulties with the ROS2 operation, which entails a relatively complex and comprehensive build-system. The documentation can often be outdated due to the fast-paced advancement of the ROS2 project at the current stage. Conversely, Ubuntu's challenge lay in the lack of driver software for the Camera Module Version 3.

Given our limited experience with the ROS2 build system and the scant documentation on manual software building and installation, we decided to attempt installing the existing Debian driver in an Ubuntu environment. This would enable us to operate ROS2 with Camera Version 3.

In the Linux operating system, drivers are typically formulated as modules and integrated with the operating system kernel. This kernel is the software component within an operating system that manages hardware resources. Within the Linux system, it is feasible to specifically compile this component for the running system, without interfering with the other system software components.

Our next step was to clone the git repository of the Raspberry Pi Foundation, given that the operating system bundled with these single-board computers (SBCs) supports that particular camera. Once completed, we managed to compile the Raspberry Pi OS kernel for our active version of Ubuntu.

After finalizing the installation and booting our new kernel, the hardware was detected, but there was no video output. At this juncture, we believed that we could get the hardware to function under Ubuntu, considering this as our best immediate course of action.

We proceeded with various tests under different configurations and compiled different versions of the Linux kernel, including an unmodified version directly from `https://kernel.org`. When none of these strategies produced the desired results, we were compelled to reassess our approach.

Our focus then transitioned towards an attempt at compiling ROS2 for Debian. We delved into the documentation to compile the software on our system. However, this task proved challenging, and we faced several hurdles while attempting to compile the software. As a temporary solution, we found someone who had compiled a Debian package on GitHub, albeit from an older build. We could not verify its applicability to our requirements, thus leaving us with necessary steps to continue deploying ROS2 while we concurrently operated an unsecured ROS2 installation, aiming to expedite development.

Following an extended period of continuous effort, we managed to compile ROS2 under a Debian system. This development allowed us access to Camera Module 3 and ROS2 on the same computer. The subsequent step involved deploying this to a Docker image, facilitating environment control and ensuring a consistent platform for our ROS2 source code.

However, an unfortunate consequence of compiling software is its substantial size growth. Post-building our Docker system, its size ballooned to 14GB, which was unwieldy for deployment and challenging to run on the Raspberry Pi Zero. Unfortunately, we were unable to rectify this issue in time for our project delivery. Although it does operate on the Raspberry Pi Zero, deployment is slow and can potentially create problems if the Zero exhausts memory while pulling the Docker image.

5.2.1 Raspberry Pi 4 and Zero 2

After reviewing our results from the selection process, we arrived at a proposed solution for our configurations based on the Raspberry Pi hardware and the Raspberry Pi OS. This solution mirrored the one depicted in figure 5.5, thereby aligning closely with our initial research and concept.

5.2.2 Jetson Nano System Software Configuration

Upon finalizing the configuration for the Raspberry Pi, we shifted our focus to the Jetson Nano. Our initial plan was to have identical configurations across all hardware. However, the proprietary nature of Nvidia's hardware precluded the support of the Raspberry Pi OS. This necessitated exploratory research on compatible software for the Jetson Nano, which would align closely with the Raspberry Pi configurations. We direct your attention to figure 5.8, depicting the versions we attempted to install and set up on the hardware.

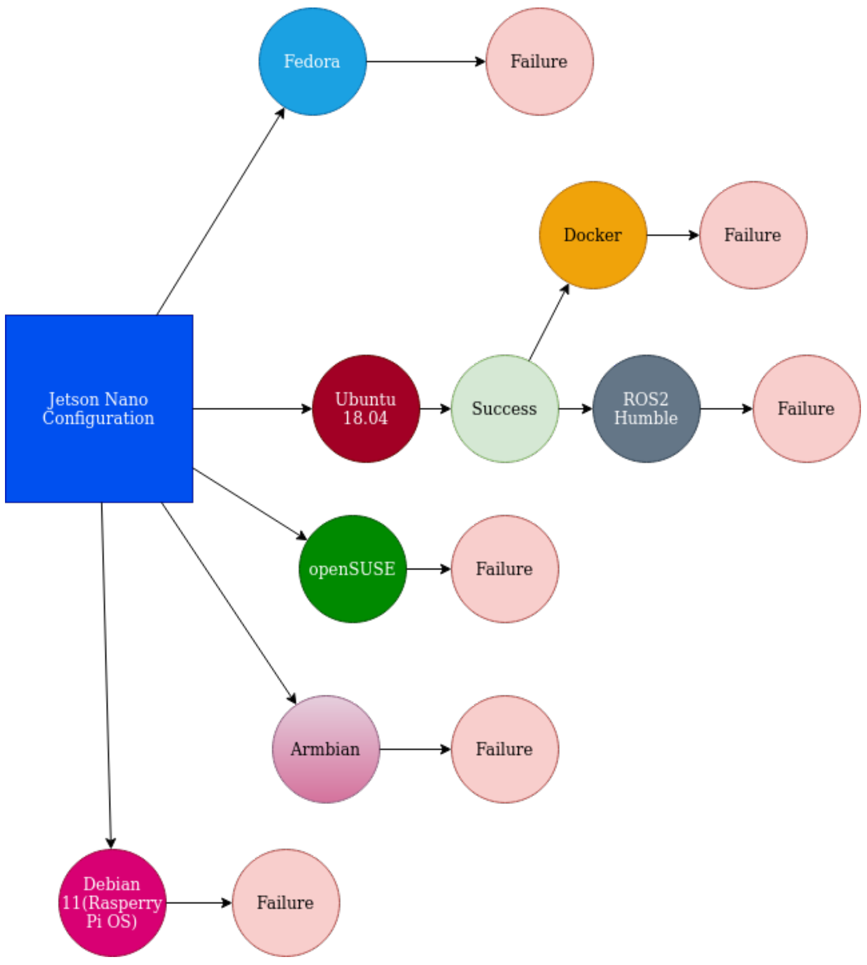


Figure 5.8: Jetson Nano Decision Map

Following a rigorous testing phase where we attempted to run several different distributions on the Nano, we concluded that this approach was not an efficient use of our time. We chose instead to focus on executing the docker architecture on the bundled Ubuntu version running on the Jetson Nano. The supported version was released in April 2018. Despite its age, we believed that since Docker was available for it, we could adjust our container setup to secure a controlled environment to the extent possible.

Docker Environment Jetson Nano

When considering Docker on the Nano, we decided to implement the same configuration we had functioning on the other systems. However, adapting this to the hardware and software on the Nano proved challenging. Nvidia distributes the necessary software for the Nano in packages referred to as ”Jetpacks”. These packages are designed for use on specific versions of Linux that they ship with Jetson Nano; our version of the Jetson

Nano was supported on Ubuntu 18.04, a five-year-old distribution. This presented challenges due to the rapid pace of change in many of the frameworks and Python libraries we wanted to use. Similar factors were also present in terms of features and performance improvements in the operating system itself.

Creating a Docker environment where we pass the GPU and every required library into the container in a functional manner proved to be a complex task. It demanded more expertise and time than we could provide during our project. Ultimately, the challenges posed by the Jetson Nano forced us to reconsider our architectural idea.

The final result for our Jetson Nano was that we utilized the system software bundled with the hardware, specifically Ubuntu 18.04. Owing to issues with Docker, we ran the software on the bare metal, as illustrated in figure 5.9. Moreover, unlike the Raspberry Pi configurations running ROS2 Humble, the Nano operates ROS2 Dashing. Despite these differences, the ROS2 architecture remains identical to the other configurations.

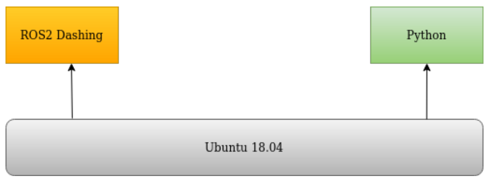


Figure 5.9: End Result Jetson Nano Architecture

5.2.3 32-bit vs. 64-bit

As Raspberry Pi OS 64-bit beats 32-bit in almost every metric regarding computation time [18] we decided to omit 32-bit based operating systems from this project.

5.3 Proposed architectures

5.3.1 Initial Proposal

Before we could start implementing our specific designs we needed to get the go-ahead from our client. We made a design sketch with five different configurations and presented these in order to reach an agreement on which areas we should focus on.

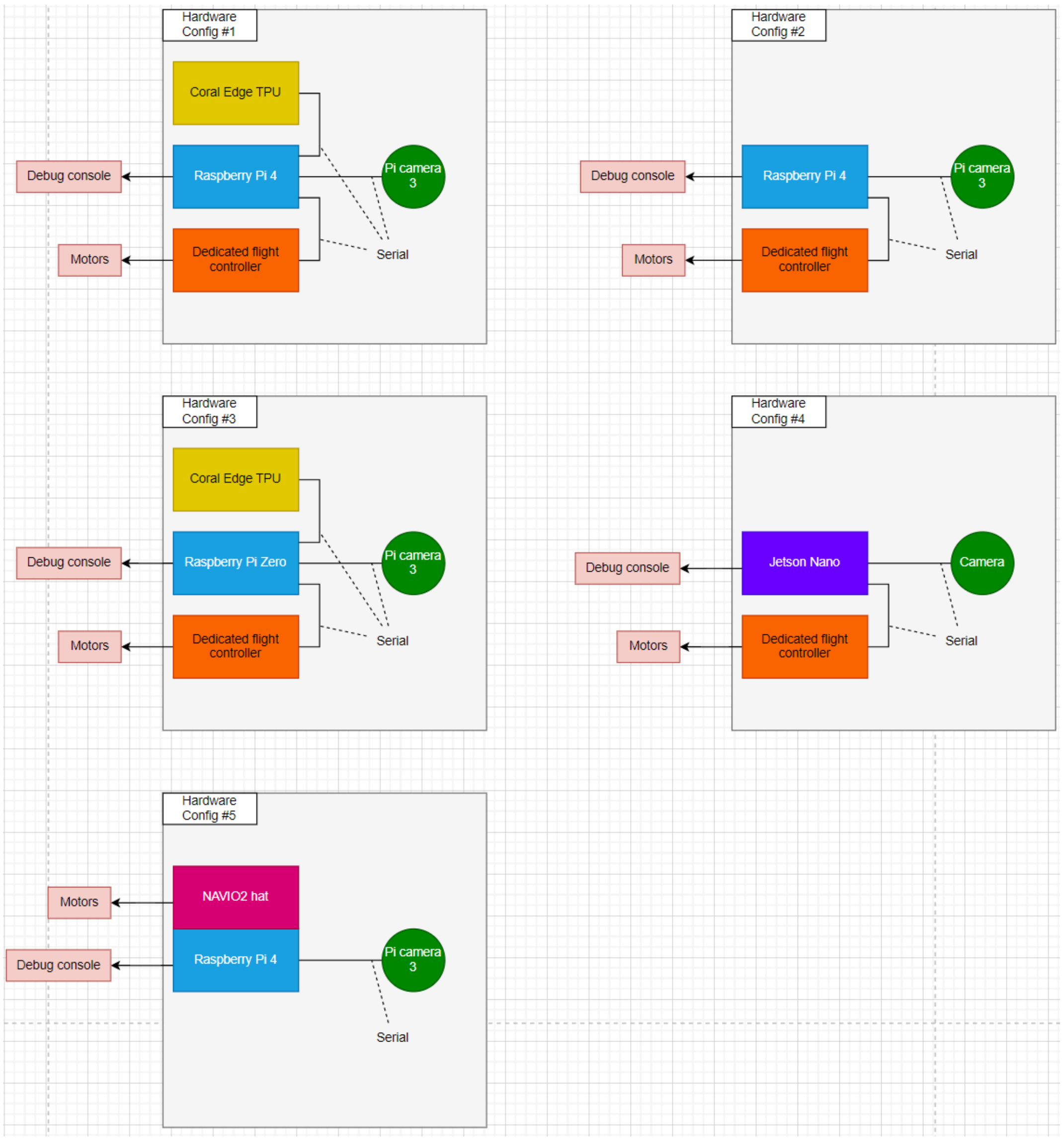


Figure 5.10: Initial architecture design

From the figure 5.10 in this figure, you can see the five proposals which we presented to the client. These proposals were based on a variety of hardware configurations we had briefly discussed during meetings and the ones that we thought were the most likely to achieve our client's goals and deliver results that would show that this idea was viable.

5.3.2 Second proposal

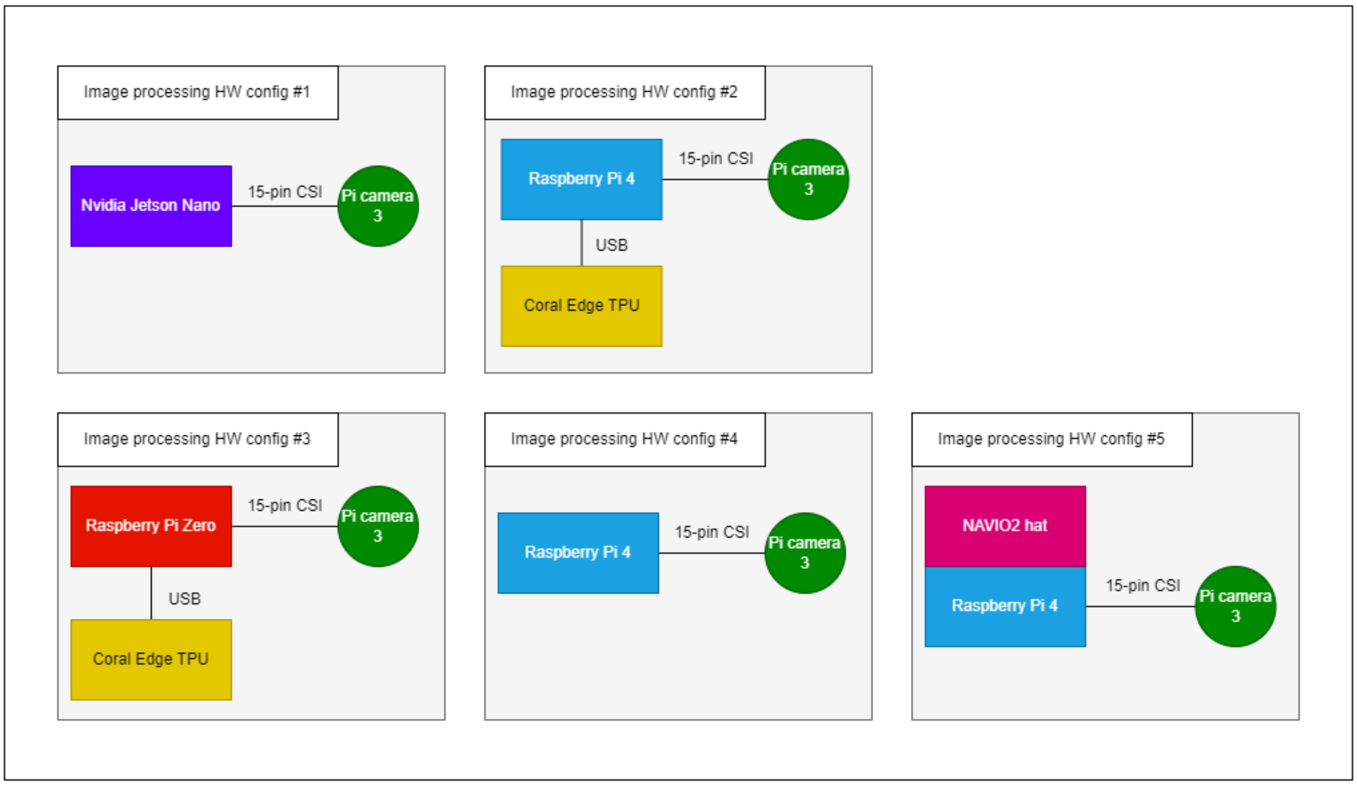


Figure 5.11: Image processing modules

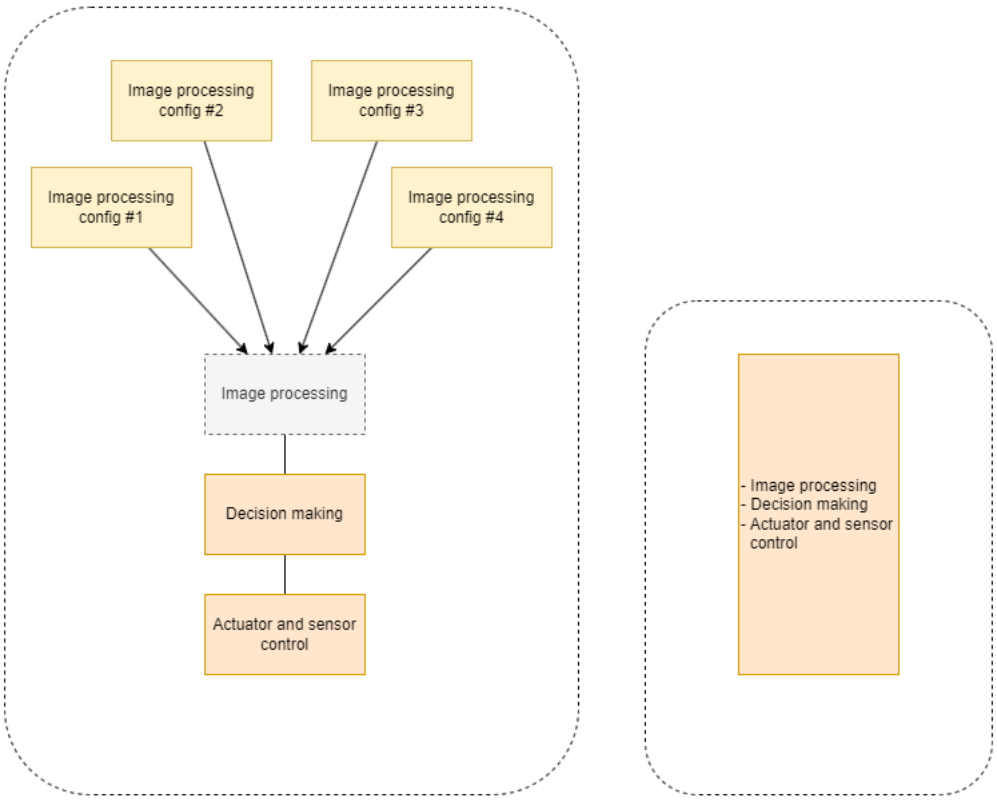


Image processing config #2
Image processing config #3
Image processing config #1
Image processing config #4
Image processing
Decision making
Actuator and sensor control
- Image processing
- Decision making
- Actuator and sensor control

Figure 5.12: Four distributed and one centralized architecture

Figures 5.11 and 5.12 describe the second iteration of our hardware configuration designs. Our image processing hardware is now drawn as interchangeable modules, which plug into the rest of the drone. The “Decision making” module in 5.12 represents a new single board computer, and the “Actuator and sensor control” module is the flight controller. After further discussing our proposals with the client we agreed to drop configuration 5 due to time constraints. Discarding the centralized configuration gave us more time to explore image processing software, which at this point seemed to be the most time-consuming task ahead.

5.3.3 Accepted architectures

Figure 5.13 illustrates the final hardware architectures which were accepted by the client.

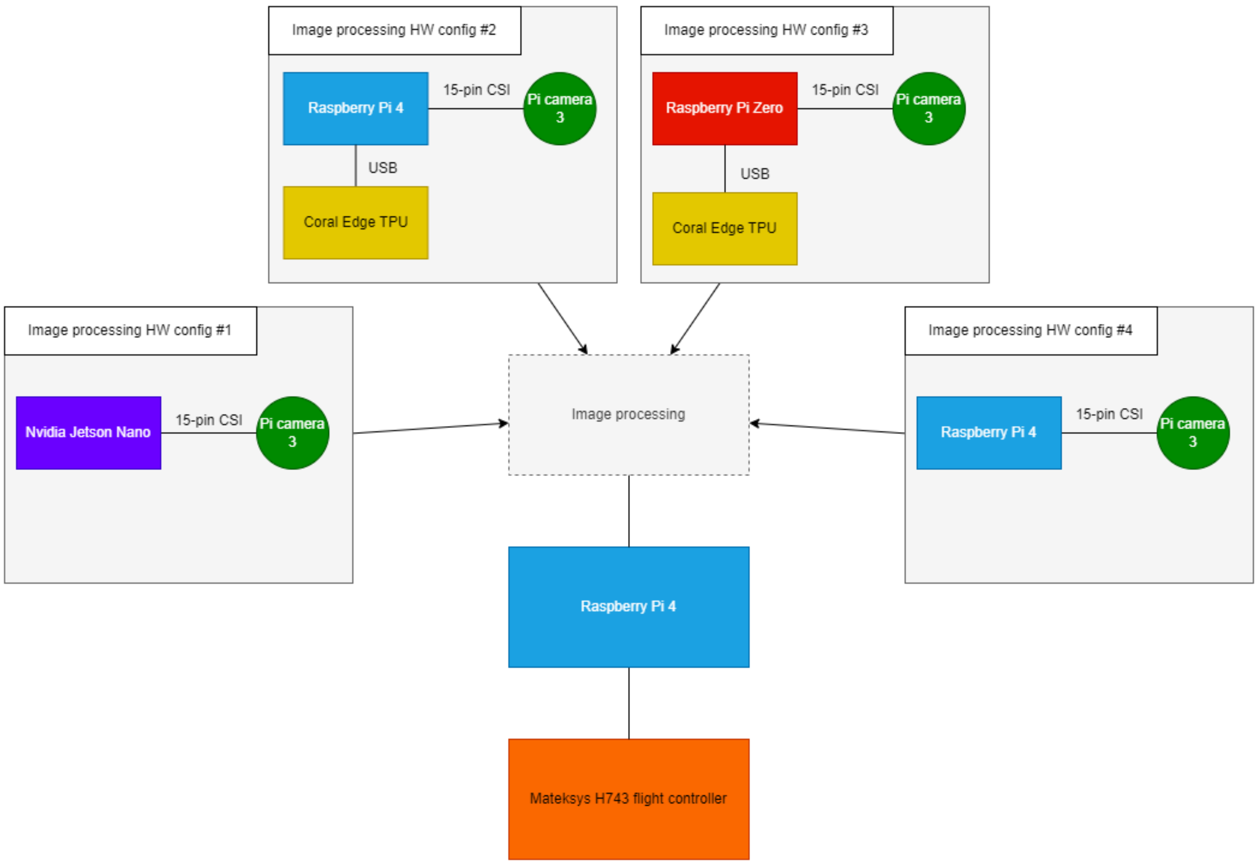


Figure 5.13: Hardware architectures accepted by client

At this stage we had designed a high-level software architecture which shows our planned ROS2 nodes and how they should interface with each other. The architecture shown in figure 5.14 is common for all our configurations. We decided that all 4 image processing modules should output the estimated distance to a detected object, and the detected object's X and Y coordinates within the image frame. As a demonstration of a use case which needs the distance and position data, we designed the software architecture with "object following" in mind. For this use case the "Follow algorithm" node in figure 5.14 would be a regulator which decides how the UAV should move to follow the detected object.

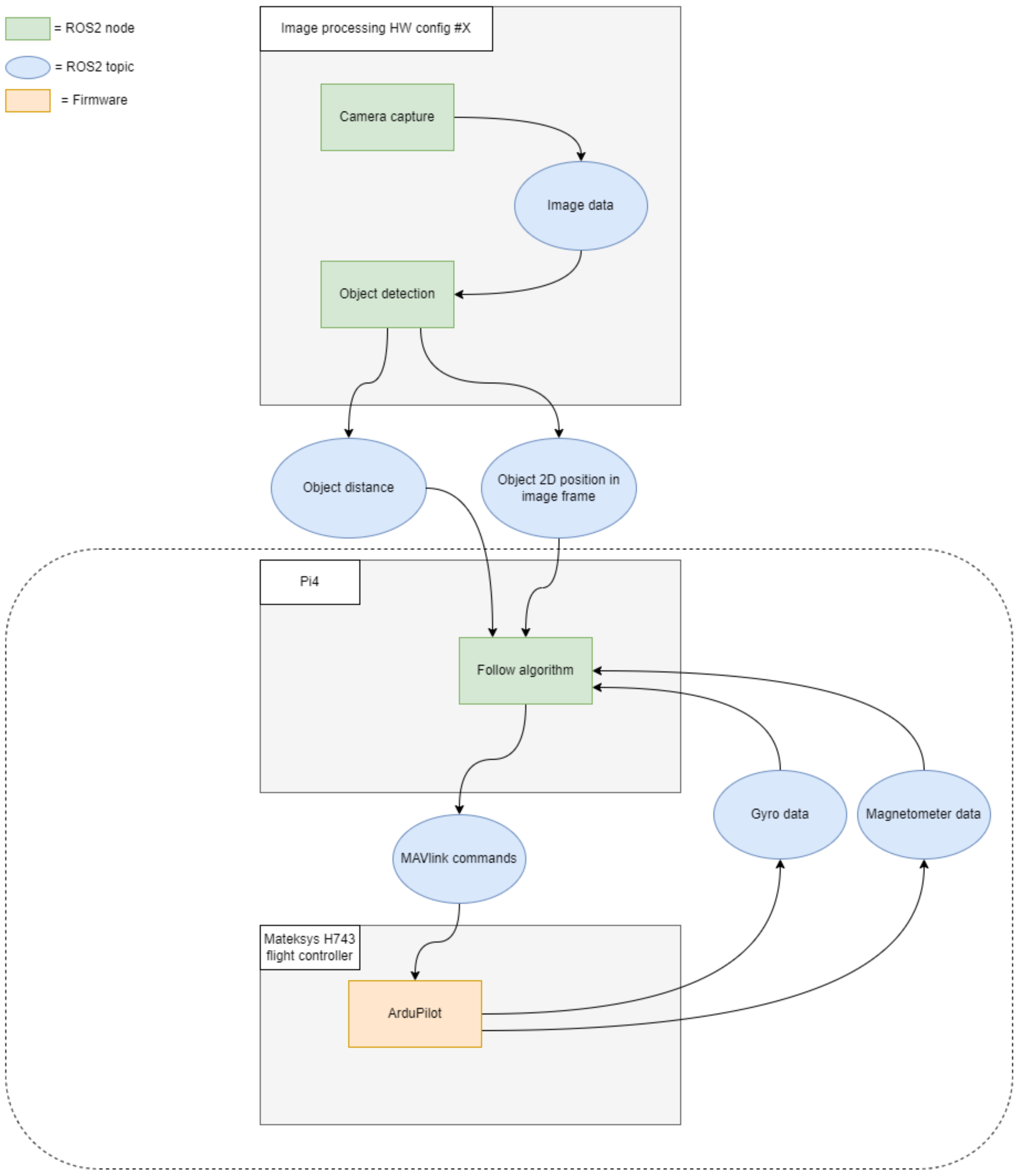


Figure 5.14: Initial software architecture design

5.4 Object detection software

One of the object detection systems we wanted to use was YOLO.

The YOLO (You Only Look Once) framework, developed by Joseph Redmon, revolutionized the field of object detection by providing a real-time object detection system. Contrary to traditional two-step object detection methods, where one step is dedicated to proposing regions where objects might exist and the second to classifying those regions, YOLO performs both tasks in one step. This makes it significantly faster and more efficient, which is particularly beneficial for applications that require real-time detection.[19]

The architecture of YOLO involves dividing the input image into a grid. Each grid cell predicts a certain number of bounding boxes and class probabilities. During the detection phase, bounding boxes with class probabilities below a certain threshold are discarded.

YOLO has gone through several iterations to improve its performance, in this study we have looked at YOLOv5 (released June 25th, 2020) [20], and the latest YOLOv8 (released January 10th, 2023)[21], both released by Ultralytics.

Given the novelty of YOLOv8, there was a lack of documentation regarding its operation on a Jetson Nano. All the available information concerning YOLO's operation on a Jetson Nano pertained to YOLOv5. Therefore, after meticulous comparison and deliberation, the decision was made to transition to the earlier iteration, YOLOv5.

Despite the commendable efforts of Ultralytics to simplify the application of YOLO, mastering the process of training on our own dataset, and effectively utilizing the diverse set of tools that accompany the framework, demanded a considerable investment of time. This learning curve demonstrates the complexity inherent in working with such advanced object detection systems. Readers are referred to the F appendix for a concise guide on how to train models with custom data. Additional information and comprehensive documentation can be found on the Ultralytics website.

Dataset

In this study we decided on a common dataset for all the object detection models to be trained on. The dataset is divided into three distinct subsets - training, validation, and testing sets - each consisting of 110, 32, and 30 images respectively. The training set is used during the learning process, the model is then validated on the validation set at the end of each epoch. The test set is a dataset never seen before by the model and is used to do a final validation of the model. Since we wanted to have a comparison between object detection and blob detection, we decided on a green tennis ball being the target for our models as this is easier to capture with blob detection.

5.5 Distance Measurement

After wishes from our client, we've included the capability to calculate the distance to a detected object in our project. This is especially useful when considering the drone's relative position to the object and is particularly relevant for use cases such as landing. Understanding this distance becomes a key factor in ensuring precise and controlled landing maneuvers.

Due to the fact that we are using one singular camera, there is a problem getting the proper depth. The method we chose to use involves the idea of similar triangles to calculate the distance between a camera and an object. This method is based on the principle that the ratio of the object's actual width to its distance from the camera matches the ratio of the object's perceived width in the captured image (in pixels) to the camera's focal length. [22]

To illustrate this concept mathematically, we introduce two similar triangles:

- The first triangle (Triangle A) is constructed with the vertices being the camera, the object, and a selected point on the camera's image plane.
- The second triangle (Triangle B) includes the camera, the image of the object as perceived by the camera, and the identical point on the image plane as used in Triangle A.

In this scenario, the parameters for the two triangles are defined as follows:

- In Triangle A, the 'height' is the distance to the object (D) and the 'base' is the actual width of the object (W).
- In Triangle B, the 'height' represents the camera's focal length (F), and the 'base' signifies the apparent width of the object in the image (P, in pixels).

Since the two triangles are similar, the ratios of the corresponding sides are equal, resulting in the equation:

$$\frac{D}{W} = \frac{F}{P} \tag{5.1}$$

Using this equation, we can derive any one of the variables given the other three. For instance, the focal length of the camera (F) can be calculated by rearranging the equation to:

$$F = \frac{P \times D}{W} \tag{5.2}$$

Furthermore, if the object's position shifts, altering the apparent width in the captured image (P'), the same equation can be adapted to compute the new distance (D'):

$$D' = \frac{W \times F}{P'} \tag{5.3}$$

5.6 Proposed Operating System and System Configuration

After a comprehensive evaluation of operating systems suitable for various hardware configurations, we concluded that the Raspberry Pi configurations would utilize the system software provided with the devices. This decision was informed by the superior hardware support and the system's ability to effectively run ROS2 as required.

Contrarily, the situation with the Jetson Nano was distinct. Given the limited time for an in-depth exploration of the software construction on this platform, we decided to adhere to the configuration supported by nVidia. The final proposed system can be seen in Figure 5.15.

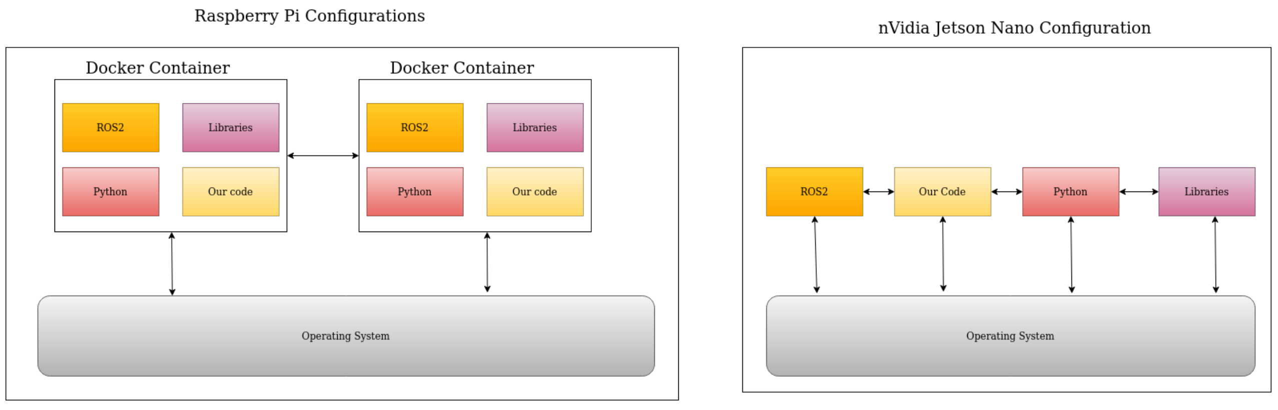


Figure 5.15: Finished Overview System Configuration

5.7 Image processing modules

In this chapter, we delve into the vital aspect of image processing configuration, a journey that has proven both challenging and enlightening. Our exploration began with an understanding of the integral role image processing plays in our project. As a technique that involves transforming or altering images using mathematical operations, image processing serves as a conduit that translates raw, visual data into a format that our systems can understand and utilize effectively.

Throughout this chapter, we discuss the myriad complexities involved in selecting, tuning, and applying appropriate image processing techniques and configurations. We recount our decision-making process, the technical considerations, the trial-and-error iterations, and the fortuitous insights that informed our eventual choices.

We acknowledge that the pathway toward effective image processing configuration is neither linear nor universally applicable. The optimal solutions often depend heavily on the specific contexts, goals, and constraints of the project at hand. However, we believe that by sharing our journey – our successes, our hurdles, and the lessons we learned along the way – we can provide valuable insights that may guide similar endeavors in the future.

Through this reflection, we aim to illuminate the depth and breadth of thought that underlies the seemingly mundane topic of image processing configuration, reinforcing its critical role in the success of our project and many others in the field. By the end of this chapter, readers will not only understand our journey toward an effective image processing configuration but also appreciate the intellectual richness that this journey entails.

5.7.1 Configuration 1, Jetson Nano

This section aims to clarify the steps we followed to make object detection work on a Jetson Nano, incorporating details of unsuccessful attempts as well.

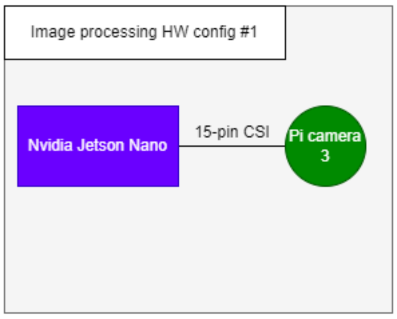


Figure 5.16: Hardware architecture, config 1

Initial setup

The NVIDIA Jetson Nano platform employs a software development kit (SDK) image known as JetPack, specifically version 4.6.1, which is the newest compatible version for the Jetson Nano device. JetPack 4.6.1 is built upon Ubuntu 18.04 (Bionic Beaver) and utilizes Python 3.6.9. The image is preconfigured with several essential developer tools, including TensorRT, cuDNN, and CUDA.

To flash the image onto the Jetson Nano, one can either use a terminal or the NVIDIA SDK Manager, which can be accessed at [23]. It is strongly recommended to employ the NVIDIA SDK Manager for this task, as it significantly simplifies the installation process for the image and any additional SDKs. However, this method necessitates that the computer used for flashing possesses the same operating system as the target image. In the present case, the computer required reformatting to Ubuntu 18.04 to ensure compatibility.

Alternatively, the SD card can be directly flashed using the terminal. The necessary image can be obtained from the NVIDIA developer site [24]. Subsequent instructions specific to your operating system can be found at [25].

Following this, the NVIDIA SDK Manager can be used to install all additional SDKs via Secure Shell (SSH). This approach upholds the user-friendliness and efficient process inherent to the SDK Manager.

Software overview

Camera interface: Gstreamer and openCV
Object detector type: Deep neural network
Machine learning framework: Trained with Pytorch, TensorRT for inference
Model type: FP32 TensorRT model (.engine)
Model 1: 640x640x3 (Yolov5n)
Model output: Bounding boxes, prediction scores, class names, number of predictions
Model size: 7MB

Config1 journey

The process of getting infernece to run on the Jetson Nano platform has been a complex undertaking, with our efforts encompassing three distinct methodologies and two different models: MobileNet-SSD and YOLO. Each subsequent three subsection will delineate the specifics of these approaches. All methodologies employs TensorRT optimization to facilitate efficient execution on the NVIDIA hardware. The final method is the one currently in operation. see figure 5.17 for a visual representation of the journey.

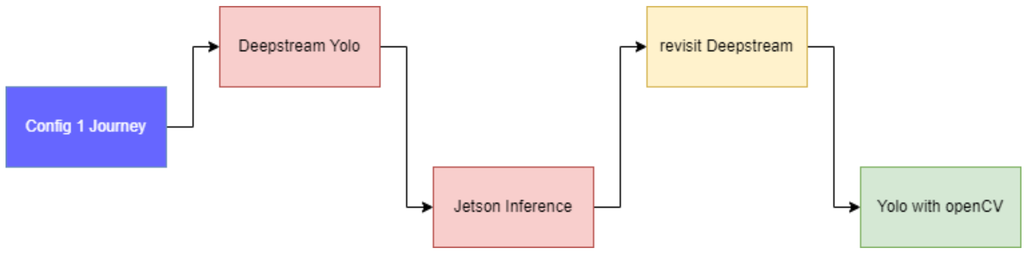


Figure 5.17: Visual representation of the journey

Yolo with Deepstream

The process of deploying YOLOv5 on the Jetson Nano platform proved to be more complex compared to the implementation of MobileNet-SSD. The YOLOv5 documentation provides a dedicated page for its application on a Jetson Nano using a Software Development Kit (SDK) from Nvidia named Deepstream. However, the tutorial [26], last updated 18 November 2022) is not as straightforward as it might appear. For the purpose of this study, the deployment of YOLOv5 on the Jetson Nano was accomplished through two distinct approaches, both of which will be thoroughly explored in the following sections.

The initial step requires the use of a PC running Ubuntu 18.04 to operate the Nvidia SDK Manager in conjunction with the Jetson Nano. Following this, the installation of several additional packages, one of which is Deepstream, is necessary. This part of the procedure is fairly linear and should already be done with the initial setup of the Jetson.

The challenge arises when attempting to install dependencies for YOLO. Since YOLOv5 necessitates Python 3.7, while the Jetson Nano only supports Python 3.6.9, after cloning the repository, all entries in the requirements.txt file were commented out. Subsequent to numerous trials and errors, the tested system runs the following versions.

Python package	Version number
gitpython	3.1.20
matplotlib	3.3.4
numpy	1.19.5
opencv-python	4.1.1
Pillow	7.1.2
psutil	
PyYAML	6.0
requests	2.18.4
scipy	1.5.4
thop	0.1.1
tqdm	4.64.1
seaborn	0.11.0
setuptools	59.6.0

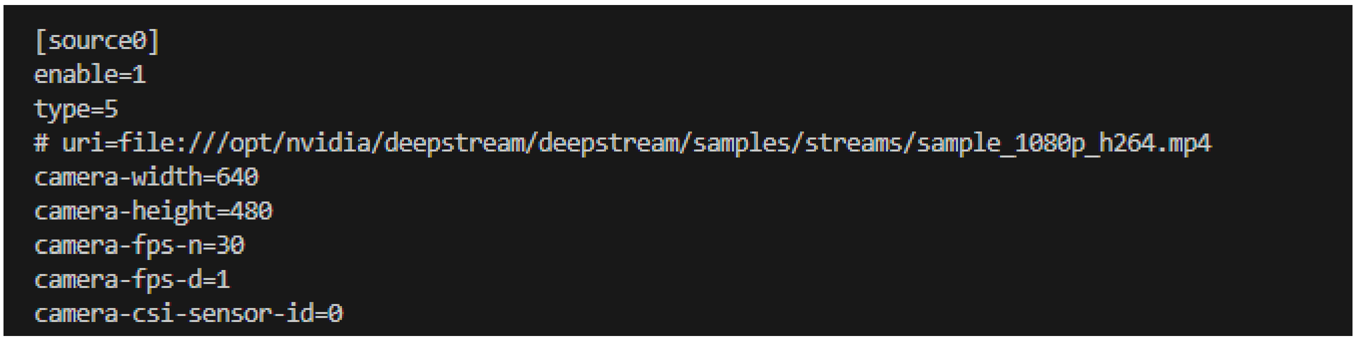
Table 5.1: pip packages used for deepstream

Continuing from this point, the subsequent steps as outlined on the GitHub repository should be followed until the DeepStream Configuration for YOLOv5 ”Step 4. Generate the cfg and wts files”. This can be accomplished with the following Python3 commands:

```
python3 gen_wts_yoloV5.py -w yolov5s.pt
python3 gen_wts_yoloV5.py -w custom.pt
```

Here, the user has the flexibility to either convert the pretrained yolov5s model or replace it with any model that has been subjected to transfer learning, as demonstrated above.

After completing the remaining steps, the user should be able to run inference on the included video. Additionally, to run inference on a Camera Serial Interface (CSI) camera, as employed in this study, the user must modify the source0 parameter in the deepstream_app_config.txt file as follows:



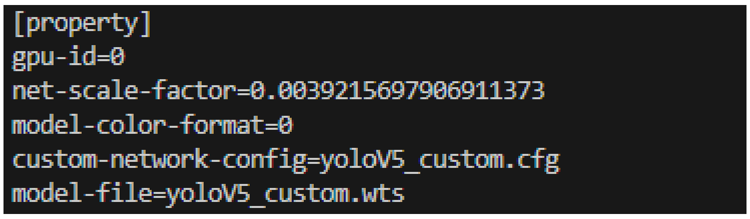
```
[source0]
enable=1
type=5
# uri=file:///opt/nvidia/deepstream/deepstream/samples/streams/sample_1080p_h264.mp4
camera-width=640
camera-height=480
camera-fps-n=30
camera-fps-d=1
camera-csi-sensor-id=0
```

Figure 5.18: Deepstream app config file, source0

DeepStream offers built-in support for CSI cameras, with type=5 indicating the usage of a CSI camera. The camera's width and height can be adjusted according to the user's needs. It should be noted, however, that the model is optimized for an input resolution of 640x640 pixels, so providing an input close to this resolution would potentially enhance performance.

It is also important to modify the config-file under primary-gie, set this to config_infer_primary_yoloV5

To use a custom model, the configuration file must be modified in line with the particular version of YOLO in use, hence necessitating amendments to the 'config_infer_primary_yoloV5.txt' file. The modifications include changes to the following parameters:



```
[property]
gpu-id=0
net-scale-factor=0.0039215697906911373
model-color-format=0
custom-network-config=yoloV5_custom.cfg
model-file=yoloV5_custom.wts
```

Figure 5.19: config_infer_primary_YoloV5.txt, custom model

In this context, the weight (wts) and configuration (cfg) files have been renamed as 'yoloV5_custom.cfg' and 'yoloV5_custom.wts', respectively, indicating their customized nature and their association with the YOLOv5 model. Now to run inference, simply use the command:

```
deepstream-app -c deepstream_app_config.txt
```

As indicated on the Ultralytics webpage [26], the potential to achieve a rate of 30 frames per second (fps) exists when deploying a Jetson Xavier NX with FP32 and the YOLOv5s model. In contrast, the Jetson Nano, possessing lesser hardware capabilities than the Xavier, managed to reach a rate of 10 FPS with the YOLOv5n (Nano) model.

However, a substantial obstacle presented itself in the form of a 400 ms latency on the inference stream. Even with thorough system resource monitoring and various attempted solutions, the latency issue could not be mitigated. The implications of this problem suggested that the procedure overtaxed the Jetson Nano's hardware, thereby making it an unsuitable choice for applications like drone operations, which require instantaneous detection capabilities.

The Deepstream approach, despite its challenges, displayed a degree of adaptability. Specifically, it allowed a broad range of easy customizability via accessible configuration files.

Mobilenet-SSD

Jetson Inference [27], a GitHub repository developed by Dustin Franklin from Nvidia, provides a well-documented tutorial complete with video walkthroughs, which is particularly beneficial for beginners in the field of computer vision working on the Jetson device. The process of getting this up and running is as simple as cloning the repo and then running the docker that is included.

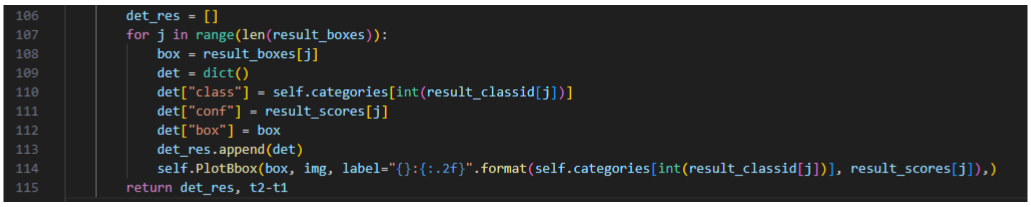
The pre-trained model demonstrated satisfactory performance at shorter distances but struggled with object detection when the objects were small or located at greater distances. To address this, transfer learning was applied to the model using our dataset, both on the Jetson device and on a separate computer equipped with a dedicated GPU. Despite training the model for over 1000 epochs, the inference failed to detect the object.

A survey of online forums confirmed that the model encountered difficulties with small objects. An attempt was made to upgrade the model to accommodate an input resolution of 512x512 instead of the standard 300x300. This was tried, and while this modification slightly improved the model's performance, it did not reach the level of the YOLO model trained earlier in the study. As a result, a decision was made to revisit YOLO.

Yolo with openCV

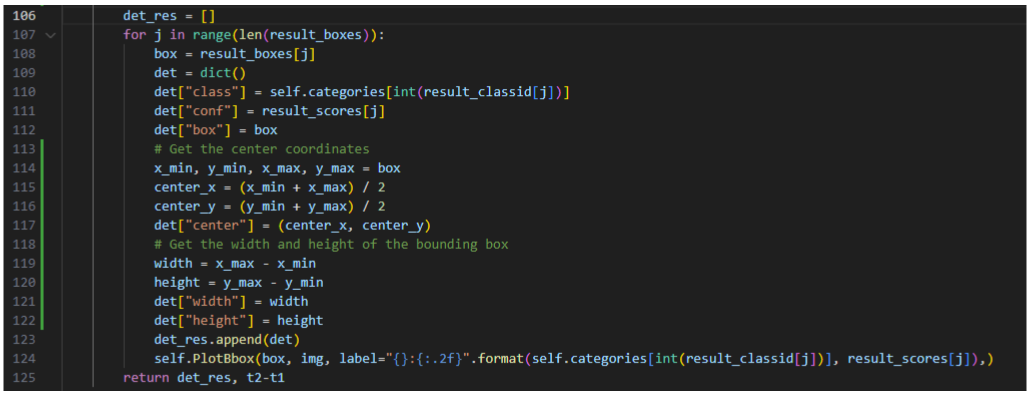
Since DeepStream did not yield satisfactory results, we searched for an alternative solution. During this exploration, we discovered newly created repository from Mailrocketsystems `https://github.com/mailrocketsystems/JetsonYolov5`. The repository applies TensorRT and OpenCV for inference. The configuration process is user-friendly, requiring merely adherence to the instructions provided in the repository's ReadMe file on GitHub.

The extraction of the right metadata from the model necessitates some modifications to the yoloDET script. Please refer to Figures 5.20 and 5.21 for the implemented changes.



```python
        det_res = []
        for j in range(len(result_boxes)):
            box = result_boxes[j]
            det = dict()
            det["class"] = self.categories[int(result_classid[j])]
            det["conf"] = result_scores[j]
            det["box"] = box
            det_res.append(det)
            self.PlotBbox(box, img, label="{}:{:.2f}".format(self.categories[int(result_classid[j])], result_scores[j]),)
        return det_res, t2-t1
```

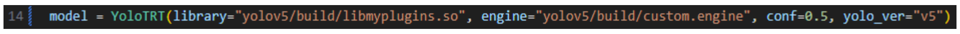
Figure 5.20: YoloDET before



```python
    det_res = []
    for j in range(len(result_boxes)):
        box = result_boxes[j]
        det = dict()
        det["class"] = self.categories[int(result_classid[j])]
        det["conf"] = result_scores[j]
        det["box"] = box
        # Get the center coordinates
        x_min, y_min, x_max, y_max = box
        center_x = (x_min + x_max) / 2
        center_y = (y_min + y_max) / 2
        det["center"] = (center_x, center_y)
        # Get the width and height of the bounding box
        width = x_max - x_min
        height = y_max - y_min
        det["width"] = width
        det["height"] = height
        det_res.append(det)
        self.PlotBbox(box, img, label="{}:{:.2f}".format(self.categories[int(result_classid[j])], result_scores[j]),)
    return det_res, t2-t1
```

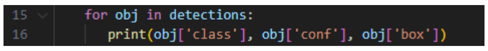
Figure 5.21: YoloDET after

Now we can simply edit the app.py script to use our model, and extract the metadata we just added in yoloDET. See figures 5.22, 5.23, 5.24



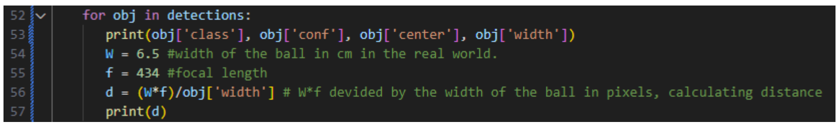
```python
model = YoloTRT(library="yolov5/build/libmyplugins.so", engine="yolov5/build/custom.engine", conf=0.5, yolo_ver="v5")
```

Figure 5.22: Selecting our model



```python
for obj in detections:
    print(obj['class'], obj['conf'], obj['box'])
```

Figure 5.23: extracting metadata original



```python
    for obj in detections:
        print(obj['class'], obj['conf'], obj['center'], obj['width'])
        W = 6.5 #width of the ball in cm in the real world.
        f = 434 #focal length
        d = (W*f)/obj['width'] # W*f devided by the width of the ball in pixels, calculating distance
        print(d)
```

Figure 5.24: extracting metadata modified

The repository enabled us to perform inference in real-time with close no none latecy, which was particularly helpful in demonstrating the capabilities of the hardware within config1.

Reflections

While the Jetson Nano boasts comparatively robust hardware, it is beginning to show signs of obsolescence, largely due to its reliance on an older version of the Jetpack SDK, which in turn is based on outdated versions of Ubuntu and Python. There have been considerable developments since Ubuntu 18.04 and Python 3.6. Nonetheless, we maintain that the Jetson Nano presents a compelling option for hobbyists seeking to delve into the realm of AI and explore its capabilities. For more advanced development pursuits, we would recommend a more contemporary model, such as the Jetson Orin Nano. This newer hardware is accompanied by support for the latest release of the Jetpack SDK, which is based on Ubuntu 20.04, and generally offers better compatibility with recent software.

The Jetson Nano was received quite late into the project, leaving us with approximately one month to set it up and get it working. With more time, and a deeper understanding of machine learning and ai, the results might have been different.

5.7.2 Configuration 2, Pi 4 w/ Coral Edge TPU

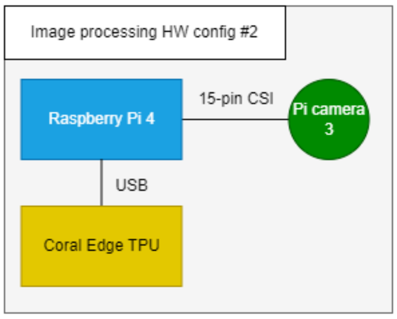


Figure 5.25: Hardware architecture, config 2

Software overview

Camera interface: picamera2 python library
Object detector type: Deep neural network
Machine learning framework: Tensorflow Lite
Model type: Tensorflow Lite model (.tflite) compiled for Coral Edge TPU
Model 1 input: 320x320x3 array (EfficientDet-lite0)
Model 1 size: 5.57MB (EfficientDet-lite0)
Model 2 input: 384x384x3 array (EfficientDet-lite1)
Model 2 size: 7.57MB (EfficientDet-lite1)
Model outputs: Bounding boxes, prediction scores, class names, number of predictions

Apt package name	Version
python3-tflite-runtime	2.5
python3-cv-bridge	1.16.2
libedgetpu1-std	16.0
python3-picamera2	0.3.9

Table 5.2: Required apt packages, config 2

Pip package name	Version
numpy	1.20

Table 5.3: Required pip packages, config 2

Video capture software setup

The configuration runs inference on image data provided by the Pi camera V2. The Picamera2 library is the Python interface to the currently supported Raspberry Pi camera stack. Picamera2 requires Raspberry Pi OS Bullseye or newer versions, and the legacy camera stack must be disabled through the raspi-config utility.

Inference software setup

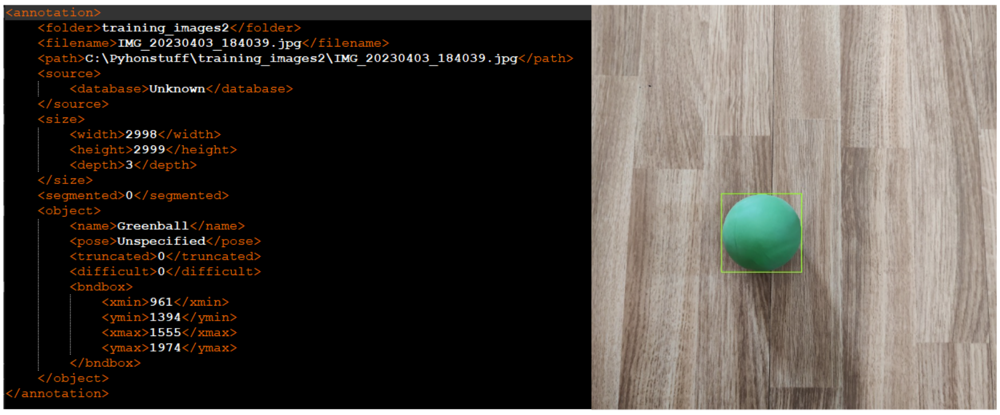
Setting up the configuration with Python is straightforward as the Coral Edge TPU is well documented by Google's Coral team. The TPU is only compatible with Tensorflow Lite models so the Tensorflow Lite Python API (Tensorflow Lite Runtime) is needed to run inference. Additionally, the Edge TPU runtime library is needed to provide an interface between Tensorflow Lite Runtime and the TPU. At the time of writing Tensorflow Lite Runtime is only compatible with Python versions 3.7 – 3.9.

Deep learning models

Using the same dataset as in configuration 1, we have trained 2 suitable models for this configuration. The models are trained with transfer learning on pre-trained neural networks. We used Tensorflow Lite Model Maker for training. This library simplifies transfer learning to a degree that allows it to be done with minimal knowledge of the inner workings of Tensorflow and neural networks. Using Tensorflow Lite Model Maker had the downside of limiting our options for pre-trained models. The library only supports 5 different object detection models, which are all from the EfficientDet family released in late 2019. We selected EfficientDet-lite0 and EfficientDet-lite1 for this configuration. These models are optimized for Tensorflow Lite and edge deployment [28] [29]. EfficientDet-lite0 is the lightest model with an input size of 320x320x3. EfficientDet-lite1 is slightly heavier with an input size of 384x384x3. An attempt was made to compile a custom Yolo v5 model for Tensorflow Lite and the Edge TPU for a better comparison against configuration 1. The Yolo model was discarded as we were unable to have it utilize the TPU. The Coral Edge TPU puts multiple constraints on the model choice [30], one of them being that not all commonly used neural network operations are supported. Deducing why our yolo model failed to run on the TPU was beyond the scope of this study, as it would likely require deep understanding of neural networks.

Training the models

We used transfer learning with the Tensorflow Lite Model Maker library to train models for configuration 2. We explored the possibility of using Keras as well, which is a high-level deep learning API with Tensorflow integration. Using Keras would enable us to use any available object detection model for transfer learning. While Model Maker only supports 5 object detectors. Model Maker was selected because of its low complexity. Due to our time constraints, we chose not to dedicate time to learning Keras. Before training we prepared our dataset by annotating our images. The dataset used is the same as the one used in configuration 1. The dataset is annotated using Labelimg, which is a GUI application for manually labeling objects. Labelimg generates an xml file for each image in the Pascal VOC annotation format. The annotation contains info on the bounding box location and class name for labeled objects in an image. Figure 5.26 shows the content of an xml file, and its corresponding image with the bounding box drawn around our labeled object.



```
<annotation>
    <folder>training_images2</folder>
    <filename>IMG_20230403_184039.jpg</filename>
    <path>C:\Pyhonstuff\training_images2\IMG_20230403_184039.jpg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>2998</width>
        <height>2999</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>Greenball</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>961</xmin>
            <ymin>1394</ymin>
            <xmax>1555</xmax>
            <ymax>1974</ymax>
        </bndbox>
    </object>
</annotation>
```

Figure 5.26: Pascal VOC .xml contents (left), corresponding image (right)

The annotated dataset is split into three categories: Training data, validation data and test data. Training data is the set used for training our model. The validation data is used to evaluate the model during the training process. Test data is used to evaluate the final model after training is complete. The final evaluation gives us the model's precision statistics. The simplicity of Tensorflow Lite Model Maker is best described by the few lines of code required to train a model. Our training script for transfer learning with the EfficientDet Lite1 model can be found in H. When the Tensorflow Lite model is trained it needs to be compiled for the TPU. The compilation is done using the Edge TPU Compiler tool which can run on any Linux machine [31]

High level data flow

Figure 5.27 shows the input and outputs of the Tensorflow Lite interpreter object which contains all the methods used to handle inference. The picamera2 python library provides an interface to the Raspberry Pi camera drivers and allows us to capture image frames as multidimensional arrays. The tflite_interpreter class provides methods for setting up and running inference with our trained object detection model. The model takes the image array as input and outputs the coordinates of bounding boxes around any detected object as well as a prediction score per detected object. The score describes how likely it is that the detection is a true positive. The full source code for configuration 2 can be found in appendix G

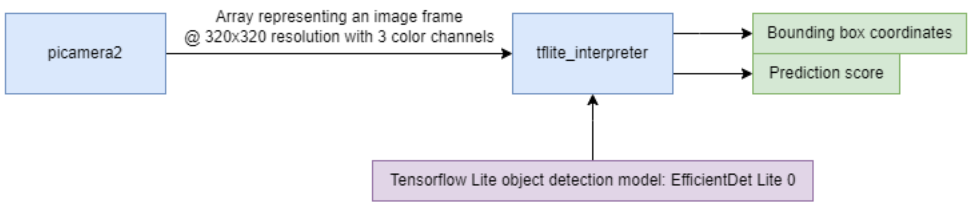

picamera2
Array representing an image frame
@ 320x320 resolution with 3 color channels
tflite_interpreter
Bounding box coordinates
Prediction score
Tensorflow Lite object detection model: EfficientDet Lite 0

Figure 5.27: Software, config 2

Reflections

This section covers aspects of configuration 3 we would have liked to explore further, if given the time.

- **Camera choice:** EfficientDet Lite 0 and 1 take input images in the sizes 320x320 and 384x384 respectively. Capturing video at resolutions above these values are a waste of resources. We should have used a camera with the ability to capture at low res with a high Field of View. Pi Camera V2 can capture at 640x480, but at this resolution the field of view is severely limited[32]. This issue could likely be resolved by using a different camera, unfortunately it was an issue we discovered very late in the project.
- **Models:** Getting the YOLOv5 model used in configuration 1 to run on the TPU would result in a better comparison between the two.
- **Multi TPU setup:** We only purchased one TPU which left us unable to explore the effects of a multi TPU setup. The Edge TPU Compiler can compile models in segments, each segment then runs on an individual TPU increasing the total throughput.

5.7.3 Configuration 3, Pi Zero w/ Coral Edge TPU

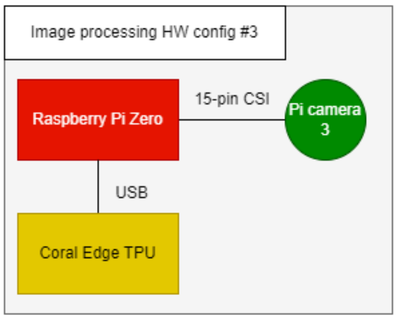


Figure 5.28: Hardware architecture, config 3

Software

Configuration 3 runs the same software as configuration 2.

5.7.4 Configuration 4, Pi 4

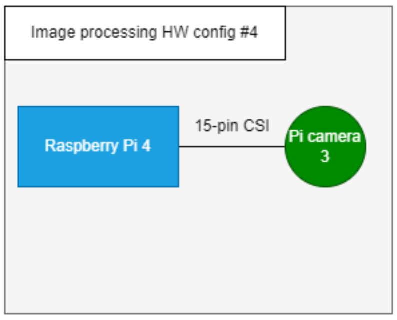


Figure 5.29: Hardware architecture, config 4

In this section, we will present Configuration 4, which stands out as the sole configuration that does not incorporate hardware acceleration, unlike the other configurations discussed in this report.

Blob detection software

For image capture, we use the Pi Camera V3, interfaced via the Picamera2 library. And for image processing, we implement OpenCV [33], a widely-used library known for its powerful computer vision and image processing capabilities. It is particularly suitable for blob detection because of its versatility, efficiency, and ease of use. We also used a helper library, cvzone [34], to reduce code and complexity.

Blob detection models and techniques

We opted for blob detection or blob analysis [35], an image processing technique that identifies and analyzes distinct areas of interest within an image. In the context of our project, blob detection was tasked with identifying balls of different colors.

Blob detection was chosen for its adaptability and precision. It can be finely tuned to detect specific objects, has a high degree of flexibility to meet different computational needs, and is particularly suited to the Raspberry Pi 4's limited processing power. These attributes made blob detection a compelling choice, as our project's aim was to balance detection accuracy with computational efficiency.

To identify the desired objects, our blob detection algorithm utilizes the HSV color space[36], which separates the hue, saturation, and value/brightness components of an image. By selecting appropriate color ranges, we are able to isolate specific objects based on their distinctive color properties.[35] Once we have found a colored object through color-based detection, our algorithm employs mathematical equations to analyze the contour properties of these objects. Specifically, we determine whether the contours of the detected blobs exhibit circular characteristics. This step helps differentiate the desired balls from other shapes or artifacts in the image. [37]

Blob detection offers a high level of modifiability, allowing it to be tailored to meet specific computational requirements. By adjusting key parameters, the algorithm can be implemented with minimal computational resources, making it suitable for applications where processing power is limited. Alternatively, a larger set of parameters can be utilized, enabling more intricate analysis and detection of complex blobs. However, it should be noted that this approach requires higher computational resources.

Considering the limitations of the Raspberry Pi 4 without acceleration hardware, the modifiability of blob detection allows us to strike a balance between detection accuracy and computational efficiency. We can optimize the algorithm by fine-tuning parameters to ensure reliable performance on this hardware platform.

Software overview

Camera interface:	Picamera2 Python library
Object detector type:	Blob detection algorithm
Image processing library:	OpenCV and cvzone
Model input:	Images from Pi Camera V3

Model output: Coordinates of HSV color values and contour properties (circle)

OpenCV and our blob detection algorithm process these frames to identify distinct blobs (regions of interest) within the image. The algorithm outputs the coordinates of detected blobs, along with their HSV color values and contour properties.

Config4 journey

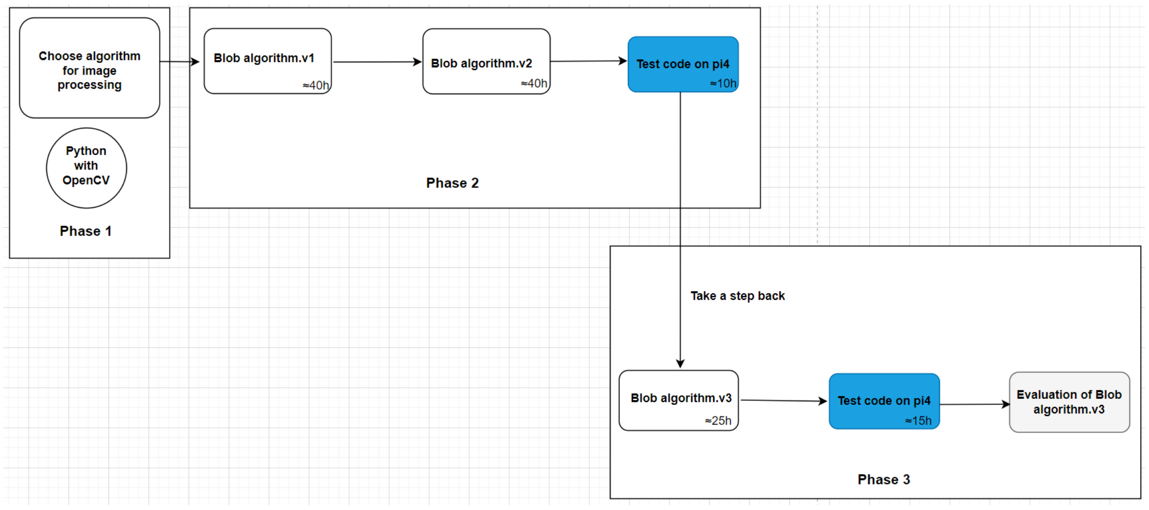


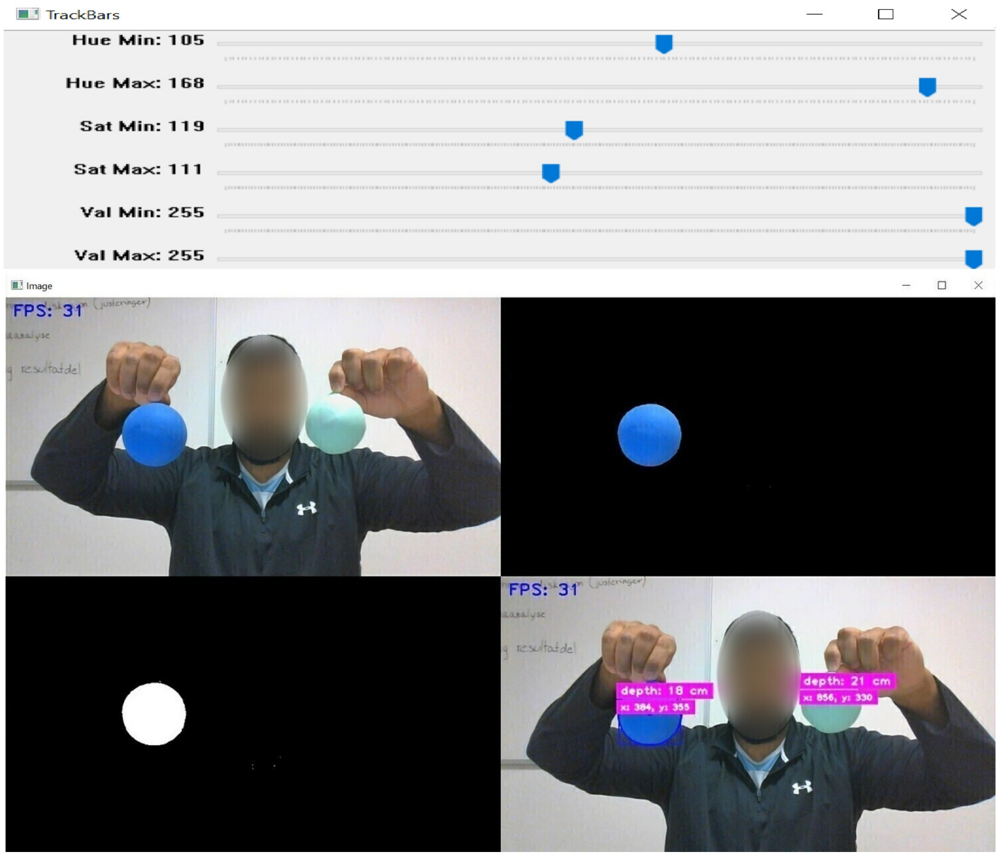
Figure 5.30: Config model, config 4

To provide a clear and comprehensive understanding of our journey through this configuration, we developed a model accompanied by an explanatory guide. The model is divided into three phases, which illustrate the different stages of our work, the problems we encountered, and the solutions we implemented. This approach offers a better view of how our project developed from start to finish.

In the first phase of our task, we decided to use Python with OpenCV for image processing. Python is a flexible and user-friendly language, making it an ideal choice for quick development. OpenCV is a powerful tool for image processing, offering a wide range of optimized algorithms. Using Python and OpenCV together enabled us to build and test our image-processing algorithms efficiently and effectively.

After deciding to use Python and OpenCV for image processing, we needed to choose a suitable algorithm for our task. We selected blob detection, considering its efficiency and simplicity. This algorithm was an easy choice due to the lack of computing resources in our configuration. It's important to note that efficiency and effectiveness can still depend on the specifics of a task.

Moving to the second phase, we knew one of our client's wishes was for us to develop a program capable of detecting three tennis-sized balls, each of a different color. The program was also required to calculate and display their x, and y coordinates, the distance of each detected ball, and FPS on the frame. Initially, we worked in such a way that we developed and tested the program on a high-end laptop with a webcam. At this point, our blob detection algorithm could only detect the ball at approximately 1m, when our goal was to detect it up to 3-4m. Here is a picture of how we defined the color range for detection.



TrackBars

Hue Min: 105
Hue Max: 168
Sat Min: 119
Sat Max: 111
Val Min: 255
Val Max: 255

Image

FPS: 31
depth: 18 cm
x: 384, y: 355
depth: 21 cm
x: 856, y: 330

Figure 5.31: ColorFinder object, config 4

Having realized that the initial version (Blob detection.v1) C of our program did not fulfill the client's requirement, it became evident that we had to improve our blob detection. As we went to the next step to expand the program's capabilities from detecting one ball to three balls, the computational demands increased, which lead to the challenge of balancing efficiency and performance within our resource constraints.

With the improvements in place, our v2 version C of the program had advanced to the point where it could successfully detect three differently colored balls, simultaneously displaying their depth along with their x and y coordinates within the frame. However, despite better efficiency and accuracy, we were yet to reach our objective of detecting the balls from a distance of 3-4 meters. Furthermore, the frame rate was still a matter of concern, as we were only able to achieve an average of 4-8 frames per second, varying based on the resolution scale.

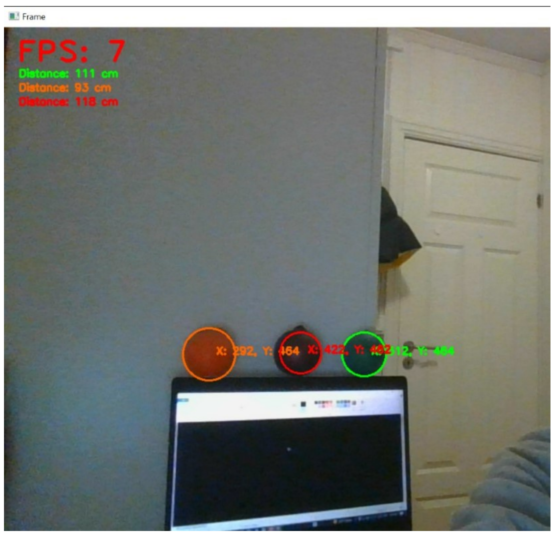


Figure 5.32: BlobDetection.v2, config 4

Upon testing the program on a Raspberry Pi, we found that the computational demands were too high for the hardware to handle effectively since we built and tested the program on a high-end laptop using the webcam. As a result, we needed to revisit our approach and develop a more simplified version that could effectively run on the Raspberry Pi.

In the third phase of our progress, in response to the computational constraints, we developed a simpler blob detection algorithm for the v3 version C. By simplifying and removing some of the more computationally intense functionalities from the program, we were able to finally achieve the detection of the balls from a distance of 3-4 meters on our PC. While the precision of this version was somewhat compromised in comparison to v2, it was a necessary trade-off given our resource constraints. This version also offered a significant improvement in frame rate performance as we were now achieving between 20-25 frames per second, also resolving the previous frame rate issues we had encountered.

Upon testing our code on the Raspberry Pi, we encountered consistent results, primarily due to the frame rate on the desktop PC being limited to 30 fps. Due to the change of cameras being used from webcam to pi camera 3, the HSV values needed to be updated. The code can be modified to be more accurate with further iterations since we see that we can press the raspberry pi even further.

To further test our configuration, we set out to evaluate the effectiveness of our blob detection. For this purpose, a script was developed to process labeled validation images, calculating precision, recall, and eventually yield the f1 score. This precision metric was chosen to compare our algorithm's performance with a trained model.

The optimal score we achieved was the result of testing over 10,000 different combinations of HSV values:

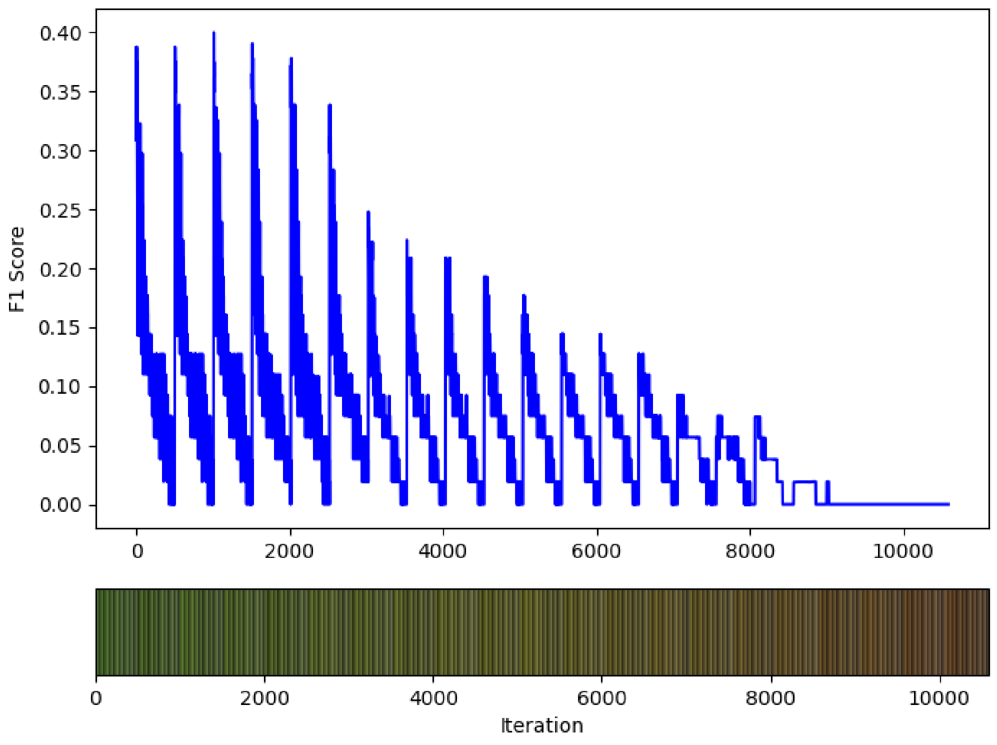


Figure 5.33: Trying different HSV combinations

It's important to clarify that this figure does not aim to showcase the accuracy of our algorithm. Rather, it demonstrates our methodology for optimizing the F1 score across a set of images with varying lighting conditions and environmental factors. C.4

However, given the algorithm's dependency on HSV values, its performance varied significantly under different lighting conditions.[38] This was a result of predetermined HSV values becoming inconsistent under varying lighting conditions, leading to inaccurate detections and incomplete detection of objects. To evaluate this, we utilized one image set captured under a wide variety of conditions. The images were taken in different environments, with lighting and distance variables changing extensively. The performance in these assorted conditions yielded varied results, indicating that the image set's robustness under uniform lighting conditions does not necessarily translate to the same level of accuracy under diverse lighting and environmental conditions.

The final test was done on FPS, where we recorded the fps over 30 secounds. This test was done using the ROS architectures and was relativly stable. We had one drop in FPS, but we did not have the time to investigate the manner further.

Complexity

While the initial implementation of the blob detection was relatively straightforward, achieving optimal performance was a more intricate process. It required meticulous fine-tuning of parameters to adapt to various conditions, and additional efforts were needed to handle the irregularities present in real-world data.

Maintenance added to the overall complexity, with a recurring need to manually adjust HSV values each time the environment changed. This repetitive process, although not adding significant complexity, was time-consuming and could potentially impact overall system efficiency. It's an area we aim to improve in future iterations of the project.

Despite these complexities, this configuration proved less complex in comparison to others, due largely to fewer encountered difficulties underway.

5.8 Configurations, full context

In this section, we present our configurations in the full UAV context, from image processing to the flight controller. We will cover ROS2 integration of the image processing modules, and the ROS2 nodes we have developed as we worked towards a complete UAV system.

5.8.1 ROS2 integration

As all image processing modules have the same outputs, the process of integrating them with ROS2 is the same for each module. An image processing module is split into two nodes which each handle a specific task. One node handles video capture, while the other node handles object detection. The benefit of this modular approach is not obvious for our configurations as both video capture and object detection run on the same hardware. Having the option to run these nodes on dedicated hardware could be useful in the future, which is why we went with the modular approach. An abstract explanation of our nodes can be found in figure 5.34. For more details see appendix G, which contains commented code for each node we have developed.

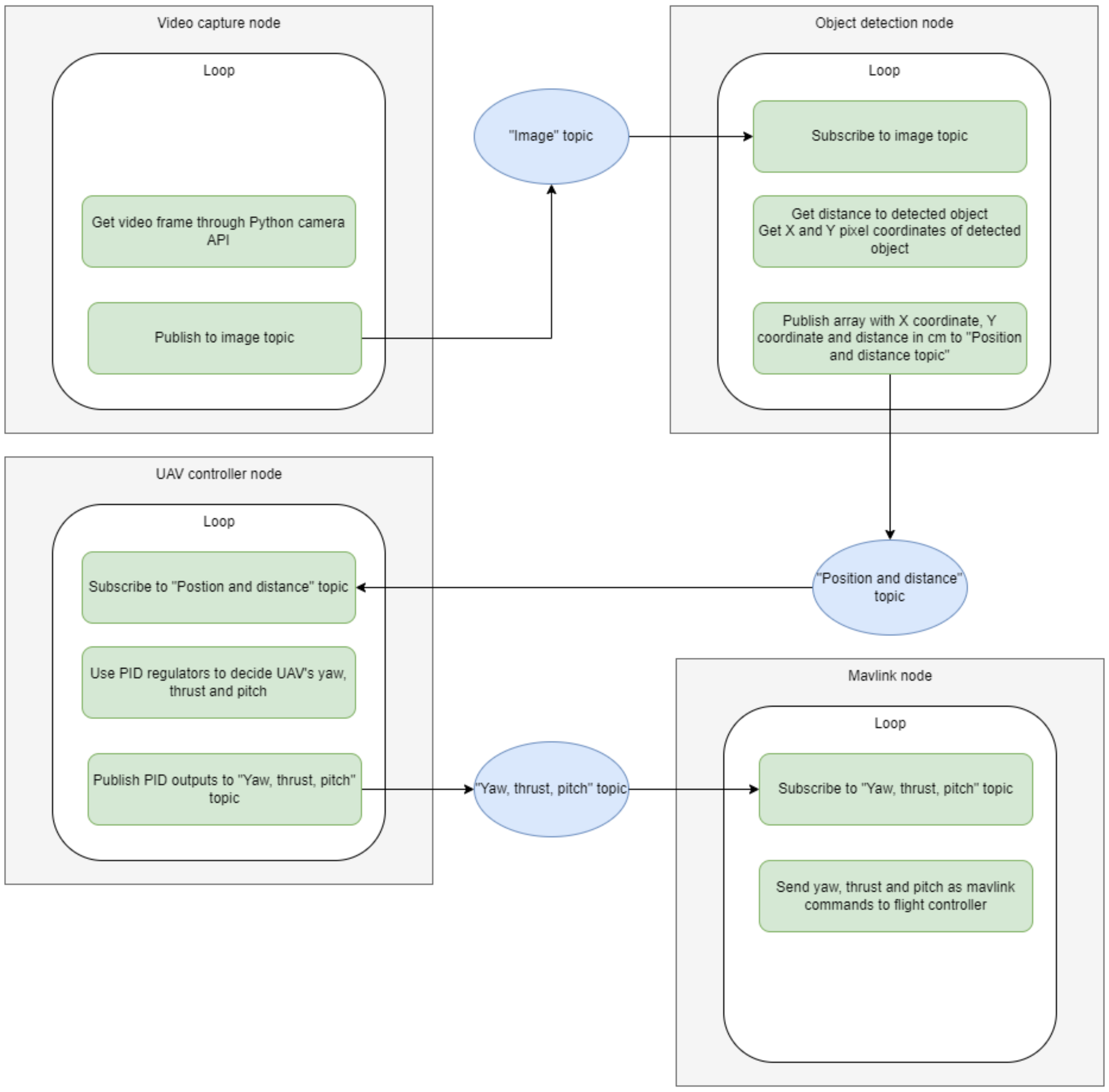

Video capture node
Loop
Get video frame through Python camera API
Publish to image topic
"Image" topic
Object detection node
Loop
Subscribe to image topic
Get distance to detected object Get X and Y pixel coordinates of detected object
Publish array with X coordinate, Y coordinate and distance in cm to "Position and distance topic"
UAV controller node
Loop
Subscribe to "Postion and distance" topic
Use PID regulators to decide UAV's yaw, thrust and pitch
Publish PID outputs to "Yaw, thrust, pitch" topic
"Position and distance" topic
"Yaw, thrust, pitch" topic
Mavlink node
Loop
Subscribe to "Yaw, thrust, pitch" topic
Send yaw, thrust and pitch as mavlink commands to flight controller

Figure 5.34: ROS2 node description

5.8.2 UAV side project

Throughout this study we have had ambition to fly a UAV with the ability to follow an object, as mentioned briefly in 5.3.3. The “UAV controller node” and “Mavlink node” in figure 5.34 were made to achieve this goal. Results from real world testing on a UAV would be valuable when comparing the performance of our configurations. Unfortunately, we didn’t reach the point where we could launch a drone with the object following capability. We elaborate on this side project in appendix B.2

5.9 Drone architecture

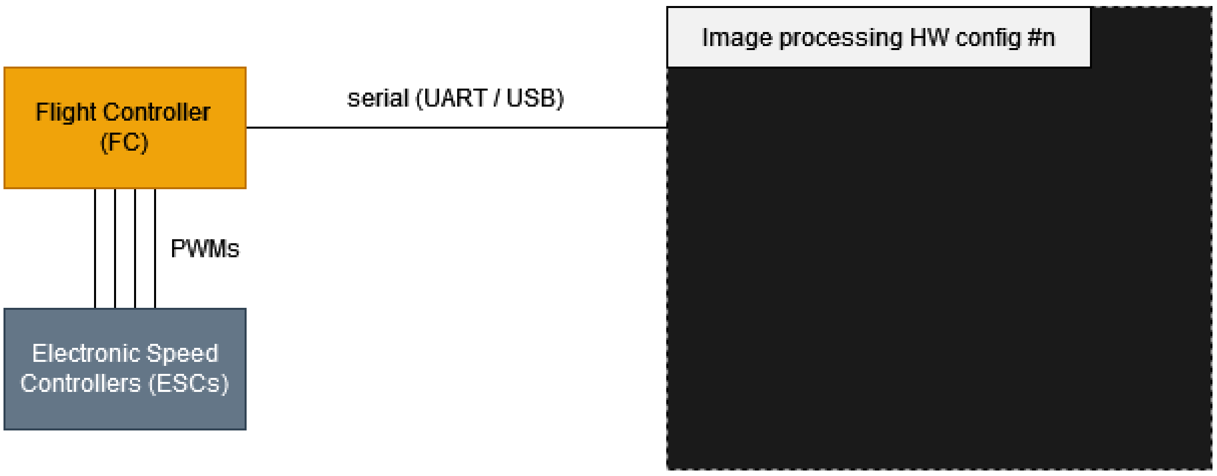


Image processing HW config #n
serial (UART / USB)
Flight Controller (FC)
PWMs
Electronic Speed Controllers (ESCs)

Figure 5.35: Hardware architecture including the flight controller and ESCs

The configuration consists of a flight controller flashed with a firmware featuring the ability to receive relatively high-level commands from any image processing-stack and turn it into real, physical movement of a drone.
The firmware chosen for this Bachelor's thesis is the "ArduCopter"-firmware from the well-known ArduPilot[2] project, widely regarded as the best open-source flight controller firmware for UAV projects.[39] The high-level commands in question being sent is from the widely used MAVLink[40] messaging protocol which can be received and transmitted through serial communications (UART / USB).

5.9.1 Flight controller firmware setup

The drone's flight controller has to be flashed with a version of ArduCopter and all necessary calibrations, ESC/motor-setup and tuning can be done through ArduPilot's official Mission Planner-application[2] installed on a desktop computer running Windows or Linux operating system.
The flight controller has to also be configured to accept the MAVLink messaging protocol on one of its serial Rx/Tx-ports, which also can be configured through Mission Planner.

5.9.2 Communication software setup

The single-board computer in the image processing config-stack needs a way to transmit and recieve MAVLink messages to and from the flight controller. There exists readily available software solutions which can encode MAVLink messages and transmit them over a serial communications interface such as UART or USB.

Two software solutions were assessed during this Bachelor's thesis:
The ”Pymavlink” Python-libraries[41] and the ”MAVROS” ROS-package[42].

While both these solutions were assessed, only ”Pymavlink” was actually implemented in a working configuration. This is due to the ”MAVROS” ROS-package for the supported ROS 2 distributions (Foxy and Humble) still being in an alpha state[43] as of the time of writing this report, and was therefore omitted.
More in-depth information and examples on MAVLink and ”Pymavlink” can be found in B.1.2.

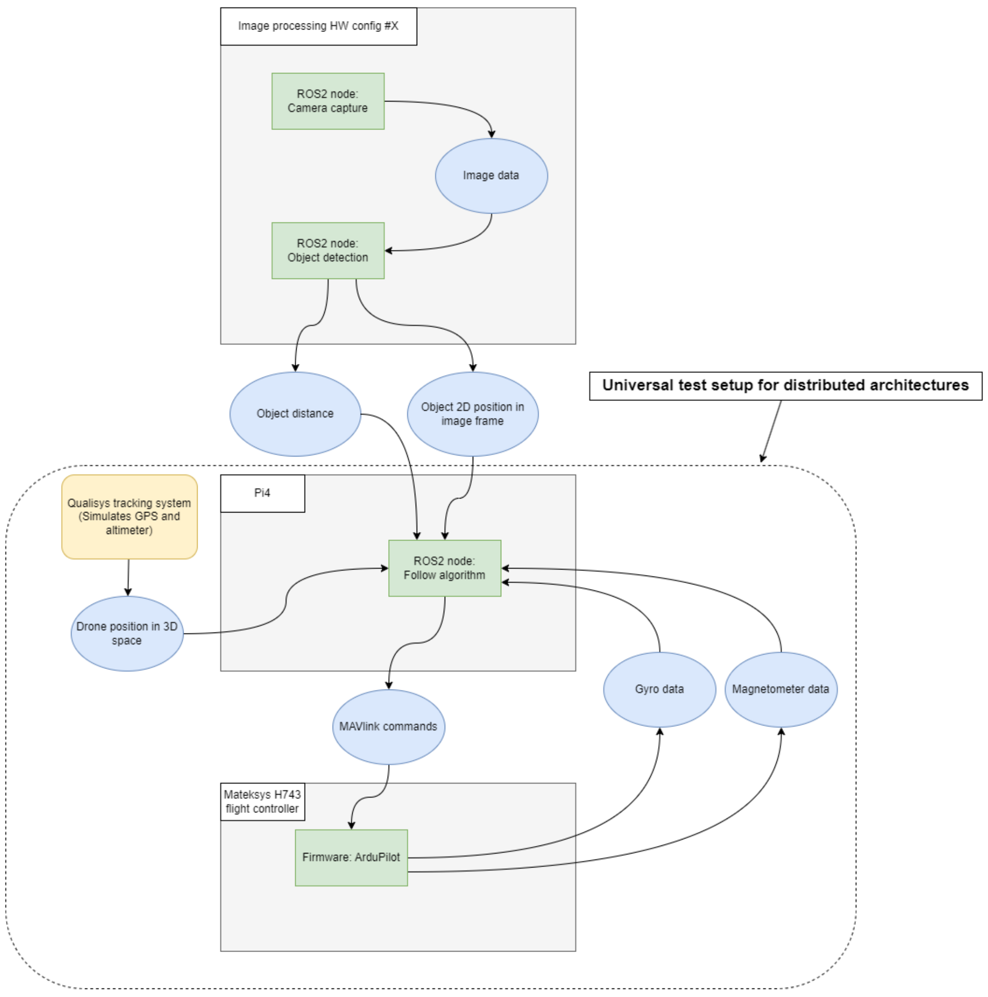


Image processing HW config #X
ROS2 node: Camera capture
Image data
ROS2 node: Object detection
Object distance
Object 2D position in image frame
Universal test setup for distributed architectures
Qualisys tracking system (Simulates GPS and altimeter)
Pi4
ROS2 node: Follow algorithm
Drone position in 3D space
Gyro data
Magnetometer data
MAVlink commands
Mateksys H743 flight controller
Firmware: ArduPilot

Figure 5.36: Dataflow during test scenario

5.10 Exploring Use cases

In this section of the research report, we delve into one of the primary motivations for this project, which is the potential applications of edge computing. Of particular interest is the use case involving drones, which encapsulates the entire spectrum of our work, ranging from the simulation of a drone via the Qualisys motion capture system to the actual construction and piloting of one.

In the following subsections, we will discuss the various tests conducted in our research, along with a detailed analysis of their respective outcomes.

5.10.1 Qualisys and drone tracking

The Qualisys motion capture system is a powerful tool that we use to pinpoint the exact location of the drone. It gives us real-time data on the drone's position in 3D space, with six degrees of freedom (6DOF). This means it can track the drone's movements forward and backward, up and down, left and right, as well as its rotations around three perpendicular axes. With this system, we can monitor and evaluate the drone's position in any environment, including a room. [44]

Drone tracking with Qualisys

Qualisys is exceptionally efficient in tracking drones. It goes beyond merely determining the drone's location; Qualisys supplies accurate tracking data that captures the drone's nuanced movements in real-time. Specifically, we are focused on the yaw, x and y coordinates. The yaw value helps us understand the drone's rotation around the vertical axis, while the x and y coordinates pinpoint the drone's position in three-dimensional space. By comparing this data with our drone location algorithm's output, we can assess the algorithm's accuracy and reliability. If there are discrepancies, we can refine our algorithm using this high-quality data, ensuring the drone's precise positioning and control.

Drone position

In order to calculate the drone position we need to get the distance data (denoted as $\boldsymbol{d}$) from our image processing configurations, and we need to get the yaw (denoted as $\boldsymbol{\theta}$) from the qualisys. We use the yaw data to simulate magnetometer data we would get from the flightcontroller. This testing have been done with a dummy drone with qualisys markers on the floor, so that we don't factor in the z axis.

We're essentially using a polar coordinate system with the object at the origin. In this system, a point is described by its distance from the origin, and its angle measured counterclockwise from a reference direction. For our specific problem, this angle is the drone's yaw.

Given this setup, the drone's x (x_{drone}) and y (y_{drone}) positions can be calculated by using trigonometric functions with the distance to the object and the drone's yaw. This calculation assumes that the object is at a location relative to origio, and the drone's yaw is defined as 0 when the drone is north of the object with the camera pointing south, increasing counterclockwise, and decreasing clockwise.

The equations to calculate the drone's position are as follows:

$$x_{\text{drone}} = x_{\text{object}} - d \cdot \sin(\theta) \tag{5.4}$$

$$y_{\text{drone}} = y_{\text{object}} + d \cdot \cos(\theta) \tag{5.5}$$

These equations account for the drone's distance from the object and its orientation. The trigonometric functions sin and cos help to decompose the total distance d into x and y components, providing an estimate for the drone's position. After the estimation, we compare the calculated position with the true position. This allows us to gauge the accuracy of our algorithm. In addition, we plot these positions over time to visually assess the performance of our drone location algorithm.

Test Results

With the use of ROS2s get_logger().info() and the plots generated, we were able to reliably read the test data from our calculations:

Yaw (rad)	Distance to Object (cm)	Estimated Position(x,y)	True Position(x,y)	Difference (%)
-0.045	92	(0.0395, 0.919)	(-0.086, 0.941)	2.65
1.524	88	(-0.878, 0.0498)	(-0.984, -0.004)	10.6
3.015	80	(-0.100, -0.793)	(-0.051, -0.810)	1.43
-1.581	86	(0.859, -0.005)	(0.773, -0.017)	11.1

Table 5.4: Drone Position Testing Data

The numbers we're discussing are average measurements taken from each yaw, using a still camera. These measurements consider all areas of the coordinate system that our Qualisys tracking system is adjusted to match. We compare our test data with the real coordinates from the Qualisys Tracking Manager (QTM). By finding the distance between each point, we can estimate the difference between them. This test was conducted with the object at the origin point (0,0).

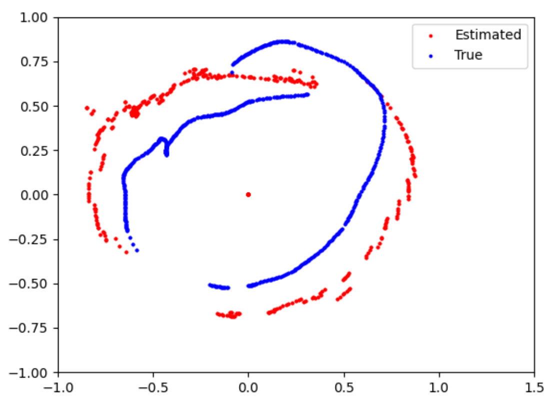


Figure 5.37: Estimated drone position vs Qualisys tracking data

Data from a moving camera, as shown in 5.37, is not as accurate as the data from a still camera. This might be because the camera was placed half a meter above the ground, something we didn't account for in our tests. Even so, the results indicate that our function is working correctly, which is our main goal.

The testing was done with image processing configuration 4. And since the contours in the frames fluctuate a little from each time, this also factor in on the estimated values.

ROS1 to ROS2

The previous project by Local Hawk had utilized Qualisys for drone tracking, and we opted to use their existing codebase. However, this code was designed for ROS1, which required us to convert it to the newer ROS2 standard.

Initially, we contemplated using a bridge between ROS1 and ROS2. However, given that this bridge is still in the alpha phase and only supports standard ROS messages, it was not suited for our application as we were dealing with custom message types developed by Local Hawk. Hence, we decided to translate the ROS related code from ROS1 to ROS2. We did not make any big changes to the calculations, other than setting some initial values, due to errors parsing the data into the calculation function.

A significant portion of the changes we made were the underlying architecture and methodology of ROS2, which diverges significantly from its predecessor.

The API client library has been changed, and they changed the achitecture to be more object oriented in ROS2. So everything from simple syntax to architecture have been changed. But most of the time was understanding how the ROS1 script worked.

One main challenge arose due to the Qualisys Tracking Manager (QTM) library's dependency on ”Asyncio”. Both Asyncio and ROS utilize event loops, which manage and distribute the execution of different tasks in an asynchronous programming environment.

This meant we needed to learn how the Asyncio event loop worked to understand what changes had been made. Essentially, these event loops allow multiple tasks to be executed concurrently, without the need for multi-threading or multi-processing. They achieve this by running one task until it needs to wait for an external event (like an I/O operation), then pausing that task and running another. This allows the program to utilize CPU time efficiently, as it can continue processing other tasks while waiting for the external event, instead of just sitting idle.[45]

Instead of using the ROS2 spin function, we put the node into the Asyncio event loop. The Local Hawk team started this, but we had to change some parts of the code. You can see these changes in our code breakdown here.

6 Measurements

Performance Measurement Results

The purpose of this section is to evaluate and compare the performance of the four different hardware configurations. This assessment will help us determine the most suitable configuration for achieving the desired balance between processing power, accuracy, and energy efficiency.

Detection accuracy (precision, recall, F1-score)

- Precision: Precision is a measure of how many of the detected objects are actually relevant. It is calculated as the number of true positives (TP) divided by the sum of true positives and false positives (FP). A high precision indicates that the object detection system is good at identifying relevant objects while avoiding false detections. Precision = TP / (TP + FP)
- Recall: Recall is a measure of how many of the relevant objects are detected by the system. It is calculated as the number of true positives (TP) divided by the sum of true positives and false negatives (FN). A high recall indicates that the object detection system is good at finding all the relevant objects in the scene. Recall = TP / (TP + FN)
- F1-score: The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both precision and recall. This is useful when you want to compare the performance of different object detection systems, especially when there's a trade-off between precision and recall. An F1-score closer to 1 indicates a better-performing object detection system. F1-score = 2 * (Precision * Recall) / (Precision + Recall) [46][p. 155-156]

Frame Rate

- Frame Rate, often measured in frames per second (fps), refers to the speed at which the system can process consecutive images for object detection. It represents the number of images the system can analyze and generate object detection results for within a second.
- Higher frame rates indicate better performance, as the system can analyze more images in a given period. This is particularly important for drone applications, where real-time or near-real-time object detection is crucial for seamless operation and rapid decision-making.

Power consumption

- Power consumption refers to the amount of electrical power used by the edge configurations while performing object detection tasks. It is measured in watts (W) and is obtained by monitoring the current and voltage supplied to the device during operation.
- Lower power consumption is generally more desirable, as it indicates higher energy efficiency and can result in longer flight times for the drone. Comparing the power consumption of the four configurations can help determine which option is better suited for a lightweight drone, where battery life and weight are critical factors.

Weight Efficiency

- Weight efficiency refers to the total weight of the configuration, including the computational hardware (like the Jetson Nano or Raspberry Pi), any attached accelerators (like the Google Coral TPU), and necessary components for their function (like heat sinks, cables, power supply, etc.). In the context of drones, lower weight is preferable as it can lead to longer flight times and improved maneuverability.

Complexity (Ease of Setup and Operation)

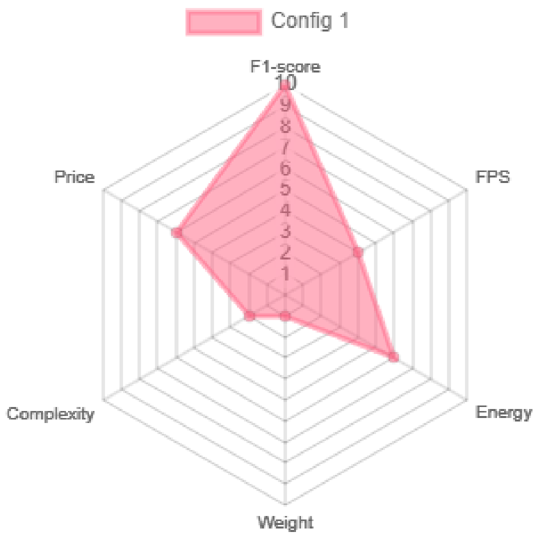
- Complexity in this context refers to the level of difficulty involved in setting up and operating the object detection system. This includes tasks such as installing and configuring the necessary software, implementing the object detection model, troubleshooting issues, and managing the system during operation.
- 1 represents a very complex system that is difficult to set up and operate. This might include challenges such as complicated installation procedures, hard-to-resolve errors, poor documentation, a steep learning curve, and significant maintenance requirements.

- 10 represents an easy-to-use system that is straightforward to set up and operate. This might include factors such as clear and thorough documentation, simple installation procedures, easy-to-use tools and interfaces, minimal troubleshooting requirements, and low maintenance needs.

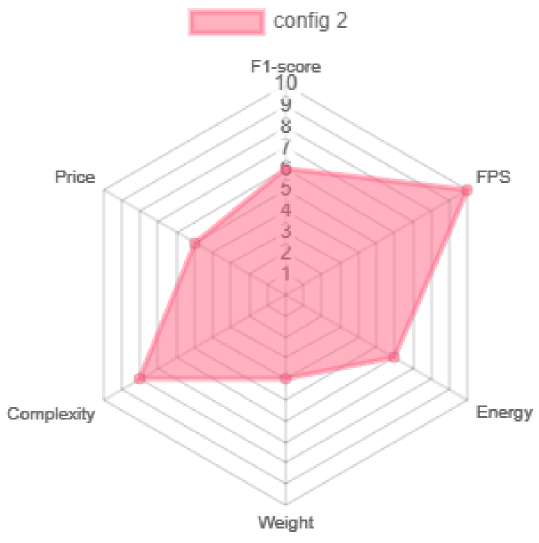
Results

Configuration	config 1 yolov5n	config 1 yolov5s	config 2 edl0	config 2 edl1	config 3 edl0	config 3 edl1	config 4 blob
Precision	0.991	0.995	0.945	0.924	0.945	0.924	0.96
Recall	1.00	1.00	0.516	0.580	0.516	0.580	0.25
F1-score	0.96	0.96	0.667	0.712	0.667	0.712	0.399
Avg FPS over 30 sec	11.03	5.966	24.64	16.46	5.70	3.92	23.33
Power Consumption (W)	10.8		11.7		11.7		7,2 W
Weight (g)	246.1		87.6		43.3		51.1
Complexity	2		9		9		7
Price	124$		145$		100$		85$

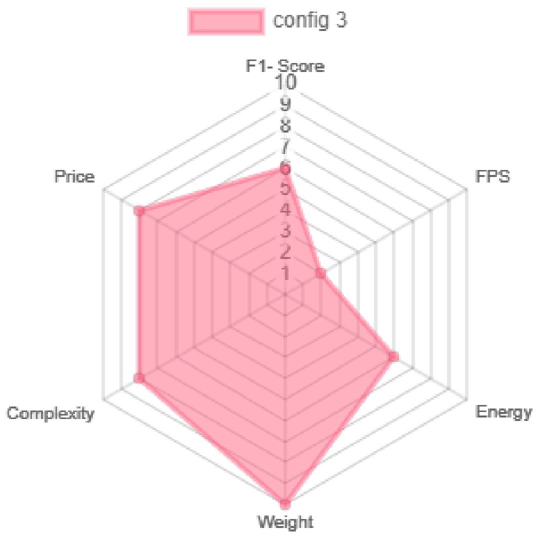
Table 6.1: Results



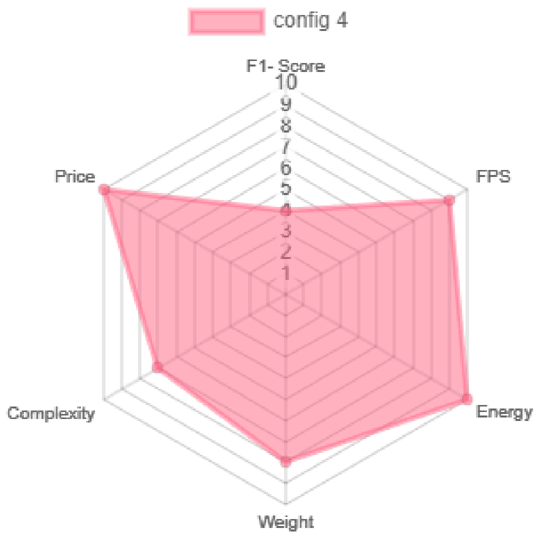
(a) Config 1 results



(b) Config 2 results



(c) Config 3 results



(d) Config 4 results

Figure 6.1: Results represented in spider diagrams

6.1 Config1

The following figure show the FPS performance of configuration 1 while running object detection. We ended up testing two Yolo models, nano and small.

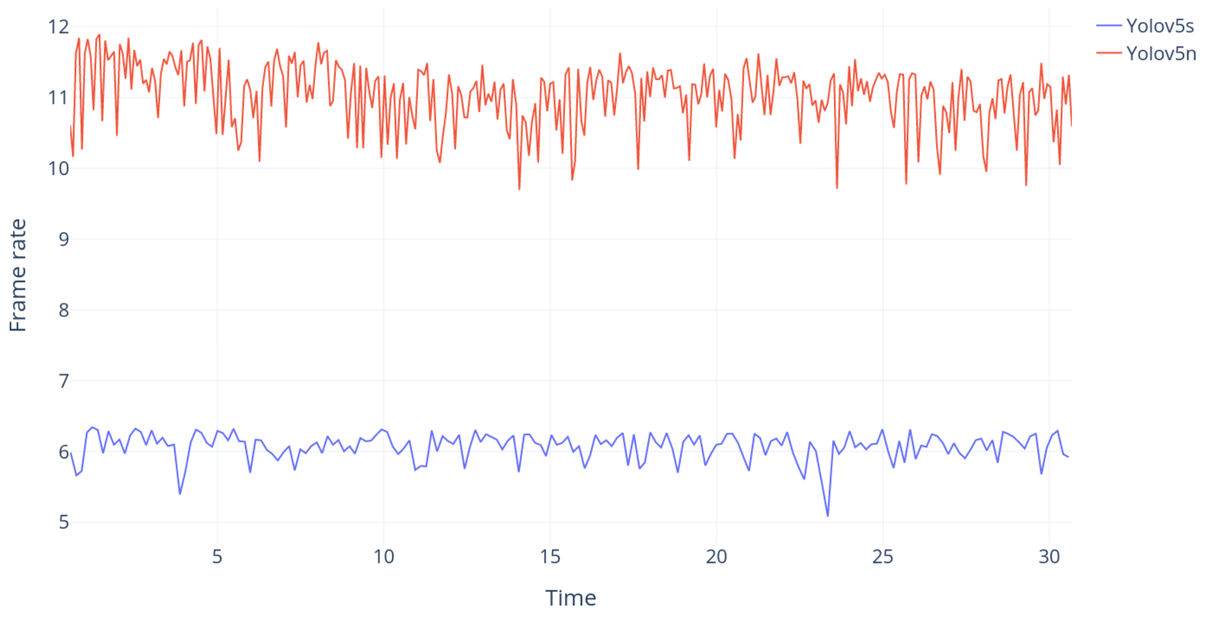


Figure 6.2: Yolov5 nano versus small model FPS

6.2 Config2 & Config3

Both the EfficientDet Lite 0 and EfficientDet Lite 1 models has been benchmarked on the follwing three hardware configurations:

- Raspberry Pi 4B w/ Coral USB Accelerator (Config 2)
- Raspberry Pi Zero2 w/ Coral USB Accelerator (Config 3)
- Raspberry Pi 3B+ w/ Coral USB Accelerator

All three hardware configurations has been benchmarked by running the Docker-container at ”benchmark/coral-1xRPi” from the ”Aerial-Edge/Drone” repository [47] with the latest Raspberry Pi OS Lite (64-bit) natively installed on a Class A2 SDXC-card.

6.2.1 EfficientDet Lite 0

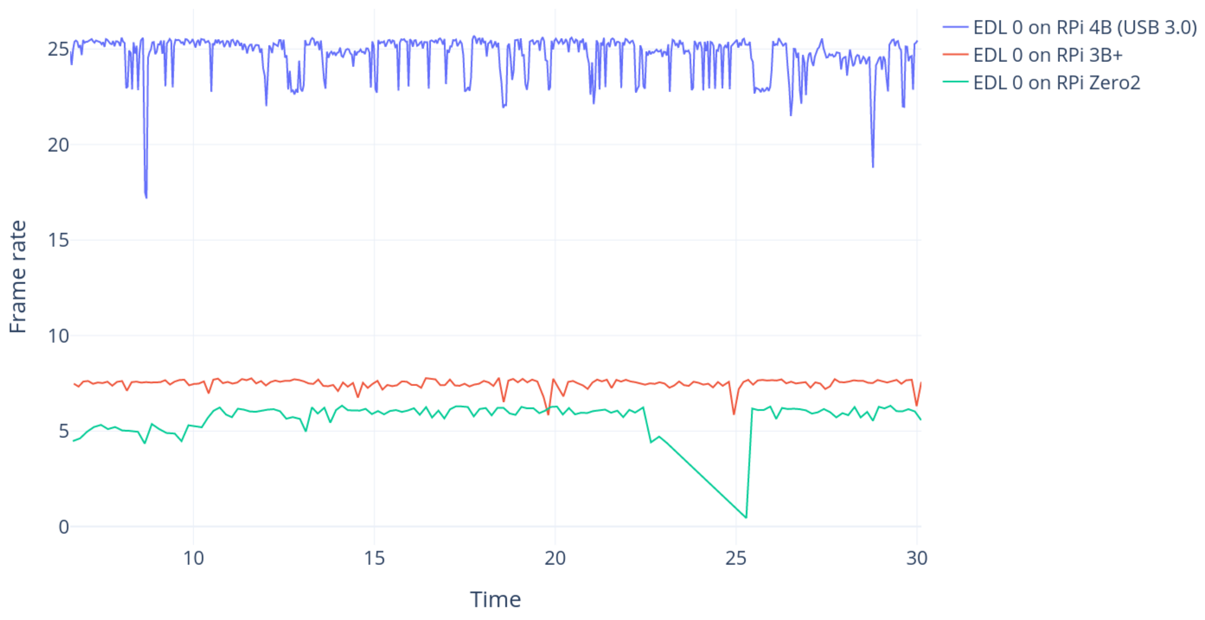


Figure 6.3: EfficientDet Lite 0 on various Raspberry Pies

6.2.2 EfficientDet Lite 1

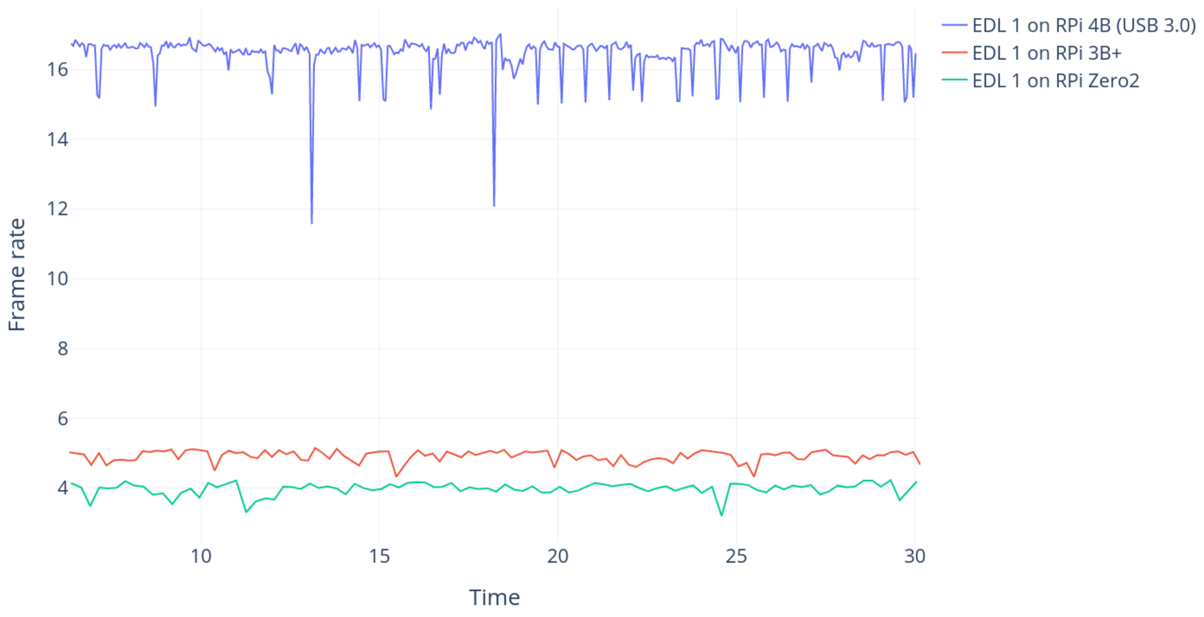


Figure 6.4: EfficientDet Lite 1 on various Raspberry Pies

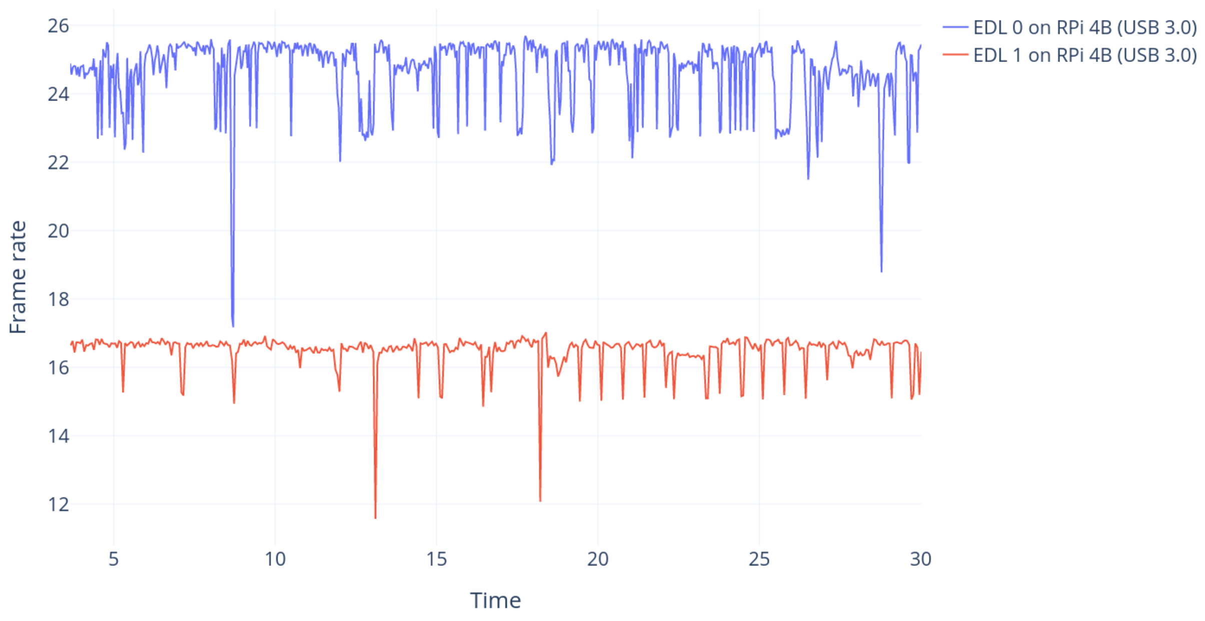


Figure 6.5: EfficientDet Lite 0 compared to EfficientDet Lite 1 on Raspberry Pi 4B

6.2.3 USB 3.0 vs. USB 2.0

Plugging the Coral USB Accelerator to a USB 2.0-port instead of 3.0-port pretty much halves the frame rate during the object-detection benchmark. This consistent with the Coral's tech specs[48] which claims that the USB Accelerator is compatible with USB 2.0 but the inferencing speed is slower. Note that both the Raspberry Pi 3B+ and Zero 2 only have USB 2.0 ports but the 4B still produces substantially higher frame rate than these two when it also is connected on USB 2.0.

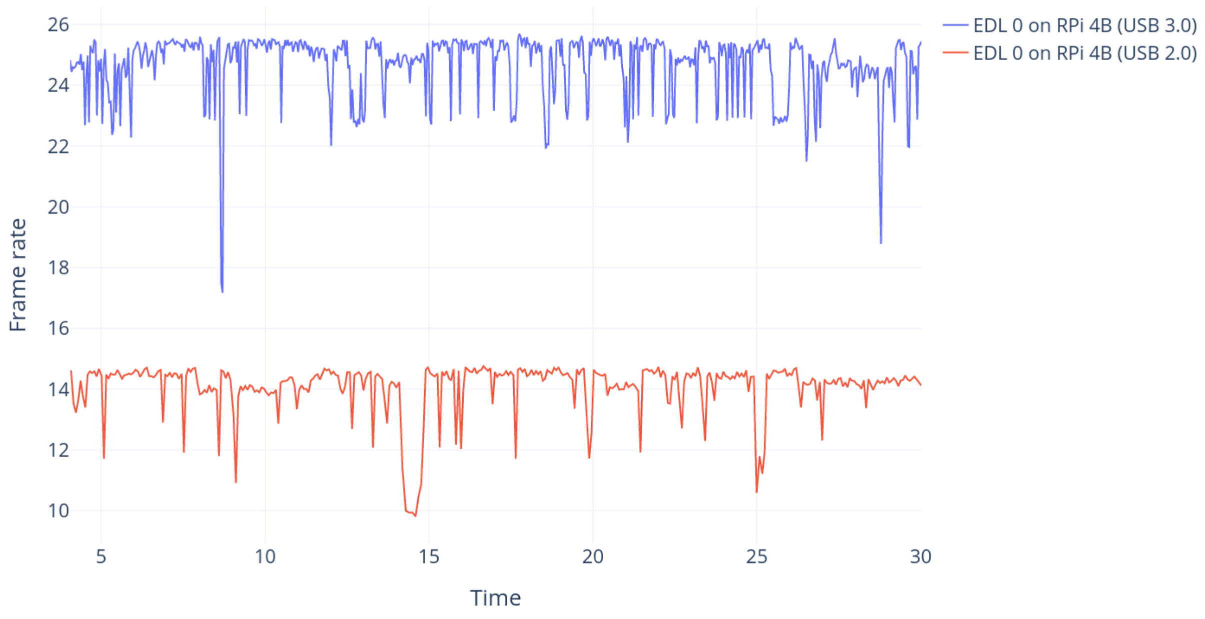


Figure 6.6: EfficientDet Lite 0, USB 3.0 vs 2.0 on RPi 4B

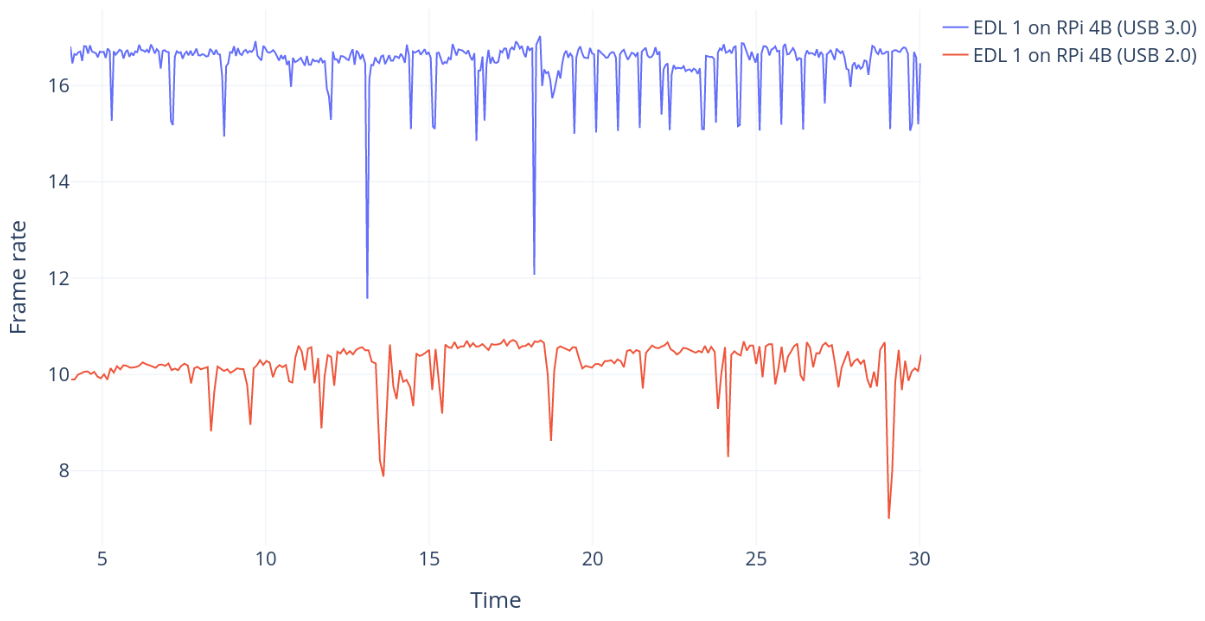


Figure 6.7: EfficientDet Lite 1, USB 3.0 vs 2.0 on RPi 4B

6.3 Config4

The following figure displays the frame per second (FPS) performance of Configuration 4 while tracking a single object over a 30-second interval with our final blob detection: G.1.4

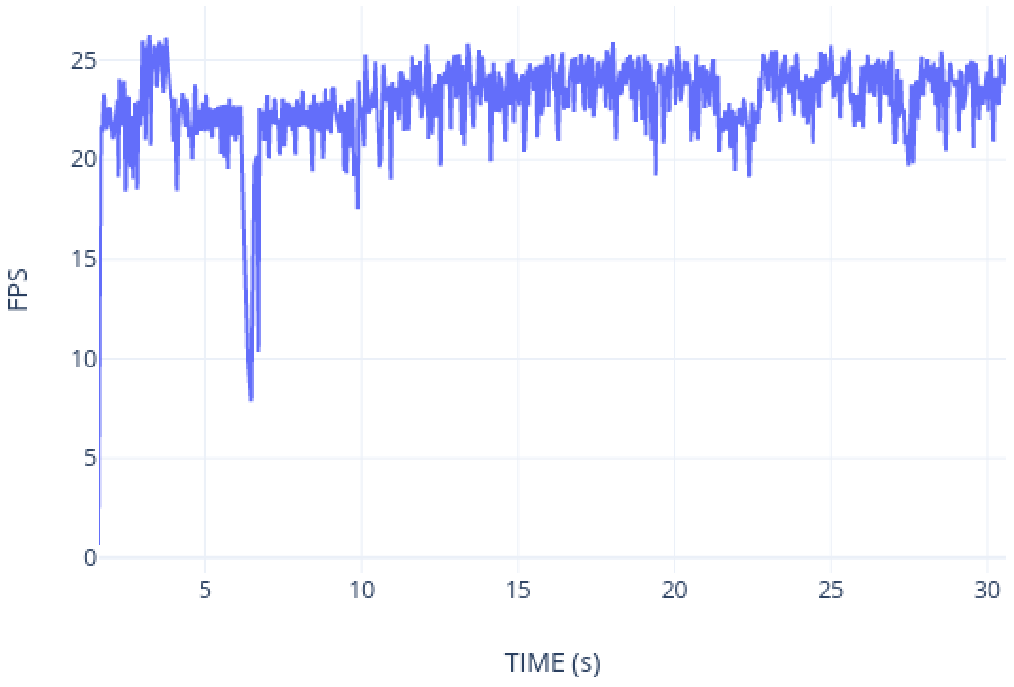


Figure 6.8: FPS, Configuration 4

7 Conclusion

Our bachelor's project aimed to provide readers with valuable insights into edge image processing for lightweight unmanned aerial vehicles (UAVs). Through our research and experimentation, we successfully achieved object detection capabilities tailored specifically for such UAVs. We utilized four different hardware and software configurations to accomplish this task.

By benchmarking our four configurations, we assessed their respective detection accuracy and frame-per-second (FPS) performance on a single object. This allowed us to comprehensively evaluate the suitability and effectiveness of each configuration for lightweight UAVs in terms of precision and real-time processing capabilities.

Our findings not only contribute to the field of edge image processing but also hold practical significance for the Local Hawk Project and its endeavors. The results highlight the importance of selecting appropriate hardware and software configurations to achieve optimal object detection performance within the constraints of lightweight UAVs.

Furthermore, this project serves as a foundation for future research and development in the area of edge image processing for lightweight UAVs. The insights gained from our study can guide further improvements in detection accuracy and FPS, leading to enhanced capabilities for lightweight UAVs in various domains, including surveillance, monitoring, and search-and-rescue operations.

Overall, our project successfully addresses the goal of providing the reader with knowledge on edge image processing for lightweight UAVs. Through our experimentation and evaluation, we have demonstrated the feasibility and efficacy of object detection using different configurations, thereby contributing to the advancement of lightweight UAV technologies and applications.

References

[1] mateksys.com. 'Flight controller h743-slim v3.' (), [Online]. Available: http://www.mateksys.com/?portfolio=h743-slim. (accessed: 20.03.23).

[2] ArduPilot.org. 'Ardupilot documentation.' (), [Online]. Available: https://ardupilot.org/ardupilot/. (accessed: 20.03.23).

[3] KongsbergGruppen. 'Om kongsberg gruppen.' (), [Online]. Available: https://www.annual-report.kongsberg.com/no/om-kongsberg-gruppen/dette-er-kongsberg-gruppen/. (accessed: 05.02.23).

[4] S. N. Leksikon. 'Dialektikk.' (), [Online]. Available: https://snl.no/dialektikk. (accessed: 08.02.23).

[5] R. Bologna, F. Trede and N. Patton, 'Bourdieu and jung: A thought partnership to explore personal, social, and collective unconscious influences on professional practices,' *Qualitative Report*, vol. 25, p. 3519, Oct. 2020. DOI: 10.46743/2160-3715/2020.4184.

[6] A. Sienkiewicz. 'Iteration vs sprint vs cadence in agile.' (), [Online]. Available: https://bigpicture.one/sprint-cadence-iteration/. (accessed: 18.05.23).

[7] L. Rosencrance. 'What is risk analysis?' (), [Online]. Available: https://www.techtarget.com/searchsecurity/definition/risk-analysis. (accessed: 20.03.23).

[8] V. Solutions. 'Risk matrix calculations – severity, probability, and risk assessment.' (), [Online]. Available: https://www.vectorsolutions.com/resources/blogs/risk-matrix-calculations-severity-probability-risk-assessment/. (accessed: 20.03.23).

[9] H. Underwood. 'Outline risk identifying risks to your project change managing change for your project.' (), [Online]. Available: https://slideplayer.com/slide/11367849/. (accessed: 20.03.23).

[10] P. Guevara. 'A guide to understanding 5x5 risk matrix.' (), [Online]. Available: https://safetyculture.com/topics/risk-assessment/5x5-risk-matrix/. (accessed: 20.03.23).

[11] A. Kayid, Y. Khaled and M. Elmahdy, 'Performance of cpus/gpus for deep learning workloads,' May 2018. DOI: 10.13140/RG.2.2.22603.54563.

[12] Nvidia. 'Jetson nano developer kit.' (), [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit. (accessed: 18.03.23).

[13] Coral.ai. 'Usb accelerator.' (), [Online]. Available: https://coral.ai/products/accelerator. (accessed: 18.03.23).

[14] R. P. Ltd. 'Raspberry pi camera module 3.' (), [Online]. Available: https://datasheets.raspberrypi.com/camera/camera-module-3-product-brief.pdf. (accessed: 20.03.23).

[15] LCSC. 'Camera module v2.' (), [Online]. Available: https://datasheet.lcsc.com/lcsc/1810010717_Raspberry-Pi-RPICAMERABOARD_C110649.pdf. (accessed: 20.03.23).

[16] R. P. Ltd. 'About the camera modules.' (), [Online]. Available: https://www.raspberrypi.com/documentation/accessories/camera.html. (accessed: 20.03.23).

[17] Branka. 'Linux statistics – 2023.' (), [Online]. Available: https://truelist.co/blog/linux-statistics/. (accessed: 18.05.23).

[18] Phoronix. 'Raspberry pi os 32-bit vs. 64-bit performance.' (), [Online]. Available: https://www.phoronix.com/review/raspberrypi-32bit-64bit. (accessed: 22.05.23).

[19] J. Redmon, S. Divvala, R. Girshick and A. Farhadi. 'You only look once: Unified, real-time object detection.' (), [Online]. Available: https://arxiv.org/abs/1506.02640. (accessed: 16.03.23).

[20] G. Jocher. 'Nvidia jetson nano deployment.' (), [Online]. Available: https://docs.ultralytics.com/yolov5/. (accessed: 12.05.23).

[21] Ultralytics. 'Ultralytics yolov8.' (), [Online]. Available: https://docs.ultralytics.com/#ultralytics-yolov8. (accessed: 16.03.23).

[22] A. Rosebrock. 'Find distance from camera to object.' (), [Online]. Available: https://pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/. (accessed: 19.05.23).

[23] Nvidia. 'Nvidia sdk manager.' (), [Online]. Available: https://developer.nvidia.com/sdk-manager. (accessed: 12.05.23).

[24] Nvidia. 'Jetpack sdk 4.6.1.' (), [Online]. Available: https://developer.nvidia.com/embedded/jetpack-sdk-461. (accessed: 18.03.23).

[25] Nvidia. 'Write image to the microsd card.' (), [Online]. Available: https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#write. (accessed: 12.05.23).

[26] G. Jocher. 'Nvidia jetson nano deployment.' (), [Online]. Available: https://docs.ultralytics.com/yolov5/tutorials/running_on_jetson_nano/. (accessed: 12.05.23).

[27] D. Franklin. 'Jetson inference.' (), [Online]. Available: https://github.com/dusty-nv/jetson-inference. (accessed: 20.05.23).

[28] Google/Tensorflow. 'Efficientdet-lite0 tensorflow hub page.' (), [Online]. Available: `https://tfhub.dev/tensorflow/lite-model/efficientdet/lite0/detection/default/1`. (accessed: 19.05.23).

[29] Google/Tensorflow. 'Efficientdet-lite1 tensorflow hub page.' (), [Online]. Available: `https://tfhub.dev/tensorflow/lite-model/efficientdet/lite1/detection/default/1`. (accessed: 19.05.23).

[30] Google/Coral. 'Tensorflow models on the edge tpu.' (), [Online]. Available: `https://coral.ai/docs/edgetpu/models-intro/`. (accessed: 19.05.23).

[31] Google/Coral. 'Edge tpu compiler.' (), [Online]. Available: `https://coral.ai/docs/edgetpu/compiler/`. (accessed: 19.05.23).

[32] P. contributors. 'Picamera documentation: Sensor modes.' (), [Online]. Available: `https://picamera.readthedocs.io/en/release-1.13/fov.html#sensor-modes`. (accessed: 20.05.23).

[33] OpenCV. 'Opencv documentation.' (), [Online]. Available: `https://docs.opencv.org/4.x/`. (accessed: 16.03.23).

[34] CVzone. 'Cvzone documentation.' (), [Online]. Available: `https://github.com/cvzone/cvzone`. (accessed: 16.03.23).

[35] 'Blob detection using opencv (python, c++).' (), [Online]. Available: `https://learnopencv.com/blob-detection-using-opencv-python-c/`. (accessed: 20.03.23).

[36] Packt. 'Object detection using color.' (), [Online]. Available: `https://subscription.packtpub.com/book/data/9781789537147/1/ch01lvl1sec09/object-detection-using-color-in-hsv`. (accessed: 20.03.23).

[37] 'Contour detection using opencv (python/c++).' (), [Online]. Available: `https://learnopencv.com/contour-detection-using-opencv-python-c/`. (accessed: 20.03.23).

[38] Aerial-Edge. 'Dataset for config4 evaluation.' (), [Online]. Available: `https://universe.roboflow.com/dataset-k3la3/ball-finder-a5yur/dataset/3`. (accessed: 20.03.23).

[39] OscarLiang.com. 'Firmware for fpv drone flight controller overview.' (), [Online]. Available: `https://oscarliang.com/fc-firmware/`. (accessed: 20.03.23).

[40] MAVLink.io. 'Mavlink developer guide.' (), [Online]. Available: `https://mavlink.io/en/`. (accessed: 20.03.23).

[41] ArduPilot. 'Python mavlink interface and utilities.' (), [Online]. Available: `https://github.com/ArduPilot/pymavlink`. (accessed: 21.05.23).

[42] ros.org. 'Mavros.' (), [Online]. Available: `http://wiki.ros.org/mavros`. (accessed: 20.03.23).

[43] mavlink. ‘Mavlink to ros gateway with proxy for ground control station.’ (), [Online]. Available: `https://github.com/mavlink/mavros`. (accessed: 21.05.23).

[44] Qualisys. ‘Qualisys tracking.’ (), [Online]. Available: `https://www.qualisys.com/engineering/robotics-and-uav/`. (accessed: 16.03.23).

[45] S. H. Alex Martelli Anna Ravenscroft, *Python in a Nutshell, 3rd Edition.* O’Reilly Media, Inc., 2017.

[46] H. S. Christopher D. Manning Prabhakar Raghavan, *An introduction to Information Retrieval.* Cambridge University Press, 2009.

[47] Aerial-Edge. ‘Composition of ros2 packages for the drone stack & benchmark script running in docker.’ (), [Online]. Available: `https://github.com/Aerial-Edge/Drone`. (accessed: 21.05.23).

[48] Coral.ai. ‘Coral about.’ (), [Online]. Available: `https://coral.ai/docs/accelerator/datasheet/`. (accessed: 16.03.23).

[49] Nvidia. ‘Nvidia tensorrt.’ (), [Online]. Available: `https://developer.nvidia.com/tensorrt`. (accessed: 16.03.23).

[50] PiDramble.com. ‘Power consumption benchmarks.’ (), [Online]. Available: `https://www.pidramble.com/wiki/benchmarks/power-consumption`. (accessed: 22.05.23).

[51] C. Software. ‘A deep dive into raspberry pi zero 2 w’s power consumption.’ (), [Online]. Available: `https://www.cnx-software.com/2021/12/09/raspberry-pi-zero-2-w-power-consumption/`. (accessed: 22.05.23).

[52] Elefun.no. ‘Mateksys h743-slim v3 flight controller.’ (), [Online]. Available: `https://www.elefun.no/p/prod.aspx?v=56509`. (accessed: 22.05.23).

[53] tensorflow.org. ‘Tfl about.’ (), [Online]. Available: `https://www.tensorflow.org/lite/guide`. (accessed: 16.03.23).

[54] wikipedia.org. ‘Single-board computer.’ (), [Online]. Available: `https://en.wikipedia.org/wiki/Single-board_computer`. (accessed: 16.03.23).

[55] OpenCV. ‘Opencv hough circle.’ (), [Online]. Available: `https://docs.opencv.org/4.x/d4/d70/tutorial_hough_circle.html`. (accessed: 16.03.23).

[56] OpenCV. ‘Opencv display.’ (), [Online]. Available: `https://docs.opencv.org/4.x/dc/da5/tutorial_py_drawing_functions.html`. (accessed: 16.03.23).

[57] P. Principles. ‘Opencv enumerate.’ (), [Online]. Available: `https://docs.opencv.org/4.x/dc/da5/tutorial_py_drawing_functions.html`. (accessed: 16.03.23).

[58] jeremiedecock. ‘Circle documentation.’ (), [Online]. Available: `https://github.com/jeremiedecock/vor12/blob/master/vor12/computer_vision/circle_detection.py`. (accessed: 16.03.23).

[59] kvaale. ‘Circle parameters.’ (), [Online]. Available: `https://github.com/Aerial-Edge/Containers/blob/main/cv-distance/app/main.py#L8`. (accessed: 16.03.23).

[60] R. P. PhD. ‘Contour, shape and color detection using opencv-python.’ (), [Online]. Available: `https://www.researchgate.net/publication/325195384_Contour_Shape_Color_Detection_using_OpenCV-Python`. (accessed: 16.03.23).

[61] TechVidvan. ‘Contour, shape and color detection using opencv-python.’ (), [Online]. Available: `https://techvidvan.com/tutorials/detect-objects-of-similar-color-using-opencv-in-python/`. (accessed: 16.03.23).

[62] goodday451999 and modalaashwin41. ‘Multiple color detection in real-time using python-opencv.’ (), [Online]. Available: `https://www.geeksforgeeks.org/multiple-color-detection-in-real-time-using-python-opencv/`. (accessed: 16.03.23).

[63] raspberrypi.com. ‘Datasheet for raspberry pi zero 2.’ (), [Online]. Available: `https://datasheets.raspberrypi.com/rpizero2/raspberry-pi-zero-2-w-product-brief.pdf`. (accessed: 20.03.23).

[64] raspberrypi.com. ‘Datasheet for raspberry pi 4b.’ (), [Online]. Available: `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf`. (accessed: 20.03.23).

[65] nvidia.com. ‘Datasheet for nvidia jetson nano.’ (), [Online]. Available: `https://developer.nvidia.com/embedded/dlc/jetson-nano-system-module-datasheet`. (accessed: 20.03.23).

[66] OscarLiang.com. ‘F1, f3, f4, g4, f7 and h7 flight controller processors explained.’ (), [Online]. Available: `https://oscarliang.com/f1-f3-f4-flight-controller/`. (accessed: 20.03.23).

[67] iNavFlight. ‘Ardupilot documentation.’ (), [Online]. Available: `https://github.com/iNavFlight/inav/wiki`. (accessed: 20.03.23).

[68] wikipedia.org. ‘Pid controller.’ (), [Online]. Available: `https://en.wikipedia.org/wiki/PID_controller`. (accessed: 20.03.23).

[69] raspberrypi.com. ‘Camera hardware specification.’ (), [Online]. Available: `https://www.raspberrypi.com/documentation/accessories/camera.html#hardware-specification`. (accessed: 20.03.23).

[70] wikipedia.org. ‘Rolling shutter.’ (), [Online]. Available: `https://en.wikipedia.org/wiki/Rolling_shutter`. (accessed: 22.05.23).

[71] R. Pi. ‘About the camera modules.’ (), [Online]. Available: `https://www.raspberrypi.com/documentation/accessories/camera.html`. (accessed: 22.05.23).

[72] libcamera.org. ‘Documentation - libcamera.’ (), [Online]. Available: `https://libcamera.org/docs.html`. (accessed: 20.03.23).

[73] raspberrypi. 'Github - raspicam.' (), [Online]. Available: `https://github.com/raspberrypi/userland/tree/master/host_applications/linux/apps/raspicam`. (accessed: 20.03.23).

[74] B. B. Rad, H. J. Bhatti and M. Ahmadi, 'An introduction to docker and analysis of its performance,' *IJCSNS International Journal of Computer Science and Network Security*, vol. 17, no. 3, pp. 228–235, 2017. [Online]. Available: `http://paper.ijcsns.org/07_book/201703/20170327.pdf`, (accessed 20.03.23).

[75] Gazebo.org. 'Gazebo.' (), [Online]. Available: `https://gazebosim.org/home`. (accessed: 22.05.23).

[76] IBM. (), [Online]. Available: `https://www.ibm.com/topics/computer-vision`. (accessed: 20.03.23).

[77] R. Kulhary. 'Opencv overview.' (), [Online]. Available: `https://www.geeksforgeeks.org/opencv-overview/`. (accessed: 20.03.23).

[78] A. Stefanuk. 'Hire opencv developers.' (), [Online]. Available: `https://mobilunity.com/blog/hire-opencv-developers/`. (accessed: 20.03.23).

[79] wikipedia. 'Blob detection.' (), [Online]. Available: `https://en.wikipedia.org/wiki/Blob_detection`. (accessed: 20.03.23).

[80] MathWorks. 'What is object detection.' (), [Online]. Available: `https://www.mathworks.com/discovery/object-detection.html`. (accessed: 20.03.23).

[81] T. A. Yuya Maruyama Shinpei Kato. 'Exploring the performance of ros2.' (), [Online]. Available: `https://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=7743223&tag=1`. (accessed: 15.05.23).

[82] . L. Puck et al. 'Performance evaluation of real-time ros2 robotic control in a time-synchronized distributed network.' (), [Online]. Available: `https://ieeexplore.ieee.org/document/9551447/`. (accessed: 15.05.23).

[83] M. K. Saeid Dehnavi. 'Compros: A composable ros2 based architecture for real-time embedded robotic development.' (), [Online]. Available: `https://www.researchgate.net/publication/354956131_CompROS_A_composable_ROS2_based_architecture_for_real-time_embedded_robotic_development`. (accessed: 15.05.23).

[84] A. Rosebrock. 'Intersection over union (iou) for object detection.' (), [Online]. Available: `https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/`. (accessed: 20.05.23).

[85] Chachart. 'Chachart spiderdiagram.' (), [Online]. Available: `https://chachart.net/radar?lang=en&fbclid=IwAR30Snjl0ijhznssThCggGnYFXUVBdKXyB_8i201KUosQIFeuUtpH`
(accessed: 21.05.23).

Appendix A

Hardware

A.1 Specification tables for configs

All values from these tables that are not referenced externally are done by own measuring. The power supply component for the single-board computers has been omitted from these tables due to there being several different options to choose from when supplying power to these boards and not all of them can be covered here.

A.1.1 Config 1

	MSRP	**Weight**	**Volume**	**Power (idle)**	**Power (max)**
Jetson Nano	\$ 99	241 g	232 cm^3	5 W	10 W
Pi camera 3	\$ 25	4 g	7 cm^3	0,66 W	0,83 W
Camera ribbon	-	1,1 g	-	-	-
SUM	\$ 124	**246,1 g**	**239 cm^3**	**5,7 W**	**10,8 W**

Table A.1: Hardware specification table for Config 1 hardware [12][69]

A.1.2 Config 2

	MSRP	Weight	Volume	Power (idle)	Power (max)
Raspberry Pi 4B 4GB	\$ 60	46 g	86 cm^3	2,7 W	6,4 W
Pi camera 3	\$ 25	4 g	7 cm^3	0,66 W	0,83 W
Camera ribbon	-	1,1 g	-	-	-
Coral USB	\$ 60	19,7 g	15,6 cm^3	2,5 W	4,5 W
USB C-A cable	-	16,8 g	-	-	-
SUM	\$ 145	**87,6 g**	**109 cm^3**	**5,7 W**	**11,7 W**

Table A.2: Hardware specification table for Config 2 hardware [64][50][69][48]

A.1.3 Config 3

	MSRP	Weight	Volume	Power (idle)	Power (max)
Raspberry Pi Zero2	\$ 15	11 g	9,75 cm^3	0,6 W	6,4 W
Pi camera 3	\$ 25	4 g	7 cm^3	0,66 W	0,83 W
Camera ribbon	-	1,1 g	-	-	-
Coral USB	\$ 60	19,7 g	15,6 cm^3	2,5 W	4,5 W
USB C-micro cable	-	7,5 g	-	-	-
SUM	\$ 100	**43,3 g**	**32 cm^3**	**3,8 W**	**11,7 W**

Table A.3: Hardware specification table for Config 3 hardware [63][51][69][48]

A.1.4 Config 4

	MSRP	Weight	Volume	Power (idle)	Power (max)
Raspberry Pi 4B	\$ 60	46 g	86 cm^3	2,7 W	6,4 W
Pi camera 3	\$ 25	4 g	7 cm^3	0,66 W	0,83 W
Camera ribbon	-	1,1 g	-	-	-
SUM	\$ 85	**51,1 g**	**93 cm^3**	**3,4 W**	**7,2 W**

Table A.4: Hardware specification table for Config 4 hardware [64][50][69]

	MSRP	Weight	Volume	Power (idle)	Power (max)
Config 1	\$ 124	246,1 g	239 cm^3	5,7 W	10,8 W
Config 2	\$ 145	87,6 g	109 cm^3	5,7 W	11,7 W
Config 3	\$ 100	43,3 g	32 cm^3	3,8 W	11,7 W
Config 4	\$ 85	51,1 g	93 cm^3	3,4 W	7,2 W

Table A.5: Hardware specification table for comparison of all configs

A.1.5 Comparison

A.2 Single-board computer (SBC)

A single-board computer (SBC) is a complete computer built on a single circuit board, with microprocessor(s), memory, input/output (I/O) and other features required of a functional computer. Single-board computers are commonly made as demonstration or development systems, for educational systems, or for use as embedded computer controllers. Many types of home computers or portable computers integrate all their functions onto a single printed circuit board.

Unlike a desktop personal computer, single board computers often do not rely on expansion slots for peripheral functions or expansion. Single board computers have been built using a wide range of microprocessors. Simple designs, such as those built by computer hobbyists, often use static RAM and low-cost 32- or 64-bit processors like ARM. Other types, such as blade servers, would perform similar to a server computer, only in a more compact format. [54]

Thanks to the characteristics of SBCs, they are a core hardware component in all of the architectural designs in this comparative study.

	RPi 4B	**RPi Zero2**	**Jetson Nano**
CPU	Cortex-A72 @ 1.5GHz	Cortex-A53 @ 1.0GHz	Cortex-A57 @ 1.43GHz
GPU	VideoCore IV @ 500MHz	VideoCore IV @ 400MHz	128-core Maxwell @ 921MHz
Memory	1 GB - 8 GB	512 MB	4 GB
Video decoding	H.264/H.265 (4Kp60)	H.264 (1080p30)	H.264/H.265 (4Kp60)
Video encoding	H.264/H.265 (1080p30)	H.264 (1080p30)	H.264/H.265 (4Kp30)
Connectivity	USB 3.0 × 2 USB 2.0 × 2 UART × 1 SPI × 2 I2C × 1	USB 2.0 × 1 UART × 1 SPI × 2 I2C × 1	USB 3.0 × 4 USB 2.0 × 1 UART × 1 SPI × 2 I2C × 3
Form factor	85mm × 56mm	65mm × 30mm	69mm × 45mm
Weight	46g	11g	250g
MSRP	$ 35 - $ 75	$ 15	$ 99

Table A.6: Comparison table for single-board computers (SBCs) [63][64][65]

When proposing what SBCs to deploy in our test-solution architectures we tend to mainly look at specifications regarding processing power (CPU & GPU), memory (RAM), hardware-accelerated video encoding/decoding and connectivity (USB, UART etc.) in relation to the weight and form factor of the board. We also need to take into consideration the availability and discrepancy between MSRP and actual sale price due to the current world-wide chip shortage disrupting the market.

We decided the top-contenders and implemented them across all configurations:

- Raspberry Pi Zero 2 (RPi Zero 2)
- Raspberry Pi 4B (RPi 4B)
- nVidia Jetson Nano

A.3 Camera

	Camera Module v2	**Camera Module 3 NoIR**	**Camera Module 3 Wide**
Video Modes	1920 × 1080p47 1640 × 1232p41 640 × 480p206	2304 × 1296p56 2304 × 1296p30 1536 × 864p120	2304 × 1296p56 2304 × 1296p30 1536 × 864p120
Focus	Adjustable	Motorized	Motorized
Depth of field	Approx 10 cm to ∞	Approx 10 cm to ∞	Approx 5 cm to ∞
Horizontal FoV	62.2 degrees	66 degrees	102 degrees
Vertical FoV	48.8 degrees	41 degrees	67 degrees
Size	25 × 24 × 9mm	25 × 24 × 11.5mm	25 × 24 × 12.4mm
Weight	3g	4g	4g
MSRP	$ 25	$ 25	$ 35

Table A.7: Comparison table for selected cameras[69]

A.3.1 Rolling vs. Global shutter

All the cameras selected for this project use a ”rolling shutter”, meaning each frame of a video is captured not by taking a snapshot of the entire scene at a single instant in time but rather by scanning across the scene rapidly, vertically, horizontally or rotationally. In other words, not all parts of the image of the scene are recorded at exactly the same instant. (Though, during playback, the entire image of the scene is displayed at once, as if it represents a single instant in time.) This produces predictable distortions of fast-moving objects or rapid flashes of light. This is in contrast with ”global shutter” in which the entire frame is captured at the same instant. [70]
A camera using a ”global shutter” would therefore cause less distortion of objects in frame moving very fast, for example in the case of a drone capturing high-speed video footage.

Raspberry Pi Global Shutter Camera

Raspberry Pi recently released a new camera during this spring named the ”Global Shutter (GS) Camera”. The Global Shutter Camera’s image sensor has a 6.3mm diagonal active sensing area, which is similar in size to Raspberry Pi’s HQ Camera. However, the pixels are larger and can collect more light. Large pixel size and low pixel count are valuable in machine-vision applications; the more pixels a sensor produces, the harder it is to process the image in real time. To get around this, many applications downsize and crop images. This is unnecessary with the Global Shutter Camera and the appropriate

lens magnification, where the lower resolution and large pixel size mean an image can be captured natively. [71]
The new ”GS Camera” would likely work substantially better for the application of capturing video on a drone for object detection, with the only drawback being additional weight from a heavier lens.

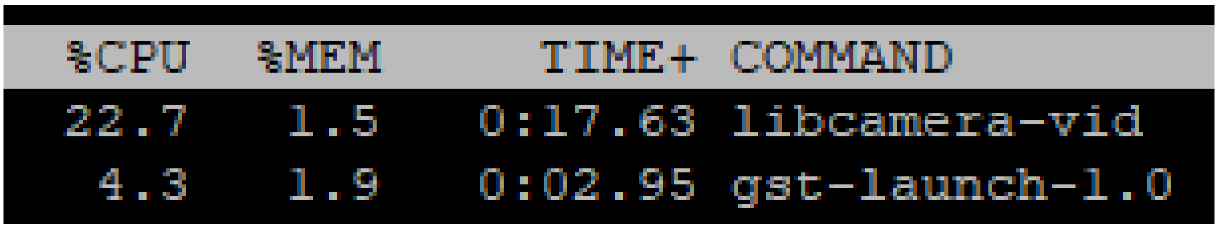
A.3.2 Camera drivers

There are currently two different driver libraries for capturing with the Raspberry Pi Cameras:

- libcamera [72]
- RaspiCam (legacy) [73]

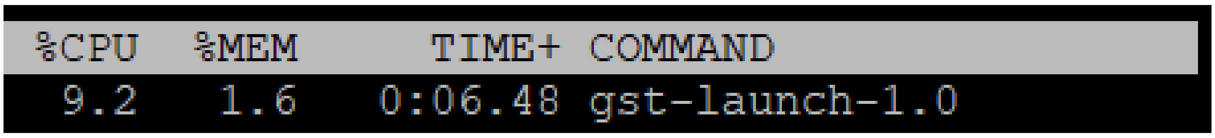
Some quick benchmark tests indicates that the legacy RaspiCam drivers outperforms the newer libcamera drivers in terms of CPU-usage. The reason for this is likely that the legacy drivers are proprietorially made by the own producer of the Rasberry Pi's GPU-stack (Broadcom) and are therefore more efficiently taking advantage of the hardware.

Below are screenshots of an RTP-stream, the first using piped output from ”libcamera-vid” and the second using ”v4l2src”, both with a Pi Camera Module v2.
The %CPU usage is over double for the ”libcamera-vid” command compared to ”v4l2src” when streaming @ 1080p30.



```
 %CPU  %MEM     TIME+ COMMAND
 22.7   1.5   0:17.63 libcamera-vid
  4.3   1.9   0:02.95 gst-launch-1.0
```

Figure A.1: libcamera-vid piping video to gstreamer



```
%CPU  %MEM     TIME+ COMMAND
 9.2   1.6   0:06.48 gst-launch-1.0
```

Figure A.2: v4l2src (RaspiCam) feeding video to gstreamer

Unfortunately, the newest Pi Camera Module v3 is not compatible with the legacy RaspiCam drivers.

A.4 Hardware Acceleration for ML Inference

A.4.1 Google Coral TPU

The Coral TPU is a compact, power-efficient chip designed by Google to accelerate TensorFlow Lite models on devices. It enables rapid machine learning processing, enhances data privacy, and eliminates the need for continuous internet connectivity. Developers can achieve high-performance machine learning inferencing, making it an ideal choice for applications like computer vision and natural language processing in various edge devices. [48]

Figure A.3: Coral USB accelerator [13]

A.5 Flight controller

When choosing flight controller (FC) board(s) for the hardware stack of our test-solution architectures we tend to mainly look at processor speed and flash memory in relation to form factor and weight as well as the availability in local retail shops. The boards' sensor and connectivity options are aspects we deem less relevant for our research project.

F1, F3, F4, G4, F7, and H7 are the different STM32 processors (aka MCU – Micro Controller Unit). The processor is the brain of a flight controller (FC), similar to the CPU in a computer.

There are currently 11 series of STM32 MCU, from faster to slower processing speeds they are: H7, F7, G4, F4, F3, F2, F1, F0, L4, L1, and L0. [66]

We decided to go for a single FC to be used in all the hardware stacks of our test-solution architectures of the latest and greatest generation FC that employs the fastest

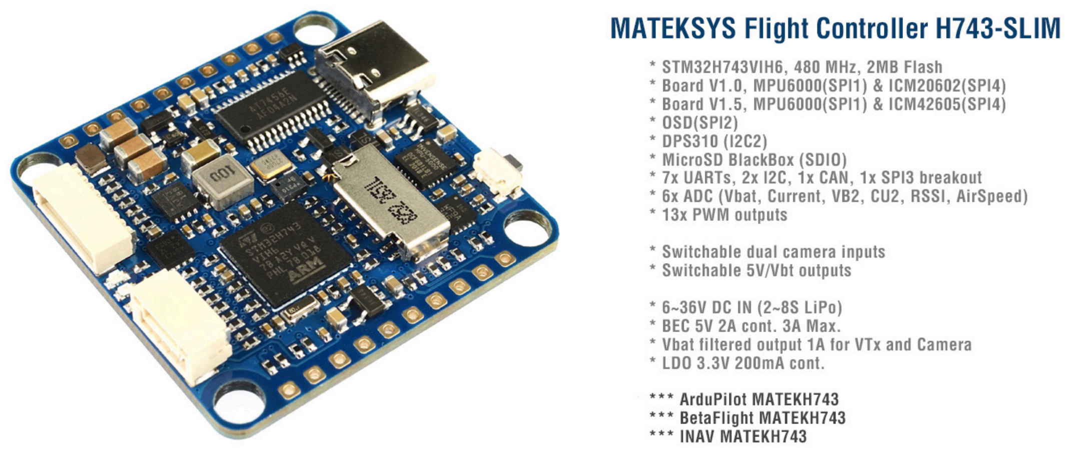
Processor	**Processor Speed**	**Flash Memory**	**SRAM**
F0 (STM32F051)	48MHz	256KB	32KB
F1 (STM32F103)	72MHZ	128KB	96KB
F3 (STM32F303)	72MHz	256KB	80KB
F4 (STM32F405)	168MHz	1MB	192KB
F4 (STM32F411)	100MHz	512KB	128KB
G4 (STM32G491)	170MHz	512KB	128KB
F7 (STM32F745)	216MHz	1MB	320KB
F7 (STM32F722)	216MHz	512KB	256KB
F7 (STM32F765)	216MHz	2MB	512KB
H7 (STM32H743)	480MHz	2MB	1MB

Table A.8: Comparison table for microcontroller unit (MCU) [66]

H7-generation microcontroller unit (MCU) with 2 MB of flash memory. This ensures the drone can run very smoothly and has sufficient flash memory to support any firmware with a full set of features.

We decided we wanted to go for a single FC for the hardware stacks of our test-solution architectures. Employing the "STM32H743" MCU, it ensures the drone can run very smoothly and has sufficient flash memory to support any firmware with a full set of features.

We ended up going for a MATEKSYS Flight Controller H743-SLIM [1] as it fulfills our criteria and could be readily ordered at the Norwegian online store "elefun.no".



MATEKSYS Flight Controller H743-SLIM

* STM32H743VIH6, 480 MHz, 2MB Flash
* Board V1.0, MPU6000(SPI1) & ICM20602(SPI4)
* Board V1.5, MPU6000(SPI1) & ICM42605(SPI4)
* OSD(SPI2)
* DPS310 (I2C2)
* MicroSD BlackBox (SDIO)
* 7x UARTs, 2x I2C, 1x CAN, 1x SPI3 breakout
* 6x ADC (Vbat, Current, VB2, CU2, RSSI, AirSpeed)
* 13x PWM outputs

* Switchable dual camera inputs
* Switchable 5V/Vbt outputs

* 6~36V DC IN (2~8S LiPo)
* BEC 5V 2A cont. 3A Max.
* Vbat filtered output 1A for VTx and Camera
* LDO 3.3V 200mA cont.

*** ArduPilot MATEKH743
*** BetaFlight MATEKH743
*** INAV MATEKH743

Figure A.4: Showcase of MATEKSYS H743-SLIM, from mateksys.com [1]

Appendix B

UAV

B.1 Flight controller

B.1.1 Firmware

Flight controller (FC) firmware is the software that runs on a flight controller and controls the operation of an FPV drone. It affects flight performance and features, and different firmware options offer various advantages and disadvantages for different flying styles and preferences. [39]

There are mainly two firmwares [39] to choose from for autonomous flying today:

- INAV
- ArduPilot

ArduPilot is perhaps the most popular open-source autopilot software suite. It supports a variety of vehicles, including quadcopters, planes, rovers, ground vehicles, even RC submarines.

ArduPilot is known for its extensive features and customization options, making it a good choice for advanced pilots and developers. It supports both autonomous and manual control modes, GPS waypoint navigation, and various sensors like barometers and magnetometers. [39]

Figure B.1: ArduPilot logo, from ardupilot.org [2]

ArduPilot was chosen as baseline firmware since it is a long-standing open-source project that's well-documented [2] compared to INAV [67]. ArduPilot is also the only firmware officially supported by the MATEKSYS Flight Controller H743-SLIM (ref: A.5) out of these two options.

B.1.2 MAVLink

MAVLink is a very lightweight messaging protocol for communicating with drones (and between onboard drone components).

MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission. [40]

MAVLink Commands

With the help of the relatively low-level "Pymavlink" Python-library we're able to send and receive messages to and from a ArduCopter-flashed flight controller. To do this, first import "mavutil" from the "pymavlink" libraries and establish a connection and wait for a heartbeat from the flight controller:

```python
from pymavlink import mavutil

# Tries to establish connection on Raspberry Pi serial UART:
the_connection = mavutil.mavlink_connection("/dev/ttyACM0", baud=57600)
the_connection.wait_heartbeat()
```

After "the_connection" is successfully initialized its from here on possible to communicate with the flight controller.

As an example, to command the flight controller to change mode, arm throttle and take off can be done by encoding each of these three desired commands in each their respective ”COMMAND_LONG” messages, including all the command’s parameters, as per the MAVLink.io[40] documentation:

```
# Sets flight controller flight mode to "GUIDED":
the_connection.mav.command_long_send(
    the_connection.target_system,           # Established after heartbeat
    the_connection.target_component,        # Established after heartbeat
    mavutil.mavlink.MAV_CMD_DO_SET_MODE,    # Command to be sent
    0,                                      # Confirmation bit
    1, 4, 0, 0, 0, 0, 0)                    # The command's 7 parameters

# Arms the throttle to allow for motors to spin:
the_connection.mav.command_long_send(
    the_connection.target_system,
    the_connection.target_component,
    mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM,
    0,
    1, 0, 0, 0, 0, 0, 0)

# Takes off to 5 metres above ground level:
the_connection.mav.command_long_send(
    the_connection.target_system,
    the_connection.target_component,
    mavutil.mavlink.MAV_CMD_NAV_TAKEOFF,
    0,
    0, 0, 0, 0, 0, 0, 5)
```

Every MAVLink command has 7 parameters that need to be set, these parameters can be looked up in the MAVLink.io[40] documentation for each respective command.

Manual Control Protocol

The ”MAVLink Commands” example above works fine for programmatically moving the drone on a ”macro”-scale where the flight controller has GNSS-signal available. To programmatically move the drone on a ”micro”-scale its possible to do this through the ”MANUAL_CONTROL”-message as per the MAVLink.io[40] documentation:

```
# After connection already established and heartbeat recieved:
the_connection.mav.manual_control_send(
    the_connection.target_system,   # Established after heartbeat
    x,                              # Pitch, in range [-1000,1000]
    y,                              # Roll, in range [-1000,1000]
    z,                              # Thrust, in range [-1000,1000]
    r,                              # Yaw, in range [-1000,1000]
    0)                              # Bitfield corresponding to extra
                                    # buttons, not needed and can be
                                    # set to 0 in this case
```

Note that the z-value (thrust) is in the range [-1000,1000] where -1000 is full negative thrust, 0 is no thrust and 1000 is full thrust. Most aircraft only operate in the range [0,1000] without any functionality for negative thrust.
Also note that the thrust can not be modified through this protocol unless the flight controller is in a flight mode that supports manual control, as per the ArduPilot.org[2] documentation.

B.1.3 Drone simulation

Simulation is implemented by using a Flight Dynamics Model (FDM) of the vehicle to simulate the physics involved with vehicle movement. It receives inputs from a SITL (Software in the Loop) program running the ArduPilot firmware (which are the firmware's servo/motor outputs) and outputs vehicle status, position, velocities, etc. that result from those inputs back to the firmware simulation. Just as sensors would in the real world case. [2]

SITL Simulator (Software in the Loop)

The SITL simulator allows you to run Plane, Copter, or Rover without any hardware. It is a build of the autopilot code using an ordinary C++ compiler, giving you a native executable that allows you to test the behavior of the code without hardware. [2]

Through simulating the drone's flight controller running ArduPilot and interfacing with it through MAVLink messages we're able to test code in a safe environment before deploying it on a physical drone. With the addition of Gazebo [75] we can see a 3D-rendering of the drone and its surrounding, making it ideal for controlling the simulated drone on a "micro"-scale.

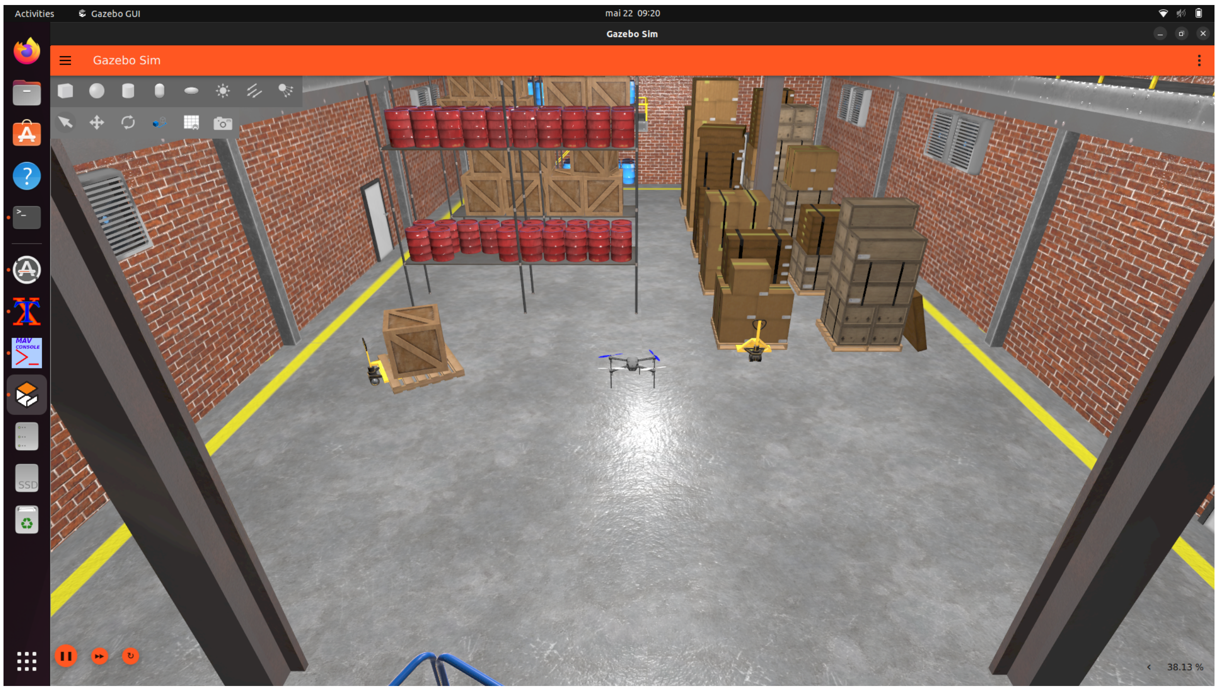


Figure B.2: SITL with Gazebo

B.2 Drone implementation

During the project a four rotor ”quadcopter” drone was built as the project’s attempt at a real-world implementation, mainly to act as a platform for our proof of concept.
Due to time constraints and hardware difficulties, the full context configuration with all the ROS2-nodes could not be tuned properly and therefore not implemented safely. The full context implementation was limited only to simulations with SITL.

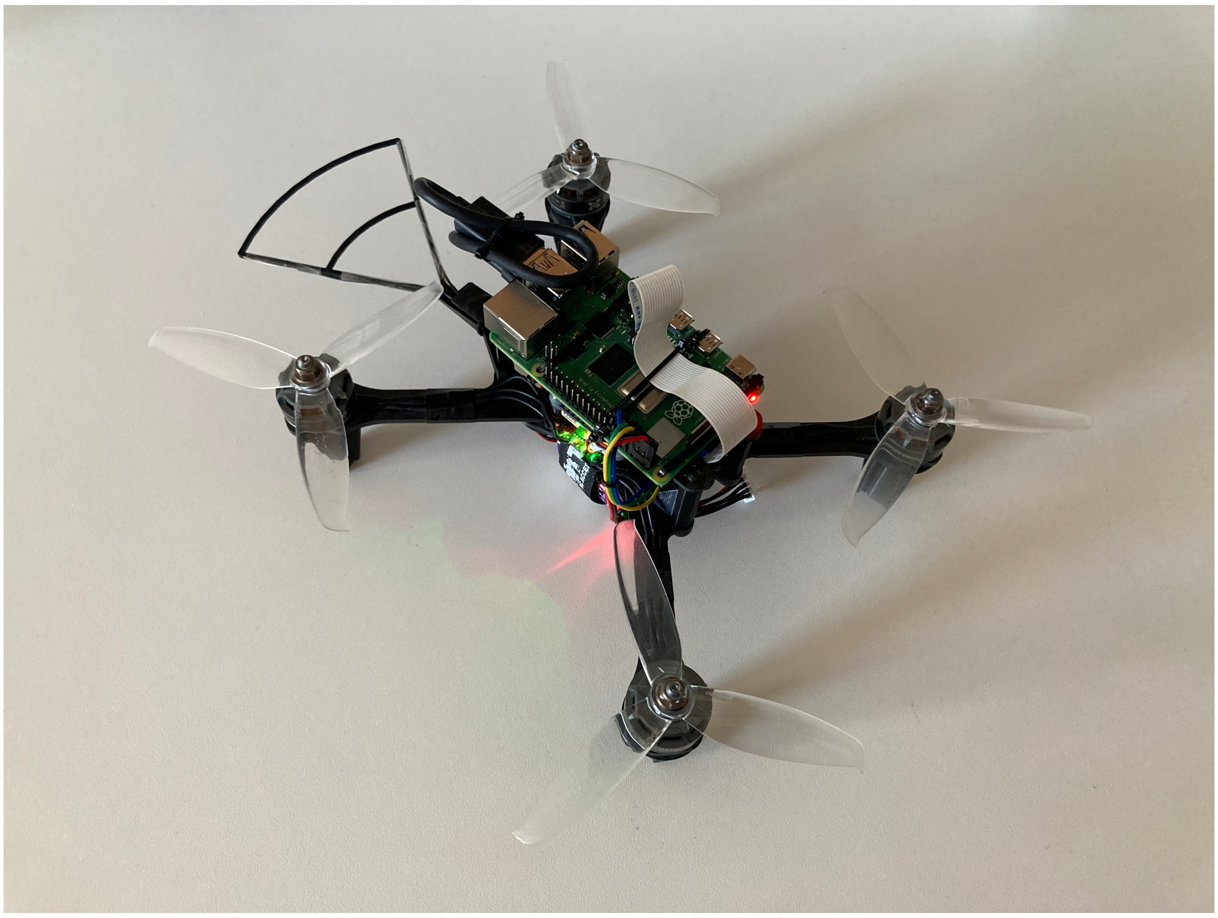


Figure B.3: Photo of drone from top

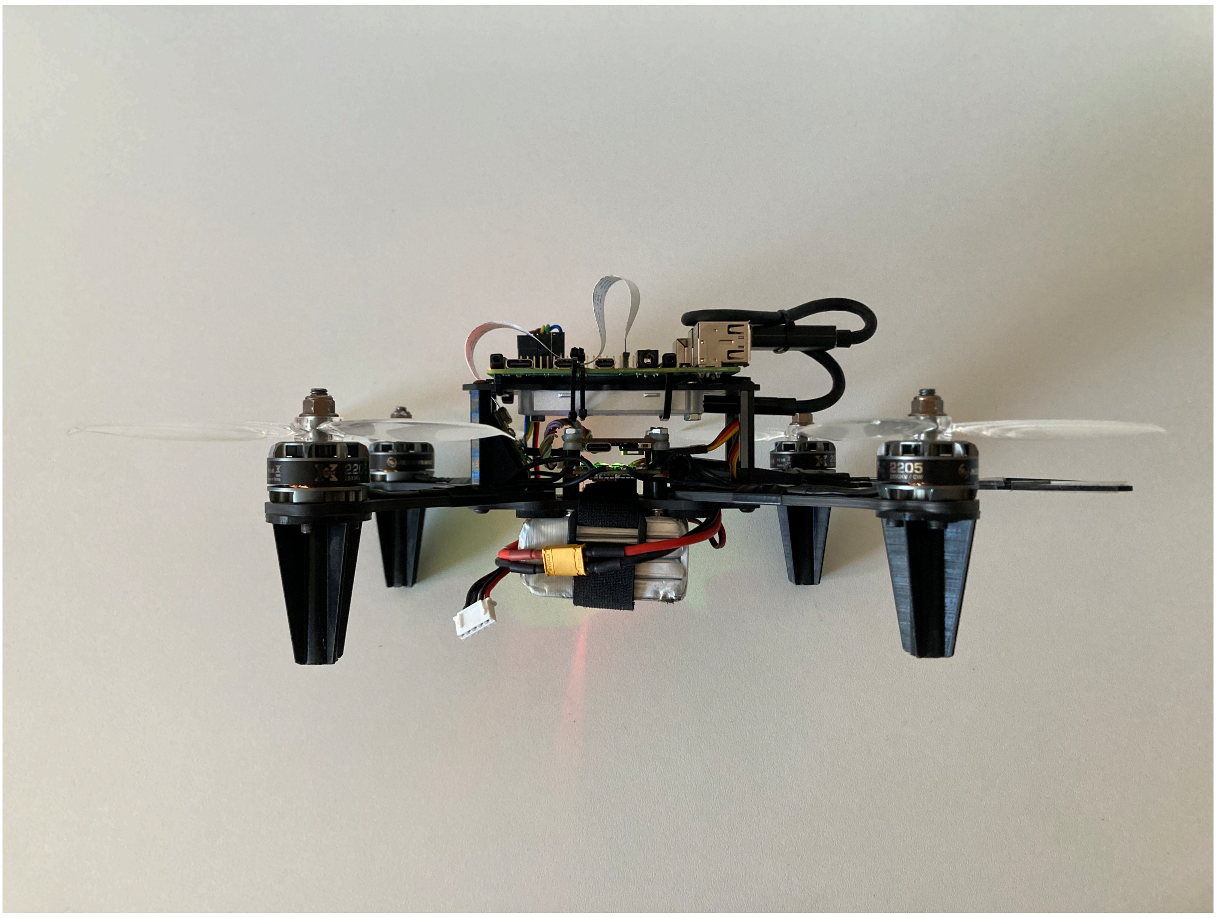


Figure B.4: Photo of drone from left

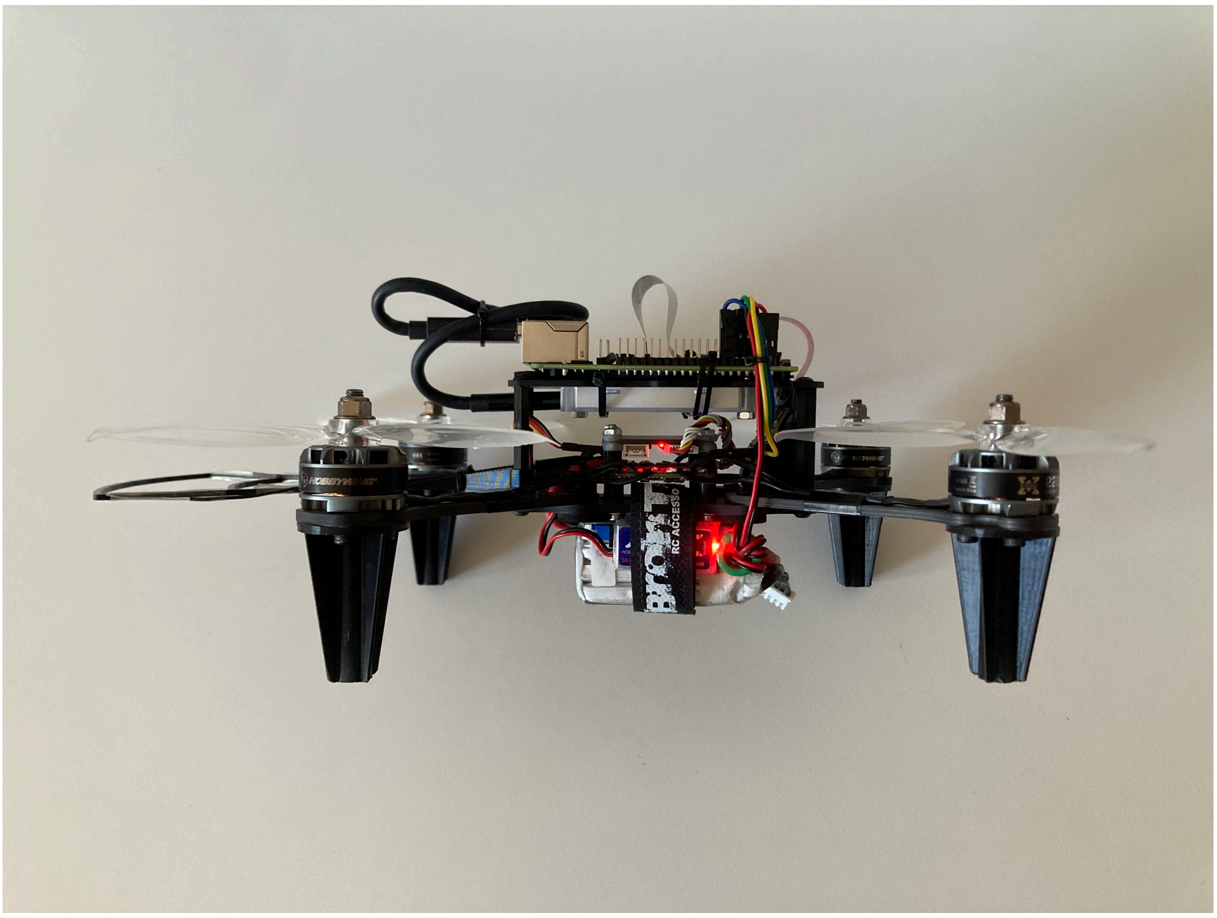


Figure B.5: Photo of drone from right

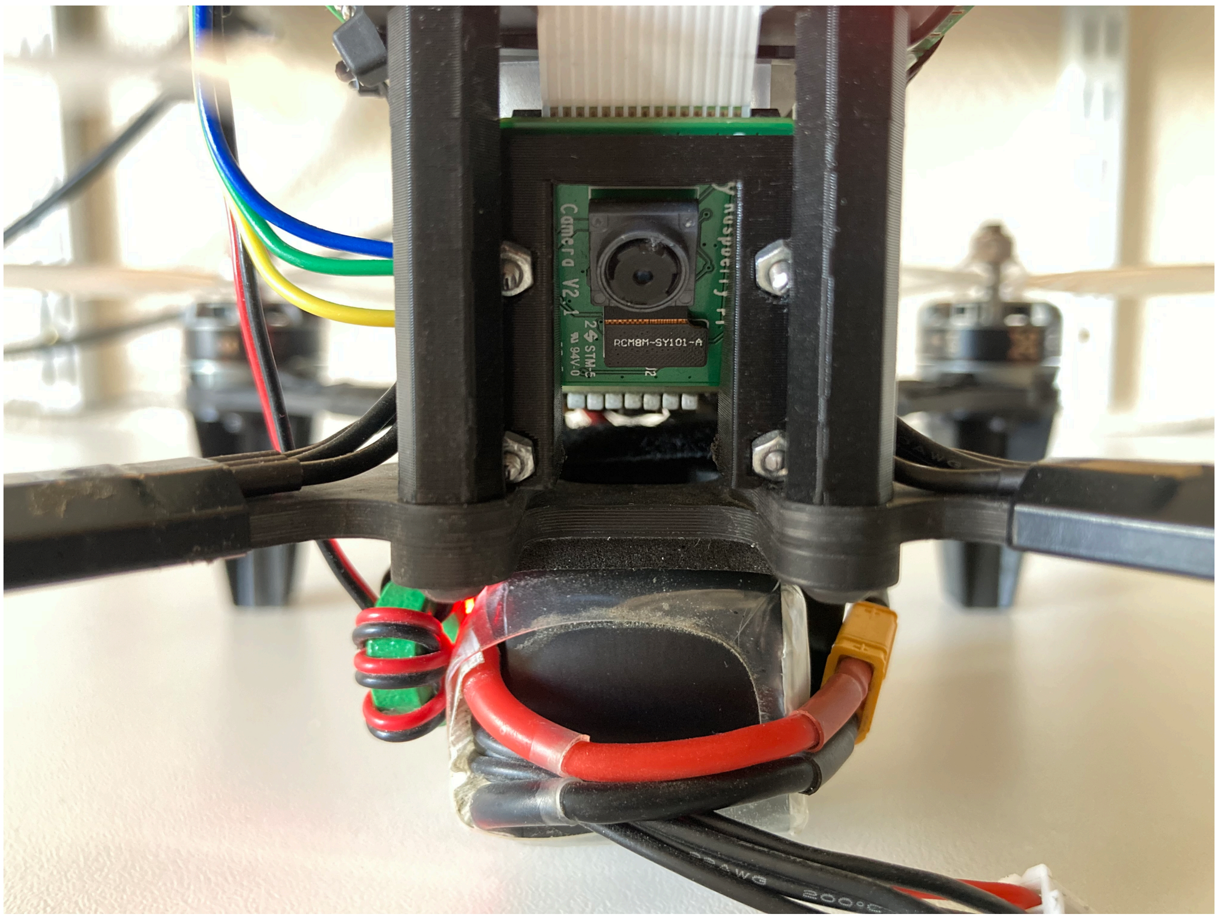


Figure B.6: Photo of drone from front

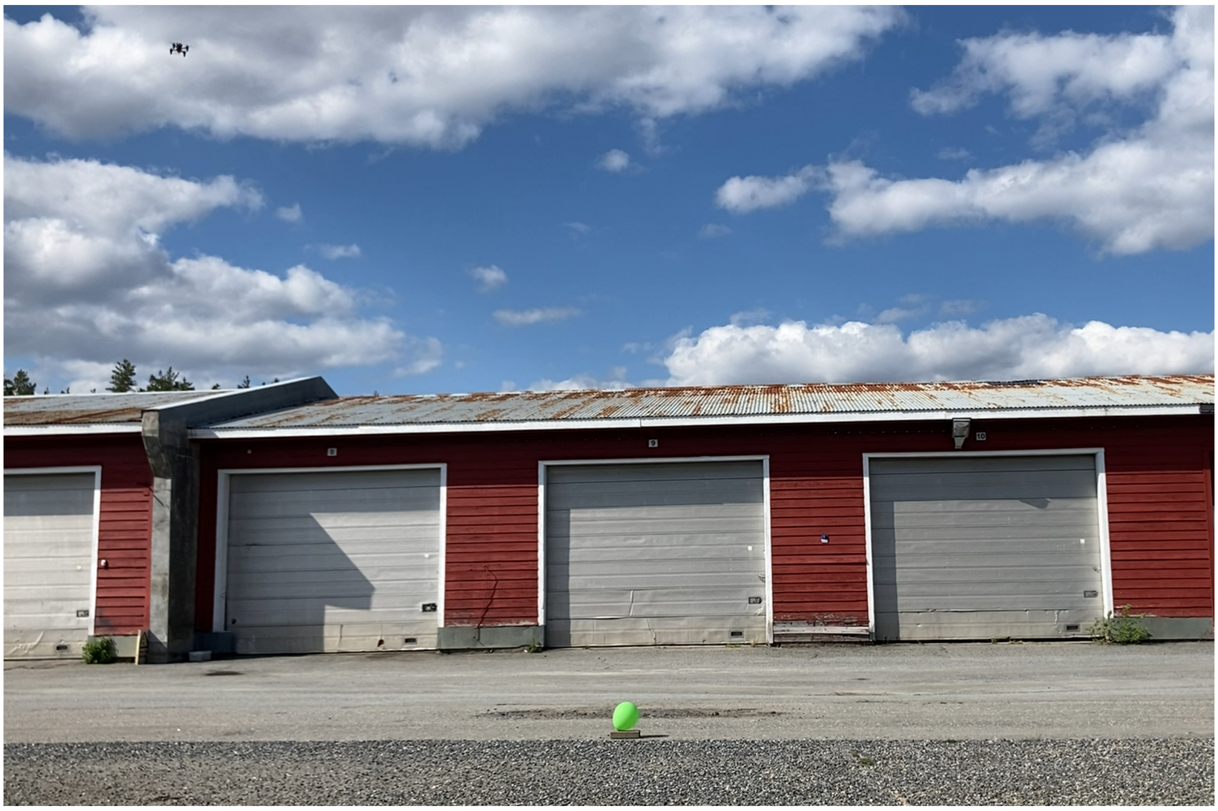


Figure B.7: Photo of drone hovering in the air

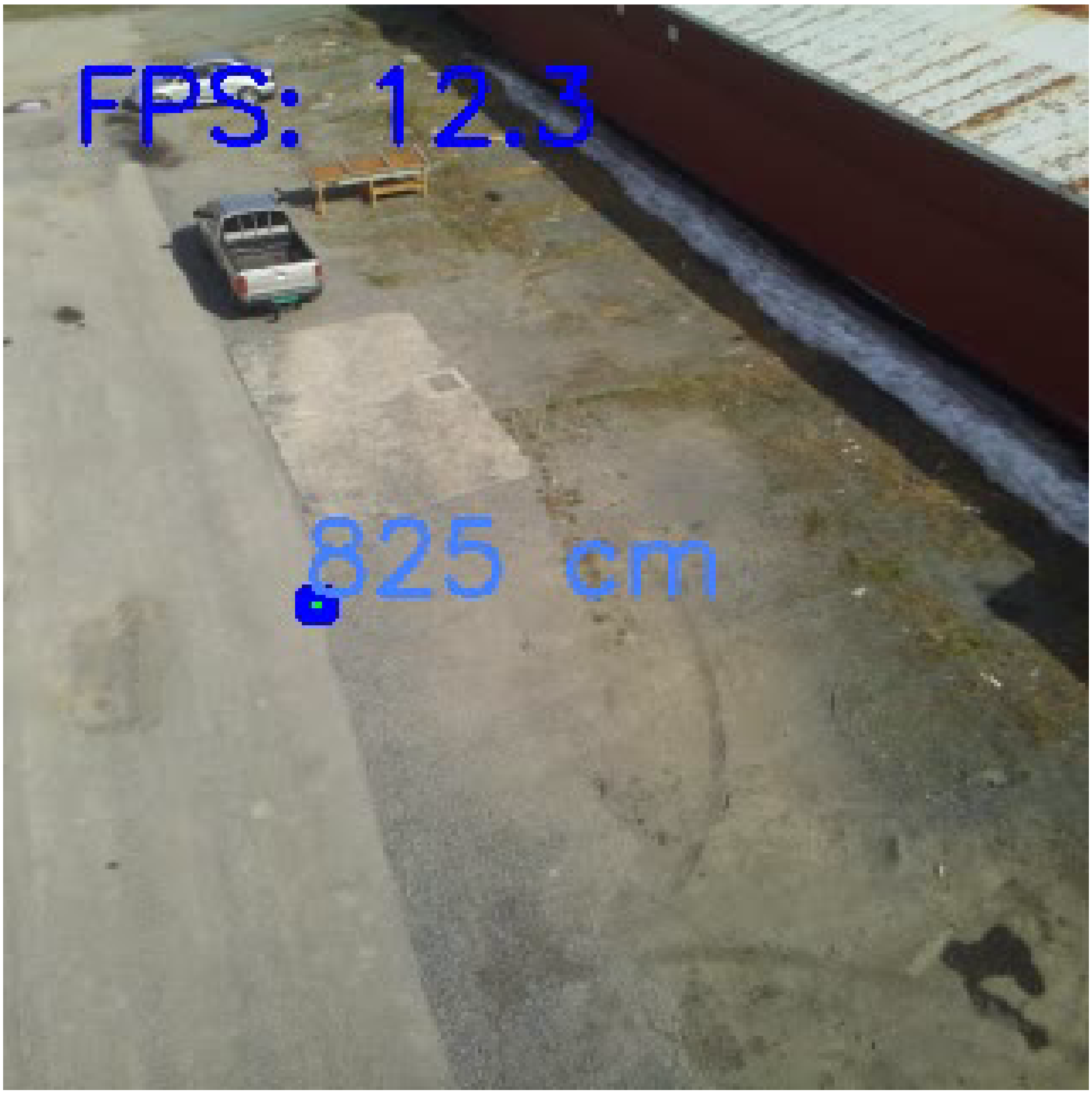


Figure B.8: Photo taken from drone's perspective doing object detection

Appendix C

Config4 Code Explanation and Analysis

C.1 Blob detection.v1

Since we use the Raspberry Pi Camera Module v3 and not a depth camera, we need to find a way to find the distance to a known object. We ended up using OpenCV and cvzone. Our approach involves detecting a specific color in the video frames, identifying the contours of the detected object, and then calculating the distance based on the object's apparent size in the image.

First, we import the necessary packages for image processing, numerical operations, and color detection:

```python
import cv2
import cvzone
from cvzone.ColorModule import ColorFinder
import numpy as np
```

We create a VideoCapture object (OpenCV) to read video frames from the camera [33], and set the capture dimensions to 640x480 pixels [33]:

```python
cap = cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)
```

Next, we create a ColorFinder object (cvzone) with automatic color range update turned off [34], and define the color range for detection (using predetermined HSV values):

```python
myColorFinder = ColorFinder(False)
hsvVals = {'hmin': 97, 'smin': 21, 'vmin': 23, 'hmax': 125, 'smax': 255, '
    vmax': 193}
```

In a continuous loop, we read the video frames from the camera (OpenCV) [33], detect the specified color range in the image using the ColorFinder object (cvzone) [34], and find the contours in the binary mask (cvzone) [34]:

```python
while True:
    success, img = cap.read()
    imgColor, mask = myColorFinder.update(img, hsvVals)
    imgContour, contours = cvzone.findContours(img, mask)
```

When contours are detected, we extract data from the first contour (assumed to be the object of interest) and calculate the distance to the object based on its apparent size in the image. We use the real-world dimensions of the object (in this case, a tennis ball with a width of 6.5 cm) and the camera's focal length to compute the distance:

```python
if contours:
    data = contours[0]['center'][0], h - contours[0]['center'][1], int(
    contours[0]['area'])

    f = 535  # focal length of the camera
    W = 6.5  # real-world width of the tennis ball

    w = np.sqrt(contours[0]['area']/np.pi) * 2  # width of the tennis ball
    in the image
    d = (W * f) / w  # calculate the distance

    print(d)
```

Finally, we display the distance in centimeters on the image (cvzone) [34] and stack the original image, detected color image, binary mask, and contour image for visualization (cvzone) [34] this is done for testing purpose:

```python
cvzone.putTextRect(img, f'depth: {int(d)} cm', (contours[0]['center'][0] -
   75, contours[0]['center'][1] - 50), scale= 2)
imgStack = cvzone.stackImages([img, imgColor, mask, imgContour], 2, 0.5)
cv2.imshow("Image", imgStack)
cv2.waitKey(1)
```

This approach is suitable for objects with known dimensions and relatively uniform color distribution. This testing was used with simple color detection. We can use this function to use the contours of trained models to find the distance as well.

C.2 Blob detection.v2

This code uses OpenCV [33] and cvzone [34] to track and measure the distance of 3 tennis-sized balls with different colors in a live video feed from a webcam.

First, we import the necessary packages:

```python
import cv2 as cv
from cvzone.FPS import FPS
import imutils
import math
import numpy as np
```

This function `calculateDistance` is used to calculate the distance from the camera to the object based on the radius of the object in pixels. This calculation is based on similar triangles and the field of view of the camera.

```python
# Function to calculate distance from the object based on its radius in
    pixels
def calculateDistance(ballRadius_px):
    return int(faktor / ballRadius_px)
```

This function `detect_colored_object` uses color-based filtering and shape-based detection to identify and locate the colored ball in the video frame [61]. The function uses color filtering to create a mask for pixels within the defined color range and applies the Hough Transform function [55] on a blurred grayscale version of the frame to detect circles. If a detected circle's center lies within the color mask, the function returns the circle's coordinates and radius[58].

```python
# Function to detect a colored object within a given color range and size
def detect_colored_object(colorLower, colorUpper, min_radius, max_radius):
    mask = cv.inRange(hsv, colorLower, colorUpper)

    # Erode the mask to remove noise & Dilate the mask to fill gaps
    mask = cv.erode(mask, None, iterations=2)
    mask = cv.dilate(mask, None, iterations=2)

    # Convert the frame to grayscale and apply median blur
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    gray = cv.medianBlur(gray, 5)

    # Detect circles using the Hough transform
    circles = cv.HoughCircles(gray, cv.HOUGH_GRADIENT, 1, 20, param1=100,
    param2=30,
                               minRadius=min_radius, maxRadius=max_radius)

    # Check if any circles were detected
    if circles is not None:
        circles = np.uint16(np.around(circles))

        for circle in circles[0, :]:
            x, y, radius = circle

            # Check if the circle's center is within the mask's boundaries
            if 0 <= x < mask.shape[1] and 0 <= y < mask.shape[0] and mask[y, x] > 0:
                return (x, y, radius)
    return None
```

The function `display_object_info` visualizes [56] details such as a circle around the detected object, x, and y coordinates, and estimated distance on the video frame.

```python
# Function to display information about the detected object on the frame
def display_object_info(frame, x, y, radius, distance, color, text_offset):
    if x is not None and y is not None:
        # Draw a circle around the detected object
        cv.circle(frame, (x, y), radius, color, 2)

        # Display the coordinates x and y
        coordinates_text = f"X: {x}, Y: {y}"

        # Display the distance text
        distance_text = f"Distance: {distance} cm"

        # Put the coordinate text on the frame
        cv.putText(frame, coordinates_text, (x + 10, y), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

        # Put the distance text on the frame
        cv.putText(frame, distance_text, (22, 70 + text_offset), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
```

The `ballRadius` is a variable indicating the radius of the tennis ball, the `cameraFOV` is the camera's field of view, and lastly, the `faktor` is a calculation parameter [59] used to determine the distance of the object from the camera.

```python
ballRadius = 3.25   # cm (radius of the ball)
cameraFOV = 62.2    # degrees (field of view of the camera)
faktor = (1280 / 2) * (ballRadius / math.tan(math.radians(kameraFOV / 2))) # Pixels from center to edge divided by minimum distance from the lens
```

For each color, the program attempts to detect an object of that color in the frame [57]. If an object is found, its properties (coordinates, distance) are updated in the color dictionary. The updated information is then displayed [56] on the frame. This process repeats for each frame, enabling real-time tracking.

```python
# Main loop
while True:
    (grabbed, frame) = videoCap.read() # Read a frame from the video capture
    fps, img = fpsreader.update(frame, color=(255, 0, 0)) # Update the FPS overlay on the frame
    frame = imutils.resize(frame, width=1280) # Resize the frame
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV) # Convert the frame to HSV format

    # Initialize x, y, and ballRadius_px values for each color
    for color_info in colors.values():
        color_info['x'] = None
        color_info['y'] = None
        color_info['ballRadius_px'] = None

    # Iterate through the defined colors and detect objects
    for idx, (color_name, color_info) in enumerate(colors.items()):
        # Call the detect_colored_object function to find objects in the frame
        obj = detect_colored_object(color_info['lower'], color_info['upper'], color_info['min_radius'],
                                    color_info['max_radius'])

        if obj: # If an object is detected, get the coordinates and radius, distance
            x, y, ballRadius_px = obj
            distance = kalkulerDistanse(ballRadius_px)

            # Update the color_info dictionary with the new values
            color_info['x'] = x
            color_info['y'] = y
            color_info['ballRadius_px'] = ballRadius_px
            color_info['distance'] = distance
```

The `display_object_info` function visually [56] presents data about the detected object on the video feed.

```
# Display information about the detected object on the frame
        display_object_info(frame, color_info['x'], color_info['y'], color_info['ballRadius_px'],
                            color_info.get('distance'), color_info['color'], color_info['text_offset'])

    cv.imshow("Frame", frame)  # Show the frame
```

Referring back to the discussion in the 'Configuration 4 Journey' section, it was clear that the computational demands of our program far exceeded the capabilities of the hardware in use, given that the development and testing were conducted on a high-end laptop utilizing a webcam. As a result, we needed to revisit our approach and develop Blob detection.v3.

C.3 Blob detection.v3

By simplifying and removing some of the more computationally intense functionalities from the program we developed Blob detection.v3.

First, we import the necessary packages:

```python
import cv2
import cvzone
from cvzone.ColorModule import ColorFinder
from cvzone.FPS import FPS
import numpy as np
```

The `is_circle` function checks if a shape is circular enough to be considered a circle. If the shape is ”round enough” (the roundness is more than the given threshold, which is set to 0.6), it filters out shapes that are not ”round enough”.

```python
# Function to check if a contour is a circle
def is_circle(cnt, threshold=0.6):
    area = cv2.contourArea(cnt) # Compute the area of the contour
    perimeter = cv2.arcLength(cnt, True) # Compute the perimeter of the contour
    if perimeter == 0: # If the perimeter is zero, it cannot be a circle, return False
        return False
    circularity = 4 * np.pi * area / (perimeter * perimeter) # Calculate the circularity of the contour
    return circularity >= threshold
```

Next, we create a `ColorFinder` object (cvzone) with automatic color range update turned off [34], and define the color range for detection using predetermined HSV (Hue, Saturation, Value) values:

```python
myColorFinder: ColorFinder = ColorFinder(False)

# Define the HSV color range for blue, green and orange
hsvValsBlue = {'hmin': 105, 'smin': 168, 'vmin': 119, 'hmax': 111, 'smax': 255, 'vmax': 255} #blue
hsvValsGreen = {'hmin': 76, 'smin': 29, 'vmin': 132, 'hmax': 97, 'smax': 124, 'vmax': 255} #green
hsvValsOrange = {'hmin': 0, 'smin': 120, 'vmin': 120, 'hmax': 20, 'smax': 255, 'vmax': 255} #orange
```

Each frame from the camera (OpenCV) [33], is read and processed for frame rate, which at first is analyzed to find areas that match the color profiles defined for blue, green, and orange in HSVvalues[34], and then analyze these colored areas further to identify contours in those regions. These contours are checked for circularity using the `is_circle` function, thereby extracting contours that are likely to be circles.

```
    # Update the color detection for blue, green and orange
    imgColorBlue, mask = myColorFinder.update(img, hsvValsBlue)
    imgColorGreen, maskRed = myColorFinder.update(img, hsvValsGreen)
    imgColorOrange, maskOrange = myColorFinder.update(img, hsvValsOrange)

    # Find contours in blue, green and orange color mask
    imgContourBlue, contours = cvzone.findContours(img, mask)
    imgContourGreen, contoursRed = cvzone.findContours(img, maskRed)
    imgContourOrange, contoursOrange = cvzone.findContours(img, maskOrange)

    # Filter contours that are circles
    circular_contours_blue = [cnt for cnt in contours if is_circle(cnt['cnt'])]
    circular_contours_green = [cnt for cnt in contoursRed if is_circle(cnt['cnt'])]
    circular_contours_orange = [cnt for cnt in contoursOrange if is_circle(cnt['cnt'])]
```

Processes and displays the depth, x-position, and y-position for each detected ball. It iterates over three lists containing circular contours for each color, and checks if there are circular contours detected. If so, it selects the first contour and calculates the depth of the ball using its area and predefined values for focal length and ball width [22]. It then prints the color of the ball and its depth.

```
# Process and display depth, x, and y position for each ball
    for color, circular_contours_list in zip(['blue', 'green', 'orange'],
                                             [circular_contours_blue, circular_contours_green, circular_contours_orange]):
        if circular_contours_list:
            cnt = circular_contours_list[0]
            data = cnt['center'][0], h - cnt['center'][1], int(cnt['area'])

            f = 474 #focal length of the camera
            W = 6.5 # real-world width of the tennis ball
            w = np.sqrt(cnt['area'] / np.pi) * 2 # width of the tennis ball in the image
            d = (W * f) / w # calculate the distance
            print(f"{color}: {d}")
```

The depth is displayed using `cvzone.putTextRect` function[34] which adds the text ”depth: int(d) cm” to the `imgContourBlue` image. The x and y position is also displayed using `cvzone.putTextRect`. `ImgStack` is a stack of the original image, color image with a blue mask, binary mask, and contour image, which is then shown with `cv2.imshow`[34].

```
            # Display depth on the frame
            cvzone.putTextRect(imgContourBlue, f'depth: {int(d)} cm',
                               (cnt['center'][0] - 75, cnt['center'][1] - 50), scale=2)
            # Display x and y position on the frame with more space between depth and position
            cvzone.putTextRect(imgContourBlue, f'x: {cnt["center"][0]}, y: {cnt["center"][1]}',
                               (cnt['center'][0] - 75, cnt['center'][1] - 10), scale=1.5)

    imgStack = cvzone.stackImages([img, imgColorBlue, mask, imgContourBlue], 2, 0.5)
    cv2.imshow("Image", imgStack)
```

This version offered a significant improvement in frame rate performance and detection and was further tested on Rasberry Pi.

C.4 Blob Detection.v3 Evaluation

We first import the necessary packages to enable image processing, numerical operations, and color detection:

```python
import cv2
import cvzone
from cvzone.ColorModule import ColorFinder
import numpy as np
import xml.etree.ElementTree as ET
import os
import csv
```

The function `is_circle` is created to check if a detected contour is circular or not. We calculate the area and the perimeter of the contour and use these values to calculate the circularity. A threshold is set to determine whether the contour is circular or not:

```python
def is_circle(cnt, threshold=0.6):
    area = cv2.contourArea(cnt)
    perimeter = cv2.arcLength(cnt, True)
    if perimeter == 0:
        return False
    circularity = 4 * np.pi * area / (perimeter * perimeter)
    return circularity >= threshold
```

The `parse_label` function is designed to parse label information from an XML file. It extracts the name and bounding box coordinates of an object:

```python
def parse_label(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()

    label = {}
    for obj in root.findall('object'):
        name = obj.find('name').text
        box = obj.find('bndbox')
        xmin = int(box.find('xmin').text)
        xmax = int(box.find('xmax').text)
        ymin = int(box.find('ymin').text)
        ymax = int(box.find('ymax').text)

        label[name] = [(xmin, ymin, xmax, ymax)]
    return label
```

We create the `calculate_f1_score` function to compute the F1 score given precision and recall:[46][p. 156]

```python
def calculate_f1_score(precision, recall):
    if precision + recall == 0:  # to avoid division by zero
        return 0
    else:
        f1_score = 2 * (precision * recall) / (precision + recall)
        return f1_score
```

Next, we define the `read_image` function that simply reads an image from a given path using the OpenCV function `imread`:

```python
def read_image(image_file):
    return cv2.imread(image_file)
```

In the `detect_color` function, we detect colors in an image using a pre-defined HSV value range. We then use cvzone's findContours function to find the contours in the binary mask. A filter is applied to only select contours that pass the is_circle test:

```python
def detect_color(image):
    myColorFinder = ColorFinder(False)
    #hsvVals = {'hmin': 49, 'smin': 69, 'vmin': 17, 'hmax': 108, 'smax':
    255, 'vmax': 181} #green
    #hsvVals = {'hmin': 0, 'smin': 42, 'vmin': 0, 'hmax': 20, 'smax': 186,
    'vmax': 219} #red
    hsvVals = {'hmin': 87, 'smin': 78, 'vmin': 0, 'hmax': 114, 'smax': 195,
     'vmax': 174} #blue
    imgColor, mask = myColorFinder.update(image, hsvVals)
    imgContour, contours = cvzone.findContours(image, mask)

    circular_contours = [cnt for cnt in contours if is_circle(cnt['cnt'])]

    results = []
    if circular_contours:
        for cnt in circular_contours:
            x, y, w, h = cv2.boundingRect(cnt['cnt'])
            results.append(('blue', (x, y, x+w, y+h)))
    return results
```

The `calculate_iou` function is designed to compute the Intersection over Union (IoU) between two bounding boxes. This metric is commonly used in computer vision tasks to evaluate the accuracy of object detection models: [84]

```python
def calculate_iou(box1, box2):
    x1, y1, w1, h1 = box1
    x2, y2, w2, h2 = box2

    xi1 = max(x1, x2)
    yi1 = max(y1, y2)
    xi2 = min(x1 + w1, x2 + w2)
    yi2 = min(y1 + h1, y2 + h2)

    inter_area = max(xi2 - xi1, 0) * max(yi2 - yi1, 0)

    box1_area = w1 * h1
    box2_area = w2 * h2
    union_area = box1_area + box2_area - inter_area

    return inter_area / union_area if union_area > 0 else 0
```

We define the `calculate_precision_recall` function to compute precision and recall based on predictions and ground truth data. For each prediction, we calculate the maximum IoU with all ground truth boxes of the same color. If this maximum IoU is greater than or equal to a set IoU threshold, it's considered a true positive; otherwise, it's a false positive. False negatives are counted as those ground truth boxes that don't have any corresponding prediction with an IoU greater than or equal to the threshold:[46][p. 155-156]

```python
def calculate_precision_recall(predictions, ground_truth, iou_threshold=0.5):
    TP = FP = FN = 0

    for image_predictions, image_ground_truth in zip(predictions, ground_truth):
        for pred_color, pred_box in image_predictions:
            if pred_color in image_ground_truth:
                ious = [calculate_iou(pred_box, truth_box) for truth_box in image_ground_truth[pred_color]]
                max_iou = max(ious) if ious else 0

                if max_iou >= iou_threshold:
                    TP += 1
                else:
                    FP += 1
            else:
                FP += 1

        for truth_color, truth_boxes in image_ground_truth.items():
            if truth_color not in [pred_color for pred_color, _ in image_predictions]:
                FN += len(truth_boxes)
            else:
                for truth_box in truth_boxes:
                    ious = [calculate_iou(pred_box, truth_box) for pred_color, pred_box in image_predictions if pred_color == truth_color]
                    max_iou = max(ious) if ious else 0
                    if max_iou < iou_threshold:
                        FN += 1

    precision = TP / (TP + FP) if TP + FP > 0 else 0
    recall = TP / (TP + FN) if TP + FN > 0 else 0

    return precision, recall
```

In our main procedure, we define paths to our image and label directories, then sort and pair up the corresponding image and label files:

```
image_dir = '/home/vaffe/RandomStuff/dataset/valid/blue/'
label_dir = '/home/vaffe/RandomStuff/dataset/valid/blue/labels/'

image_files = sorted(os.listdir(image_dir))
label_files = sorted(os.listdir(label_dir))
```

The code processes each image file and corresponding label file one by one. It reads the image, detects the color (our function returns the bounding boxes of detected objects), and parses the XML label file to obtain the ground truth bounding boxes. The predictions and ground truth are stored for later evaluation:

```
for image_file, label_file in zip(image_files, label_files):
    image_path = os.path.join(image_dir, image_file)
    label_path = os.path.join(label_dir, label_file)

    image = read_image(image_path)
    label = parse_label(label_path)

    prediction = detect_color(image)

    predictions.append(prediction)
    ground_truths.append(label)

    print('Predicted: ', prediction)
    print('Ground Truth: ', label)
```

Finally, we calculate precision, recall, and F1 score, which are commonly used metrics to evaluate the performance of object detection models:

```
precision, recall = calculate_precision_recall(predictions, ground_truths)
f1_score = calculate_f1_score(precision, recall)

print('Precision: ', precision)
print('Recall: ', recall)
print('F1 Score: ', f1_score)
```

The precision metric quantifies the number of correct positive predictions made, while recall (also known as sensitivity) quantifies the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive). The F1 Score is the harmonic mean of precision and recall and provides a single score that balances both the concerns of precision and recall in one number. [46][p. 155-156]

Appendix D

Config4 Source code

D.1 Blob Detection.v1

```python
import cv2
import cvzone
from cvzone.ColorModule import ColorFinder
from cvzone.FPS import FPS
import numpy as np

fpsreader = FPS()
cap = cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)

success, img = cap.read()
h, w, _ = img.shape

myColorFinder = ColorFinder(False)
hsvVals = {'hmin': 36, 'smin': 29, 'vmin': 44, 'hmax': 90, 'smax': 150, '
    vmax': 187}

while True:
    success, img = cap.read()
    fps, img = fpsreader.update(img)
    imgColor, mask = myColorFinder.update(img, hsvVals)
    imgContour, contours = cvzone.findContours(img, mask)

    if contours:
        data = contours[0]['center'][0], h - contours[0]['center'][1], int(
    contours[0]['area'])

        f = 535
        W = 6.5
        w = np.sqrt(contours[0]['area'] / np.pi) * 2
        d = (W * f) / w

```

```
            x, y = contours[0]['center'][0], contours[0]['center'][1]

            print("x: ", x)
            print("y: ", y)

            print(fps)

            cvzone.putTextRect(img, f'depth: {int(d)} cm', (contours[0]['center
        '][0] - 75, contours[0]['center'][1] - 50),
                               scale=2)
            cv2.putText(img, f'x: {int(x)}, y: {int(y)}', (20, h - 600), cv2.
        FONT_HERSHEY_SIMPLEX, 1.5, (255, 0, 0),
                        thickness=2)

        imgStack = cvzone.stackImages([img, imgColor, mask, imgContour], 2,
                                      0.5)
        cv2.imshow("Image", imgStack)

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

D.2 Blob Detection.v2

```
import cv2 as cv
from cvzone.FPS import FPS
import imutils
import math
import numpy as np

def calculateDistance(ballRadius_px):
    return int(faktor / ballRadius_px)

def detect_colored_object(colorLower, colorUpper, min_radius, max_radius):

    mask = cv.inRange(hsv, colorLower, colorUpper)
    mask = cv.erode(mask, None, iterations=2)
    mask = cv.dilate(mask, None, iterations=2)

    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    gray = cv.medianBlur(gray, 5)

    circles = cv.HoughCircles(gray, cv.HOUGH_GRADIENT, 1, 20, param1=100,
    param2=30,
                              minRadius=min_radius, maxRadius=max_radius)

    if circles is not None:
        circles = np.uint16(np.around(circles))

        for circle in circles[0, :]:
            x, y, radius = circle

            if 0 <= x < mask.shape[1] and 0 <= y < mask.shape[0] and mask[y, x] > 0:
                return (x, y, radius)
    return None

def display_object_info(frame, x, y, radius, distance, color, text_offset):
    if x is not None and y is not None:

        cv.circle(frame, (x, y), radius, color, 2)
        coordinates_text = f"X: {x}, Y: {y}"
        distance_text = f"Distance: {distance} cm"

        cv.putText(frame, coordinates_text, (x + 10, y), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
        cv.putText(frame, distance_text, (22, 70 + text_offset), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

ballRadius = 3.25   # cm (radius of the ball)
cameraFOV = 62.2    # degrees (field of view of the camera)
faktor = (1280 / 2) * (ballRadius / math.tan(math.radians(kameraFOV / 2)))

colors = {
    'green': {
        'lower': (72, 70, 32),
        'upper': (99, 244, 107),
        'min_radius': 0, #ex between 0
        'max_radius': 0, #to ex 60 pixels
        'color': (0, 255, 0),
        'text_offset': 0, #Distance text position under FPS
    },
    'orange': {
        'lower': (0, 115, 99),
        'upper': (18, 255, 255),
        'min_radius': 0,
        'max_radius': 0,
        'color': (0, 102, 255),
        'text_offset': 20,
    },
    'red': {
        'lower': (119, 37, 0),
        'upper': (179, 179, 147),
        'min_radius': 0,
        'max_radius': 0,
        'color': (0, 0, 255),
        'text_offset': 40,
    },
}

fpsreader = FPS()
videoCap = cv.VideoCapture(0)
videoCap.set(3, 1280)
videoCap.set(4, 720)

while True:
    (grabbed, frame) = videoCap.read()
    fps, img = fpsreader.update(frame, color=(255, 0, 0))
    frame = imutils.resize(frame, width=1280)
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)

    for color_info in colors.values():
        color_info['x'] = None
        color_info['y'] = None
        color_info['ballRadius_px'] = None

    for idx, (color_name, color_info) in enumerate(colors.items()):
        obj = detect_colored_object(color_info['lower'], color_info['upper'
    ], color_info['min_radius'],
                                    color_info['max_radius'])
        if obj:
            x, y, ballRadius_px = obj
            distance = kalkulerDistanse(ballRadius_px)

            color_info['x'] = x
            color_info['y'] = y
            color_info['ballRadius_px'] = ballRadius_px
            color_info['distance'] = distance

        display_object_info(frame, color_info['x'], color_info['y'],
    color_info['ballRadius_px'],
                            color_info.get('distance'), color_info['color'
    ], color_info['text_offset'])

    cv.imshow("Frame", frame)
    key = cv.waitKey(1)
    if key == ord("q"):
        break
```

D.3 Blob Detection.v3

```
import cv2
import cvzone
from cvzone.ColorModule import ColorFinder
from cvzone.FPS import FPS
import numpy as np
```

```python

def is_circle(cnt, threshold=0.6):
    area = cv2.contourArea(cnt)
    perimeter = cv2.arcLength(cnt, True)
    if perimeter == 0:
        return False
    circularity = 4 * np.pi * area / (perimeter * perimeter)
    return circularity >= threshold

fpsreader = FPS()

cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

ret, frame = cap.read()
h, w, _ = frame.shape

myColorFinder: ColorFinder = ColorFinder(False)
hsvValsBlue = {'hmin': 105, 'smin': 168, 'vmin': 119, 'hmax': 111, 'smax': 255, 'vmax': 255}
hsvValsGreen = {'hmin': 76, 'smin': 29, 'vmin': 132, 'hmax': 97, 'smax': 124, 'vmax': 255}
hsvValsOrange = {'hmin': 0, 'smin': 120, 'vmin': 120, 'hmax': 20, 'smax': 255, 'vmax': 255}

while True:
    ret, frame = cap.read()
    fps, img = fpsreader.update(frame)

    imgColorBlue, mask = myColorFinder.update(img, hsvValsBlue)
    imgColorGreen, maskRed = myColorFinder.update(img, hsvValsGreen)
    imgColorOrange, maskOrange = myColorFinder.update(img, hsvValsOrange)

    imgContourBlue, contours = cvzone.findContours(img, mask)
    imgContourGreen, contoursRed = cvzone.findContours(img, maskRed)
    imgContourOrange, contoursOrange = cvzone.findContours(img, maskOrange)

    circular_contours_blue = [cnt for cnt in contours if is_circle(cnt['cnt'])]
    circular_contours_green = [cnt for cnt in contoursRed if is_circle(cnt['cnt'])]
    circular_contours_orange = [cnt for cnt in contoursOrange if is_circle(cnt['cnt'])]

    for color, circular_contours_list in zip(['blue', 'green', 'orange'],
                                             [circular_contours_blue, circular_contours_green, circular_contours_orange]):
        if circular_contours_list:
            cnt = circular_contours_list[0]
            data = cnt['center'][0], h - cnt['center'][1], int(cnt['area'])

            f = 474
            W = 6.5
            w = np.sqrt(cnt['area'] / np.pi) * 2
            d = (W * f) / w
            print(f"{color}: {d}")

            cvzone.putTextRect(imgContourBlue, f'depth: {int(d)} cm',
                               (cnt['center'][0] - 75, cnt['center'][1] - 50), scale=2)
            cvzone.putTextRect(imgContourBlue, f'x: {cnt["center"][0]}, y: {cnt["center"][1]}',
                               (cnt['center'][0] - 75, cnt['center'][1] - 10), scale=1.5)

    imgStack = cvzone.stackImages([img, imgColorBlue, mask, imgContourBlue], 2, 0.5)
    cv2.imshow("Image", imgStack)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Appendix E

ROS1 to ROS2

E.1 Translating from ROS1 to ROS2

The following documentation describes our process of translating from ROS1 to ROS2. It's important to note that this translation does not represent an optimal approach. Although improvements could be implemented to enhance the efficiency and readability of the code, the aim of this translation was to ensure compatibility with ROS2, maintaining performance, and providing satisfactory results in the tests we conducted. This document also assumes that the reader have some knowledge about ROS2 or 1.

E.1.1 Syntax Changes

The syntax of ROS1 and ROS2, while largely similar, exhibits several distinctions primarily associated with changes in the architectural structure of scripts. Among the differences, library usage stands out:

```
#ROS1
import rospy
import rosgraph

#ROS2
import rclpy
from rclpy.node import Node
```

In ROS1, rospy and rosgraph serve as the key libraries for node creation and computation graph visualization. Transitioning to ROS2, rclpy replaces these libraries, providing a Python API for ROS2 interactions, with the Node class encapsulating a node within the ROS graph.

Moreover, modifications were also made to logging functions. Despite the changes, the functions retain a similar operational manner:

```
#ROS1
rospy.loginfo()

#ROS2
get_logger().info()
```

E.1.2 Architecture

The most significant modifications involved the creation and architecture of the ROS nodes. In ROS2, a class is established that inherits from the rclpy node class. Within this class, the node is named, and subscribers, publishers, and other necessary variables for the node are added. Subsequently, this node is initialized within the main function:

```
#ROS2
class Qualisys_node(Node):

    def __init__(self):
        super().__init__('qualisys_node')

        self.pub = self.create_publisher(DronePose, 'drone_pose', 10)
```

All the functions that the class will utilize are defined within it, incorporating the `self` parameter for invocations within the class:

```
#ROS2
class Qualisys_node(Node):

    def __init__(self):
    def talker(self, data):
    def create_msg(self,x,y,z,v_x,v_y,v_z,a_x,a_y,a_z,yaw,v_yaw,freq,
   full_msg):
    def calculate_vel_a(self,position,freq, yaw):
    def calc(self,position,rot,freq):
    def create_body_index(self, xml_string):
    def get_freq(self, xml_string):
    async def qtmMain(self):
```

This stands in contrast to the ROS1 architecture, where an overarching class to manage the functions does not exist. Instead, a main function is utilized that accepts the publisher as a parameter:

```
#ROS1
async def mainQualisys(pub):
```

In ROS1, this main function for the node is what runs indefinitely. Unlike in ROS2, where a class object is initiated to run endlessly. Our modifications to the loops were minor and mainly involved creating variables and tasks:

```python
#ROS1
asyncio.ensure_future(mainQualisys(pub))
asyncio.get_event_loop().run_forever()

#ROS2
loop = asyncio.get_event_loop()
qtm_task = loop.create_task(qtm_node.qtmMain())
...
asyncio.ensure_future(qtm_task)
loop.run_forever()
```

Although the main functions in ROS1 and ROS2 might appear different, they operate similarly in that the node is initialized within the main function. However, in ROS1, subscribers and publishers are also created within the main:

```python
#ROS1
if __name__ == "__main__":
    rospy.init_node('qualisys', anonymous=True) #Node initialization
    pub = rospy.Publisher('/drone_controller/current_pos', DronePose,
   queue_size=10) #create publisher

#ROS2
def main(args=None):
    rclpy.init(args=args) #Node initialization
    qtm_node = Qualisys_node() #Creates Node
```

Upon executing the code with these modifications, we encountered some issues with the data received from Qualisys. It was not in the same format as expected by the `create_msg` function. To resolve this issue, we initialized the values outside the class, which corrected the problem of acquiring initial values from Qualisys:

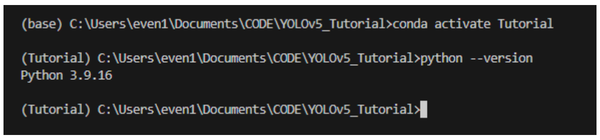
```python
#ROS2
if prev_msg is None:
    prev_msg = DronePose()
    prev_msg.pos.x = 0.0
    prev_msg.pos.y = 0.0
    prev_msg.pos.z = 0.0
    prev_msg.vel.x = 0.0
    prev_msg.vel.y = 0.0
    prev_msg.vel.z = 0.0
    prev_msg.yaw.data = 0.0

if prev_vel is None:
    prev_vel = [0.0, 0.0, 0.0]
```

Appendix F

Yolo training Tutorial

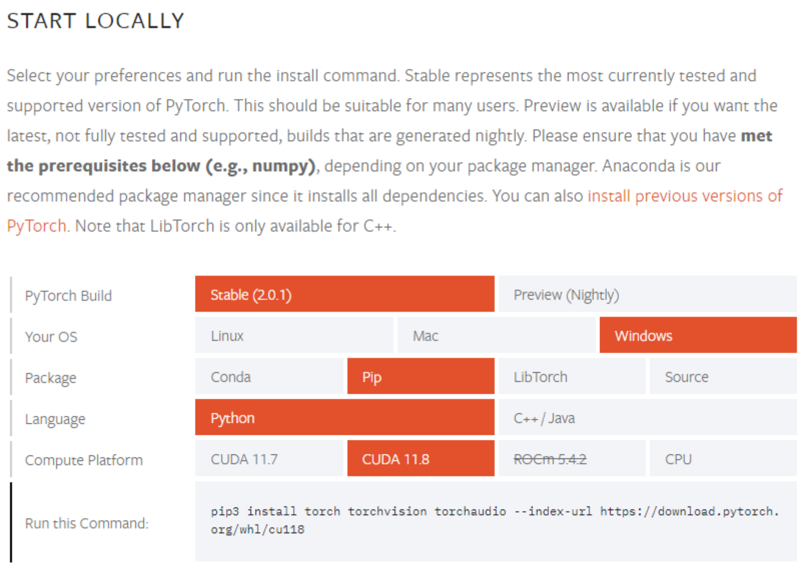
Prerequisites

A computer with an NVIDIA graphics card that is CUDA compatible (RTX 2070 was used in this study). Although you can train with a CPU, it will be significantly slower. A virtual environment with Python 3.8 – 3.11 (Python 3.9.16 was used in this study).



```
(base) C:\Users\even1\Documents\CODE\YOLOv5_Tutorial>conda activate Tutorial

(Tutorial) C:\Users\even1\Documents\CODE\YOLOv5_Tutorial>python --version
Python 3.9.16

(Tutorial) C:\Users\even1\Documents\CODE\YOLOv5_Tutorial>
```

1. Install Pytorch: Start by installing the correct version of Pytorch. Visit Pytorch's website and select the version that suits your hardware. If your GPU supports CUDA, select "compute platform" CUDA 11.8. If your GPU is not supported, opt for the CPU version.

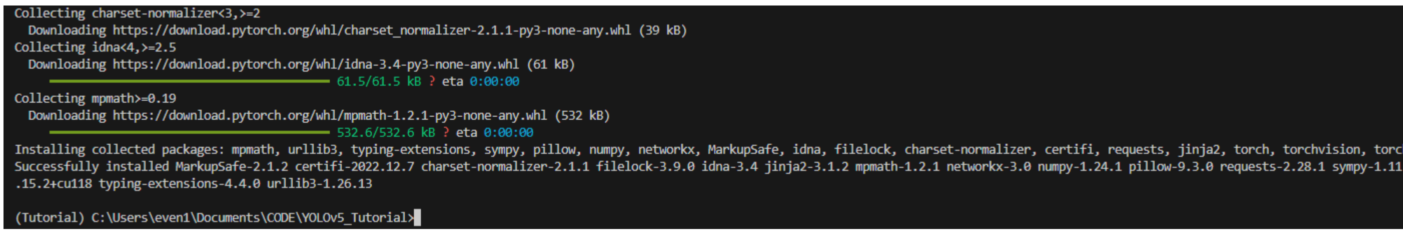


START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

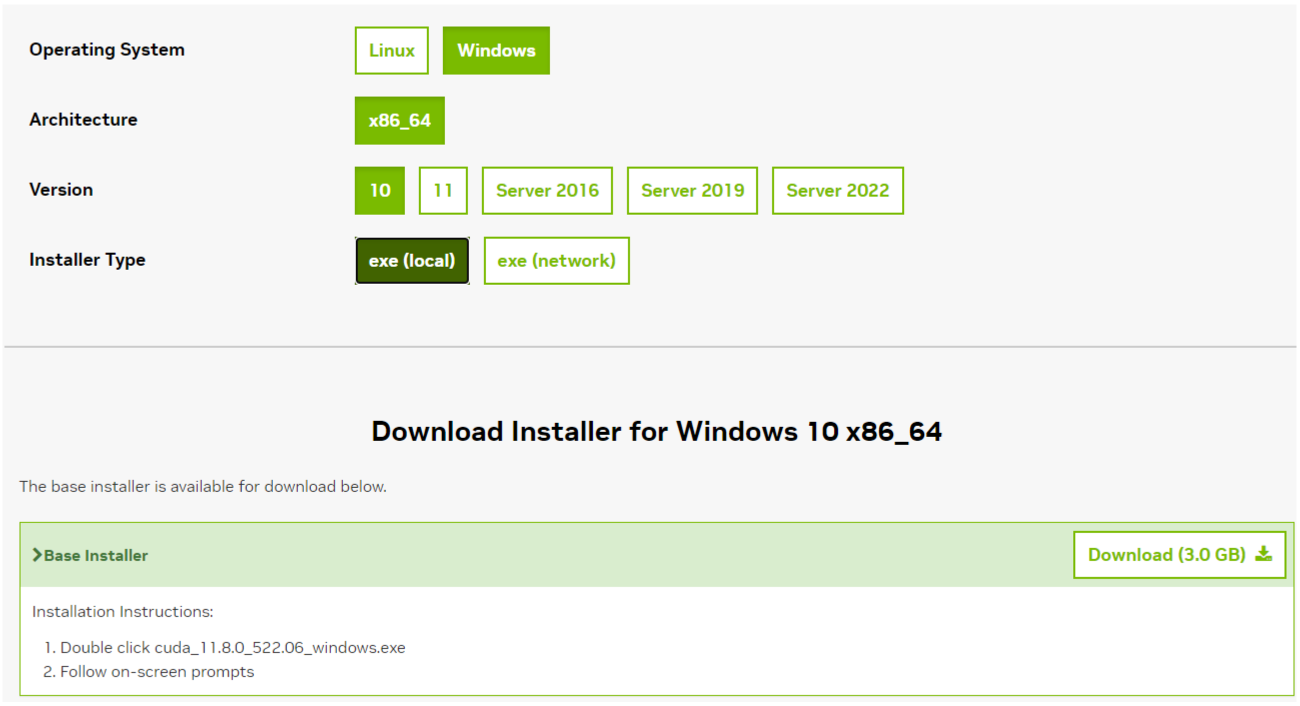
PyTorch Build	[✓ Stable (2.0.1)]	Preview (Nightly)		
Your OS	Linux	Mac	[✓ Windows]	
Package	Conda	[✓ Pip]	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	CUDA 11.7	[✓ CUDA 11.8]	~~ROCm 5.4.2~~	CPU
Run this Command:	pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118			

2. Run the Pytorch Command: Execute the provided command in your Python environment. This might take some time, as Pytorch will download and install all necessary packages.



```
Collecting charset-normalizer<3,>=2
  Downloading https://download.pytorch.org/whl/charset_normalizer-2.1.1-py3-none-any.whl (39 kB)
Collecting idna<4,>=2.5
  Downloading https://download.pytorch.org/whl/idna-3.4-py3-none-any.whl (61 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 61.5/61.5 kB ? eta 0:00:00
Collecting mpmath>=0.19
  Downloading https://download.pytorch.org/whl/mpmath-1.2.1-py3-none-any.whl (532 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 532.6/532.6 kB ? eta 0:00:00
Installing collected packages: mpmath, urllib3, typing-extensions, sympy, pillow, numpy, networkx, MarkupSafe, idna, filelock, charset-normalizer, certifi, requests, jinja2, torch, torchvision, torc
Successfully installed MarkupSafe-2.1.2 certifi-2022.12.7 charset-normalizer-2.1.1 filelock-3.9.0 idna-3.4 jinja2-3.1.2 mpmath-1.2.1 networkx-3.0 numpy-1.24.1 pillow-9.3.0 requests-2.28.1 sympy-1.11
.15.2+cu118 typing-extensions-4.4.0 urllib3-1.26.13

(Tutorial) C:\Users\even1\Documents\CODE\YOLOv5_Tutorial>
```

3. Download and Install the Correct CUDA Version.



Operating System Linux [✓ Windows]

Architecture [✓ x86_64]

Version [✓ 10] 11 Server 2016 Server 2019 Server 2022

Installer Type [✓ exe (local)] exe (network)

Download Installer for Windows 10 x86_64

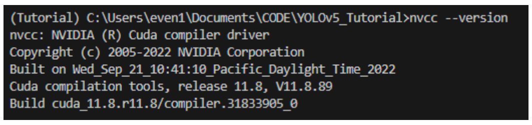
The base installer is available for download below.

>Base Installer Download (3.0 GB)

Installation Instructions:

1. Double click cuda_11.8.0_522.06_windows.exe
2. Follow on-screen prompts

4. Verify CUDA Installation: Follow the steps in the installer. After installation, use the command nvcc –version to ensure that CUDA has been installed correctly.



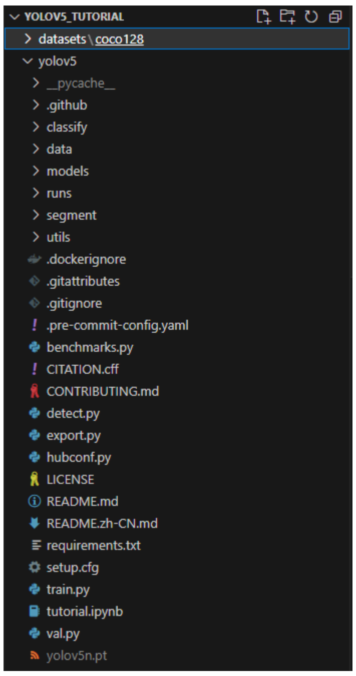
```
(Tutorial) C:\Users\even1\Documents\CODE\YOLOv5_Tutorial>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:41:10_Pacific_Daylight_Time_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

5. Clone the YOLO Repository and Install Requirements.

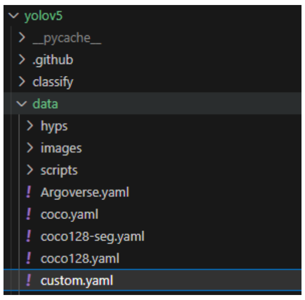
git clone https://github.com/ultralytics/yolov5
cd yolov5
pip install -r requirements.txt

6. We recommend running the following script once to train on the COCO dataset, as this will create the correct folder structure and download the necessary weights: python

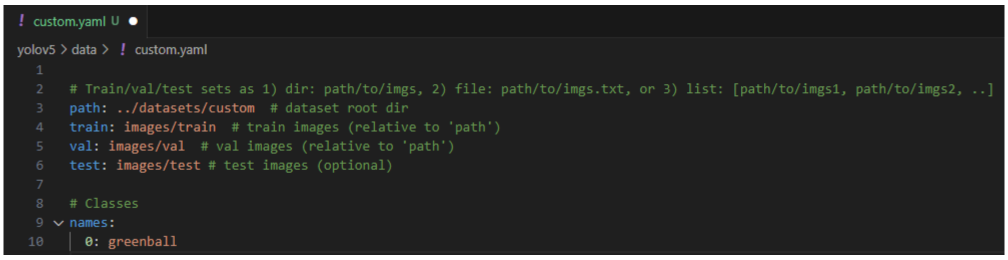
train.py –img 640 –epochs 3 –data coco128.yaml –weights yolov5n.pt


```
YOLOV5_TUTORIAL
> datasets \ coco128
v yolov5
  > __pycache__
  > .github
  > classify
  > data
  > models
  > runs
  > segment
  > utils
  .dockerignore
  .gitattributes
  .gitignore
  .pre-commit-config.yaml
  benchmarks.py
  CITATION.cff
  CONTRIBUTING.md
  detect.py
  export.py
  hubconf.py
  LICENSE
  README.md
  README.zh-CN.md
  requirements.txt
  setup.cfg
  train.py
  tutorial.ipynb
  val.py
  yolov5n.pt
```

7. Duplicate and Rename the YAML File: Make a copy of the coco128.yaml file found under yolov5/data/ and rename it to match your dataset. This file instructs the training command where to find the dataset and corresponding labels.


```
v yolov5
  > __pycache__
  > .github
  > classify
  v data
    > hyps
    > images
    > scripts
    Argoverse.yaml
    coco.yaml
    coco128-seg.yaml
    coco128.yaml
    custom.yaml
```

8. Modify the YAML File: Adjust the file to meet your requirements. For instance, in this study, we only have one class, named ”greenball”.



```yaml

# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]
path: ../datasets/custom  # dataset root dir
train: images/train  # train images (relative to 'path')
val: images/val  # val images (relative to 'path')
test: images/test # test images (optional)

# Classes
names:
  0: greenball
```

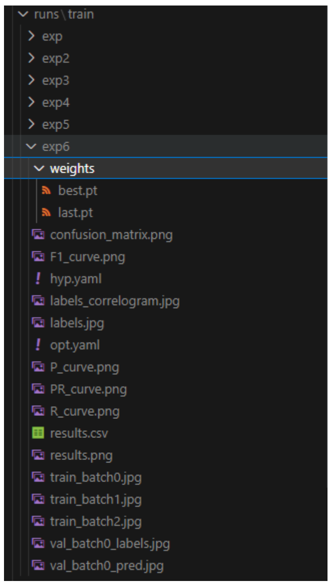
10. You're now ready to begin training your YOLO model on your custom data. Decide on the model you want to use for transfer learning. We recommend either yolov5n (nano) or yolov5s (small), as larger models result in slower frame rates.

python train.py –img 640 –epochs 3 –data custom.yaml –weights yolov5n.pt –device 0

Ensure that the YAML file selected is the custom one you created earlier.

Choose the number of epochs for training; a range of 500 to 1000 is recommended. The img 640 parameter determines the size to which the image is scaled (640 is YOLO's default).
To execute the process on CPU, remove the –device 0 parameter.

11. Examine Training Results: Upon completion of training, you'll find all related materials saved under the runs/train/ directory. The most recent training session will be in the latest exp folder. This folder contains the retrained weights and general statistics from the training process, including plots that visualize the training progress.



```
runs\train
  exp
  exp2
  exp3
  exp4
  exp5
  exp6
    weights
      best.pt
      last.pt
    confusion_matrix.png
    F1_curve.png
    hyp.yaml
    labels_correlogram.jpg
    labels.jpg
    opt.yaml
    P_curve.png
    PR_curve.png
    R_curve.png
    results.csv
    results.png
    train_batch0.jpg
    train_batch1.jpg
    train_batch2.jpg
    val_batch0_labels.jpg
    val_batch0_pred.jpg
```

12. After successfully training your YOLO model, you can test it in real-time using your webcam. Run the following command:

python detect.py –weights <path to weights> –source 0

Replace <path to weights> with the path to your trained weights file. The –source 0 specifies that the webcam (usually denoted by '0' in most systems) should be used as the input source.

Appendix G

Source code, ROS2 nodes

G.1 Source code, ROS2 nodes

G.1.1 Image processing, configuration 1

Video capture node

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
import gi
gi.require_version("Gst", "1.0")
from gi.repository import Gst, GObject
import time

class ImagePublisher(Node):
    def __init__(self):
        super().__init__('image_publisher')
        self.publisher = self.create_publisher(Image, 'video_frames', 10)
        timer_period = 0.000166
        self.timer = self.create_timer(timer_period, self.timer_callback)
        Gst.init(None)
        # Configure a GStreamer pipeline to capture from a jetson.
        # nvarguscamerasrc sensor-id=0 select which camera sensor to capture from
        # video/x-raw(memory:NVMM) sets the output of the source to raw video stored in NVIDIA's proprietary NVMM format.
        # the width and the height is set to the expected input to the model
        # video/x-raw, format=BGRx converts the NVMM-formatted video to raw input
        # appsink drop=True parameter means that if the pipeline is running faster than it can handle, it will drop frames to
      maintain performance.
        self.pipeline = ("nvarguscamerasrc sensor-id=0 ! "
                                    "video/x-raw(memory:NVMM), width=640, height=640, framerate=30/1, format=NV12 ! "
                                    "nvvidconv flip-method=0 ! "
                                    "video/x-raw, width=640, height=640, format=BGRx ! "
                                    "videoconvert ! "
                                    "video/x-raw, format=BGR ! appsink drop=True")
        self.cap = cv2.VideoCapture(self.pipeline, cv2.CAP_GSTREAMER)
        self.br = CvBridge()
        self.timer_start = time.time()
        self.timer_end = time.time()
        self.period = 0
        self.fps = 0

    def timer_callback(self):
        ret, frame = self.cap.read()
        self.timer_end = time.time()
        self.period = self.timer_end - self.timer_start
        self.fps = 1/self.period
        self.timer_start = time.time()
        # CV bridge wraps the image array in ROS image message
        self.publisher.publish(self.br.cv2_to_imgmsg(frame))

        self.get_logger().info('Video frame published')


```

```python



def main(args=None):
    # Initialize ROS client library
    rclpy.init(args=args)
    image_publisher = ImagePublisher()
    rclpy.spin(image_publisher)
    image_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Object detection node

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import Int32MultiArray
from cv_bridge import CvBridge
import cv2
import numpy as np
import time
from .yoloDet import YoloTRT


# Absolute path to used library and tensorRT model
model = YoloTRT(library="/home/gruppe6/ros2_ws/src/config1/config1/yolov5/build/libmyplugins.so",
                engine="/home/gruppe6/ros2_ws/src/config1/config1/yolov5/build/best2.engine", conf=0.5, yolo_ver="v5")



class Detect(Node):
    def __init__(self):
        # Instantiate parent class object (Node)
        super().__init__('detect')
        # Create subscription to video_frames topic
        self.subscription = self.create_subscription(Image, 'video_frames', self.listener_callback, 10)
        # Create publisher to object_pos_and_distance topic
        self.publisher = self.create_publisher(Int32MultiArray, 'object_pos_and_distance', 10)
        # cv_bridge i s used to extract the image array from the ROS image msg
        self.br = CvBridge()

        self.frame_count = 0
        self.period_timer_start = time.time()
        self.period_timer_end = time.time()
        self.fps = 0



    # Callback for video_frames subscription , called every
    # time a new video frame i s recieved
    def listener_callback(self, data):
        self.frame_count += 1
        self.period_timer_end = time.time()
        self.timer_period = self.period_timer_end - self.period_timer_start
        self.fps = 1 / self.timer_period
        self.period_timer_start = time.time()
        self.get_logger().info('Recieving video frame, current FPS: {:.2f}'.format(self.fps))
        current_frame = self.br.imgmsg_to_cv2(data)
        detections, t = model.Inference(current_frame)
        for obj in detections:
            # print(obj['class'], obj['conf'], obj['center'], obj['width'])
            cx = int(obj['center'][0])
            cy = int(obj['center'][1])
            f = 434
            W = 6.5
            w = obj['width']
            distance = int((W*f)/w)

            # creating a ros2 int32multiarray
            msg = Int32MultiArray()
            msg.data = [cx, cy, distance]
            self.publisher.publish(msg)



def main(args=None):
    # Initialize ROS client library
    rclpy.init(args=args)
    # Instantiate the detect node
    detect = Detect()
    # Spin node forever
    rclpy.spin(detect)

    detect.destroy_node()
    rclpy.shutdown()


if (__name__ == "__main__"):
    main()
```

G.1.2 Image processing, configuration 2/3

Video capture node

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
from picamera2 import Picamera2

width = 384
height = width

class VideoCapture(Node):
    def __init__(self):
        super().__init__('video_capture')
        self.publisher = self.create_publisher(Image, 'video_frames', 10)
        # Timer callback is triggered 60 times per second
        timer_period = 1/60
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.picam2 = Picamera2()
        # Configure picamera2 to capture video with the format and size expected by the model.
        # This way we avoid having to reformat and resize later
        # raw is set to 1640x1232 to force high FOV sensor mode.
        # FrameDurationLimits is the min and max frame period in microseconds, 40000us limits capture
        # to 25 FPS
        self.picam2.configure(self.picam2.create_video_configuration(main={"format": 'RGB888', "size": (width, height)},
                                                    raw={"size": (1640,1232)},
                                                    controls={"FrameDurationLimits": (40000, 40000)}))
        self.picam2.start()
        self.br = CvBridge()

    def timer_callback(self):
        frame = self.picam2.capture_array()

        # Cv bridge wraps the image array in a ROS image message
        self.publisher.publish(self.br.cv2_to_imgmsg(frame))
        self.get_logger().info('Video frame published')


def main(args=None):
    # Initialize ROS client library
    rclpy.init(args=args)

    # Instantiate the video_capture node
    video_capture = VideoCapture()

    # Spin node forever
    rclpy.spin(video_capture)
    video_capture.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Object detection node

Appendix G Source code, ROS2 nodes

```python

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import Int32MultiArray
from cv_bridge import CvBridge
import cv2
import numpy as np
import tflite_runtime.interpreter as tflite
import time
from picamera2 import Picamera2



# Absolute path to tflite model
model ='/home/gruppe6/models/edl1_1k_edgetpu.tflite'

# Load edgetpu runtime shared library
tpu_interpreter = tflite.Interpreter(model, experimental_delegates=[
    tflite.load_delegate('libedgetpu.so.1.0')])

cpu_interpreter = tflite.Interpreter(model)
# We discard detections with lower confidence score than the threshold
threshold = 0.80

class Detect(Node):
    def __init__(self):
        # Instantiate parent class object (Node)
        super().__init__('detect')

        # Create subscription to video_frames topic
        self.subscription = self.create_subscription(Image, 'video_frames', self.listener_callback, 10)

        # Create publisher to object_pos_and_distance topic
        self.publisher = self.create_publisher(Int32MultiArray, 'object_pos_and_distance', 10)

        # cv_bridge is used to extract the image array from the ROS image msg
        self.br = CvBridge()

        # Selected interpreter, change to cpu_interpreter to run
        # model on cpu only
        self.interpreter = tpu_interpreter

        # Allocate tensors, must be called to start inference
        self.interpreter.allocate_tensors()

        # Input details for the loaded model
        self.input_details = self.interpreter.get_input_details()

        # Output details for the loaded model
        self.output_details = self.interpreter.get_output_details()

        self.period_timer_start = time.time()
        self.period_timer_end = time.time()
        self.fps = 0
        # Multiplier used for distance calculation
        self.focal_length_multiplier = 847 # pi camera 2

        self.ball_real_diameter = 6.5 # cm
        # ROS2 message type, used to publish to 'object pos and distance'
        self.output_array = Int32MultiArray()

        # Get image dimensions expected by the model
        self.width = self.input_details[0]['shape'][1]
        self.height = self.input_details[0]['shape'][2]

    # Returns False if val is outside cutoff, used to discard
    # distance measurements at the edges of the frame
    def constrain_detection(self, val, frame_dim, cutoff):
        return val > cutoff and val < (frame_dim - cutoff)

    # Callback for video_frames subscription, called every
    # time a new video frame is recieved
    def listener_callback(self, data):
        self.period_timer_end = time.time()
        self.timer_period = self.period_timer_end - self.period_timer_start
        self.fps = 1 / self.timer_period
        self.period_timer_start = time.time()
        self.get_logger().info('Current FPS: {:.2f}'.format(self.fps))
        current_frame = self.br.imgmsg_to_cv2(data)


        # Add batch dimension expected by the model,
        # new shape = [1, WIDTH, HEIGHT, 3]
        input_data = np.expand_dims(current_frame, axis=0)

        # Set input_data as the model input
        self.interpreter.set_tensor(self.input_details[0]['index'],
                                    input_data)

        # Start inference by invoking the interpreter
        self.interpreter.invoke()

        # boxes, classes, scores and number of detections are the
        # available model outputs
        # We are only using boxes and scores in this node

        # output_details returns the tensor index needed by get_tensor()
        boxes = self.interpreter.get_tensor(self.output_details[1]['index'])[0]
        scores = self.interpreter.get_tensor(self.output_details[0]['index'])[0]




        # Loop over all detections
        for i in range(len(scores)):
            if ((scores[i] > threshold) and (scores[i] <= 1.0)):
                # Get corner coordinates of bounding box
                x1, x2 = int(boxes[i][1] * self.width) , int(boxes[i][3] * self.width)
                y1, y2 = int(boxes[i][0] * self.height), int(boxes[i][2] * self.height)

                w, h = x2 - x1, y2 - y1
                # Get center of bounding box
                cx, cy = (int(x1 + 0.5*w),int(y1+0.5*h))

                # Only send info on the best score
                if (scores[i] > best_score):
                    best_score = scores[i]
                    cx_out = cx
                    cy_out = cy
                    distance_out = int((self.dist_multiplier)/w)

                # Because distance calculation is based on width, we need to discard
                # detections at the left and right edges as the width may be cropped
                if (self.constrain_detection(cx, self.width, 30)):
                    dist = int((self.ball_real_diameter * self.focal_length_multiplier) / w)
                    self.output_array.data = [cx_out, cy_out, distance_out]
                    self.publisher.publish(self.output_array)
                else:
                    self.output_array.data = [cx_out, cy_out, -1]
                    self.publisher.publish(self.output_array)
            # No detections
            else:
                self.output_array.data = [-1, -1, -1]
                self.publisher.publish(self.output_array)



def main(args=None):
    # Initialize ROS client library
    rclpy.init(args=args)

    # Instantiate the detect node
    detect = Detect()

    # Spin node forever
    rclpy.spin(detect)

    detect.destroy_node()
    rclpy.shutdown()


if (__name__ == "__main__"):
    main()
```

G.1.3 Object follower

Controller node

```

import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32MultiArray
from .object_follower import ObjectFollower
```

```python



class Controller(Node):
    def __init__(self):
        super().__init__('controller')
        self.subscription = self.create_subscription(Int32MultiArray, 'object_pos_and_distance', self.listener_callback, 10)
        self.publisher = self.create_publisher(Int32MultiArray, 'yaw_thrust_pitch', 10)
        # Instantiate an ObjectFollower object
        # PIDs must be set up by editing the constructor in object_follower.py
        self.object_follower = ObjectFollower()

    # Callback for the 'object_pos_and_distance' subscription
    def listener_callback(self, msg_in):
        # Feed values from 'object_pos_and_distance' to the object follower
        self.object_follower(x=msg_in.data[0], y=msg_in.data[1], distance=msg_in.data[2])
        msg_out = Int32MultiArray()
        # Publish object follower outputs to 'yaw_thrust_pitch' topic
        msg_out.data = [self.object_follower.yaw_out, self.object_follower.thrust_out, self.object_follower.pitch_out ]
        self.publisher.publish(msg_out)


def main(args=None):
    rclpy.init(args=args)
    controller = Controller()
    rclpy.spin(controller)
    controller.destroy_node()
    rclpy.shutdown()


if (__name__ == "__main__"):
    main()
```

Object follower class

```python

from .pid import PID




class ObjectFollower:
    # Setpoints for yaw and thrust PIDs are 160 to keep detected
    # object in the center of a 320x320 frame.
    # Pitch setpoint of 70 to keep detected object at 70 cm distance
    def __init__(self):
        self.yaw_pid = PID(160, 0.5, 0.0, 0.0, -1000, 1000)
        self.thrust_pid = PID(160, 0.5, 0.0, 0.0, 300, 700)
        self.pitch_pid = PID(70, 0.5, 0.0, 0.0, -1000, 1000)
        self.yaw_out = 0
        self.thrust_out = 0
        self.pitch_out = 0

    def __call__(self, x, y, distance):
        if (x != -1):
            self.yaw_out = int(self.yaw_pid(x))
            self.thrust_out = int(self.thrust_pid(y))
            self.pitch_out = int(self.pitch_pid(distance))
        # if x = -1 no object is being detected, drone should
        # move slowly towards ground with 300 thrust (500 is neutral thrust)
        else:
            self.yaw_out = 0
            self.thrust_out = 300
            self.pitch_out = 0

    def tune_yaw(self, kp, ki, kd):
        self.yaw_pid.tune(kp, ki, kd)

    def tune_thrust(self, kp, ki, kd):
        self.thrust_pid.tune(kp, ki, kd)

    def tune_pitch(self, kp, ki, kd):
        self.pitch_pid.tune(kp, ki, kd)
```

PID class

```

import time




def clamp(val: float, lower_limit: float, upper_limit: float):
    return lower_limit if val <= lower_limit else upper_limit if val > upper_limit else val

class PID:

    def __init__(self,
                 setpoint: float = 0.0,
                 kp: float = 0.0,
                 ki: float = 0.0,
                 kd: float = 0.0,
                 min_output: float = None,
                 max_output: float = None,
                 dt_min: float = 0.01):


        # setpoint: PID controller setpoint
        # kp: proportional gain constant
        # ki: integral gain constant
        # kd: derivative gain constant
        # dt_min: minimum time between error corrections

        self.setpoint = setpoint
        self.kp, self.ki, self.kd = kp, ki, kd
        self.dt_min = dt_min
        self.min_output = min_output
        self.max_output = max_output

        self.p_out: float = 0.0
        self.i_out: float = 0.0
        self.d_out: float = 0.0
        self.prev_time = time.time()
        self.prev_input: float = 0.0
        self.prev_output: float = 0.0


    def __call__(self, input):
        current_time = time.time()
        dt = current_time - self.prev_time
        self.prev_time = current_time
        d_input = input - self.prev_input

        if dt < self.dt_min:
            return self.prev_output

        error = self.setpoint - input

        # proportional control signal
        self.p_out = self.kp * error

        # integral control signal
        self.i_out += self.ki * error * dt
        self.i_out = clamp(self.i_out, self.min_output, self.max_output)

        # derivative control signal
        self.d_out = self.kd * d_input / dt

        output = self.p_out + self. i_out + self.d_out
        output = clamp(output, self.min_output, self.max_output)

        self.prev_input = input
        self.prev_output = output

        return output

    def tune(self, kp, ki, kd):
        self.kp = kp
        self.ki = ki
        self.kd = kd

    def get_kp(self):
        return self.kp

    def get_ki(self):
        return self.ki

    def get_kd(self):
        return self.kd
```

Autopilot

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32MultiArray
from pymavlink import mavutil
import socket

# When simulating the flight controller firmware through SITL,
# MAVLink connection is transmitted over UDP instead of UART / USB

# Quickly creates a socket to get programmatically get ahold of current IP—address:
def get_ip_address():
    = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("192.168.1.1", 80))
    return s.getsockname()[0]

class Autopilot(Node):
    def __init__(self):
        super().__init__('autopilot')
        self.get_logger().info('Node started')
        # Subscribes to yaw, thrust and pitch values being computed from the PIDs in "Controller node":
        self.subscription = self.create_subscription(Int32MultiArray, 'yaw_thrust_pitch', self.listener_callback, 10)
        self.serial0_udp = 'udpin:' + get_ip_address() + ':14550'
        # Tries to establish MAVLink connection on udpin:[IP_ADDRESS]:14550 and waits for heartbeat:
        self.the_connection = mavutil.mavlink_connection(self.serial0_udp)
        self.the_connection.wait_heartbeat()
        self.get_logger().info('Heartbeat from: %s' % self.the_connection.target_system)

    def listener_callback(self, msg_in):
        self.the_connection.mav.manual_control_send(    self.the_connection.target_system,  # Established after heartbeat
                                                        msg_in.data[2],                     # Established after heartbeat
                                                        0,                                  # Roll value (static)
                                                        msg_in.data[1],                     # Thrust value
                                                        msg_in.data[0],                     # Yaw value
                                                        0)                                  # Bitfield corresponding to extra
                                                                                            # buttons , not needed and can be
                                                                                            # set to 0 for this purpose

def main(args=None):
    rclpy.init(args=args)
    autopilot = Autopilot()
    rclpy.spin(autopilot)
    autopilot.destroy_node()
    rclpy.shutdown()


if (__name__ == "__main__"):
    main()
```

G.1.4 Image Processing, configuration 4

Video capture node

This script creates a ROS2 node capable of capturing and streaming video data from a Raspberry Pi Camera using ROS2, OpenCV, cv_bridge, and picamera2.

First, we import the required packages that allow us to establish ROS2 nodes, manage ROS2 image data, work with the Raspberry Pi camera, and convert between ROS2 and OpenCV image formats:

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
import cv2
from cv_bridge import CvBridge
from picamera2 import Picamera2
from libcamera import controls
```

Then we define the `CameraCapture` class, inheriting from Node. In this class's constructor, we initialize the parent class with the name `camera_capture`, generate a publisher to transmit Image messages over the `image_data` topic, establish a timer with a callback function for publishing image data, configure and initiate the Raspberry Pi camera, and instantiate a bridge for converting between OpenCV and ROS image formats:

```python
class CameraCapture(Node):
    def init(self):
        super().init('camera_capture')
        self.publisher_ = self.create_publisher(Image, 'image_data', 10)
        self.timer = self.create_timer(1/30, self.publish_image_data)
        self.opencv_video = Picamera2()
        self.opencv_video.configure(self.opencv_video.
    create_preview_configuration(main={"format": 'RGB888', "size": (640,
    480)}))
        self.opencv_video.set_controls({"AwbEnable": True})
        self.opencv_video.start()
        self.bridge = CvBridge()
```

In the `publish_image_data` function, we capture a frame from the Raspberry Pi camera, convert it to a ROS2 message, and publish the message:

```python
def publish_image_data(self):
    frame = self.opencv_video.capture_array()
    msg = self.bridge.cv2_to_imgmsg(frame, encoding='bgr8')
    self.publisher_.publish(msg)
```

The main function initializes the ROS2 client library (rclpy), creates an instance of our `CameraCapture` node, waits for incoming messages, properly destroys the node when done, and finally shuts down the ROS2 client library:

```python
def main(args=None):
    rclpy.init(args=args)
    node = CameraCapture()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if name == 'main':
    main()
```

Blob Detection

This task involves developing a ROS2 node capable of detecting specific color objects in a video stream, estimating their relative positions and distances, and publishing this information using ROS2. The utilized packages include ROS2 (rclpy), OpenCV, cv_bridge, cvzone, and numpy.

First, we import the necessary packages for establishing ROS2 nodes, handling ROS2 messages, image processing, blob detection and numerical operations:

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import Int32MultiArray
import cv2
from cv_bridge import CvBridge
import cvzone
from cvzone.FPS import FPS
from cvzone.ColorModule import ColorFinder
import numpy as np
import time
import math
```

We define the `ObjectDetection` class, which inherits from `Node`. In the constructor of this class, we create a subscriber to receive Image messages from the `image_data` topic, initialize the FPS reader, bridge for converting between OpenCV and ROS2 image formats, and a publisher to broadcast `Int32MultiArray` messages (the object position and distance data) on the `distance_and_pos` topic:

```python
class ObjectDetection(Node):
    def init(self):
        super().init('object_detection')
        self.subscription = self.create_subscription(Image, 'image_data',
   self.process_image, 10)
        self.bridge = CvBridge()
        self.fpsreader = FPS()
        self.distance_and_position_publisher = self.create_publisher(
   Int32MultiArray, 'distance_and_pos', 10)
```

The `is_circle` function is defined to assess the circularity of a contour based on its area and perimeter:

```python
def is_circle(self, cnt, threshold=0.75):
    area = cv2.contourArea(cnt)
    perimeter = cv2.arcLength(cnt, True)
    if perimeter == 0:
        return False
    circularity = 4 * np.pi * area / (perimeter * perimeter)
    return circularity >= threshold
```

In the `process_image` function, we convert the incoming ROS message to an OpenCV image, detect specific colors (blue, green, red) in the image, filter the resulting contours for circularity, compute the distance and position of each detected object based on the size and location of the contours, log this information, and publish it as a Int32MultiArray message. If you want to read more about how the detection algorithm works read ??.

```python
def process_image(self, msg):

    opencv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    h, w, _ = opencv_image.shape

    myColorFinder: ColorFinder = ColorFinder(False)
    hsvValsBlue = {'hmin': 104, 'smin': 128, 'vmin': 0, 'hmax': 120, 'smax': 255, 'vmax': 152}
    hsvValsGreen = {'hmin': 26, 'smin': 54, 'vmin': 48, 'hmax': 90, 'smax': 137, 'vmax': 255}
    hsvValsRed = {'hmin': 0, 'smin': 120, 'vmin': 120, 'hmax': 20, 'smax': 255, 'vmax': 255}

    fps, img = self.fpsreader.update(opencv_image)

    _, maskBlue = myColorFinder.update(img, hsvValsBlue)
    _, maskGreen = myColorFinder.update(img, hsvValsGreen)
    _, maskRed = myColorFinder.update(img, hsvValsRed)

    _, contoursBlue = cvzone.findContours(img, maskBlue)
    _, contoursGreen = cvzone.findContours(img, maskGreen)
    _, contoursRed = cvzone.findContours(img, maskRed)

    circular_contours_blue = [cnt for cnt in contoursBlue if self.is_circle(cnt['cnt'])]
    circular_contours_green = [cnt for cnt in contoursGreen if self.is_circle(cnt['cnt'])]
    circular_contours_red = [cnt for cnt in contoursRed if self.is_circle(cnt['cnt'])]

    for color, circular_contours_list in zip(['blue', 'green', 'red'],
                                             [circular_contours_blue, circular_contours_green, circular_contours_red]):
        if circular_contours_list:

            cnt = circular_contours_list[0]
            x, y = cnt['center']

            f = 889
            W = 6.5
            w = cnt['bbox'][3]
            d = (W * f) / w

            self.publish_dist_and_pos(x , y, d)
```

The `publish_dist_and_pos` function prepares and publishes a `Int32MultiArray` message containing the x and y position and distance of a detected object:

```
def publish_dist_and_pos(self, x, y, distance):
    msg = Int32MultiArray()
    msg.data = [int(x), int(y), int(distance)]
    self.distance_and_position_publisher.publish(msg)
```

The main function initializes the ROS2 client library, creates an instance of our `ObjectDetection` node, waits for incoming messages, properly destroys the node when done, and finally shuts down the ROS2 client library:

```
def main(args=None):
    rclpy.init(args=args)
    node = ObjectDetection()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

G.1.5 Drone Position Estimation

In our application, we focus on estimating the drone's position relative to a stationary object. For this, we implement a `DronePositionEstimator` algorithm. This algorithm makes use of yaw angle and the distance between the drone and the object to estimate the position of the drone.

To start with, we import the necessary packages for ROS2 node creation, message handling, drone position calculation, and plotting:

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu
from std_msgs.msg import Int32MultiArray
from pymavlink import mavutil
from pymavlink_msgs.msg import DronePose
import math
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np
```

Next, we define a class `DronePositionEstimator` that inherits from the ROS2 Node class. The `def __init__` function is for when the drone is first initialized and we use this function to define variables and the creation of subscribers and publishers. This class subscribes to the topics `distance_and_pos` and `drone_pose` that provide the distance to the object and the pose data of the drone, and initializing the variables that we use in the plot.

```python
class FollowAlgorithm(Node):
    def __init__(self):
        super().__init__('follow_algorithm')

        self.create_subscription(Int32MultiArray, 'distance_and_pos', self.
   position_and_distance_callback, 10)

        self.create_subscription(DronePose, 'drone_pose', self.
   qualisys_callback, 10)
        self.distance = 0.0

        #self.fig = plt.axes(projection='3d')
        arrsize = 1000
        self.drone_x_arr = np.zeros(arrsize)
        self.drone_y_arr = np.zeros(arrsize)
        self.drone_z_arr = np.zeros(arrsize)
        self.qualisys_x_arr = np.zeros(arrsize)
        self.qualisys_y_arr = np.zeros(arrsize)
        self.qualisys_z_arr = np.zeros(arrsize)
        self.counter = 0
```

In the function `qualisys_callback`, which is triggered when a new message is published on the `drone_pose` topic, the estimated drone position is calculated based on the yaw angle and the distance to the object and then plotting:

```python
def qualisys_callback(self, msg: DronePose):

    object_x = 0.0 # meters
    object_y = 0.0 #meters
    object_z = 0.0 #meters
    camera_angle = 45  # degrees
    yaw = msg.yaw.data # radians -pi to pi

    true_x = msg.pos.x # x position in meters from qualisys
    true_y = msg.pos.y # y position in meters from qualisys
    true_z = msg.pos.z # z position in meters from qualisys

    distance_to_object = self.distance# meters

    # Calculate the drone's position
    drone_x, drone_y, drone_z = self.find_drone_position(yaw,
   distance_to_object, object_x, object_y, object_z, camera_angle)

    # Plot data
    self.plotter((drone_x, drone_y, drone_z), (true_x, true_y, true_z))
```

The function `find_drone_position` is responsible for the calculation of the drone's position:

```python
def find_drone_position(self, yaw, distance_to_object, object_x, object_y,
   object_z, camera_angle):

    drone_x = object_x - distance_to_object * math.sin(yaw)
    drone_y = object_y + distance_to_object * math.cos(yaw)
    drone_z = 0

    return drone_x, drone_y, drone_z
```

The function `position_and_distance_callback` updates the distance when a new message is published on the `distance_and_pos` topic. This function also converts cm into meters and adds the distance from the camera to the center of the drone:

```python
def position_and_distance_callback(self, msg: Int32MultiArray):
    self.distance = (msg.data[2] + 12) / 100
```

The function plotter collects and plots the estimated position of the drone and the actual position of the drone, saving the plots when the array is full:

```python
def plotter(self, drone_pos, qualisys_pos):
    if (self.counter < len(self.drone_x_arr)):
        # Scatter plots
        x_drone = drone_pos[0]
        y_drone = drone_pos[1]
        z_drone = drone_pos[2]
        x_qualisys = qualisys_pos[0]
        y_qualisys = qualisys_pos[1]
        z_qualisys = qualisys_pos[2]
        ndx = self.counter
        self.counter += 1

        self.drone_x_arr[ndx] = x_drone
        self.drone_y_arr[ndx] = y_drone
        self.drone_z_arr[ndx] = z_drone
        self.qualisys_x_arr[ndx] = x_qualisys
        self.qualisys_y_arr[ndx] = y_qualisys
        self.qualisys_z_arr[ndx] = 0
    else:
        self.save_plot()

def save_plot(self):
    plt.figure()
    fig = plt.axes()
    fig.scatter(self.drone_x_arr, self.drone_y_arr, s=len(self.drone_x_arr)
   , c='Blue', marker='.')
    fig.scatter(self.qualisys_x_arr, self.qualisys_y_arr, s=len(self.
   qualisys_x_arr), c='Green', marker='.')
    plt.savefig("/home/vaffe/ros/log/plot.png")
```

Lastly, in the main function, we initialize a ROS2 node for the `DronePositionEstimator` and spin the node:

```python
def main(args=None):
    rclpy.init(args=args)
    node = DronePositionEstimator()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

It's important to note that the accuracy of the estimated position depends on the precision of the yaw angle and distance data.

G.1.6 Image Processing, configuration 4 (Code Only)

Camera Capture

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image # import the Image message type
import cv2
from cv_bridge import CvBridge
from picamera2 import Picamera2
from libcamera import controls

class CameraCapture(Node): # inherits from Node
    def __init__(self): # constructor
        super().__init__('camera_capture') # call the constructor of the parent class
        self.publisher_ = self.create_publisher(Image, 'image_data', 10) # create a publisher
        self.timer = self.create_timer(1/30, self.publish_image_data) # create a timer
        self.opencv_video = Picamera2() # open the video capture device
        self.opencv_video.configure(self.opencv_video.create_preview_configuration(main={"format": 'RGB888', "size": (640, 480)}))
        self.opencv_video.set_controls({"AwbEnable": True})
        self.opencv_video.start()
        self.bridge = CvBridge() # create a bridge between OpenCV and ROS

    def publish_image_data(self): # callback function
        frame = self.opencv_video.capture_array() # read a frame from the video capture device
        #if not success:
       #     self.get_logger().info('Failed to read frame from camera')
       #     return
        msg = self.bridge.cv2_to_imgmsg(frame, encoding='bgr8') # convert the image to a ROS message


        self.publisher_.publish(msg) # publish the message

def main(args=None): # args is a list of strings
    rclpy.init(args=args) # initialize the ROS client library

    node = CameraCapture() # create a node

    rclpy.spin(node) # wait for messages
    node.destroy_node() # destroy the node explicitly

    rclpy.shutdown() # shutdown the ROS client library

if __name__ == '__main__': # if this file is run as a script
    main() # run the main function
```

Blob Detection

```python

#!/usr/bin/etv python

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import Int32MultiArray
import cv2
from cv_bridge import CvBridge
import cvzone
from cvzone.FPS import FPS
from cvzone.ColorModule import ColorFinder
import numpy as np
import time
import math



class ObjectDetection(Node):
    def __init__(self):
        super().__init__('object_detection') # call the constructor of the parent class
        self.subscription = self.create_subscription(Image, 'image_data', self.process_image, 10) # create a subscriber
        self.bridge = CvBridge() # create a bridge between OpenCV and ROS
        self.fpsreader = FPS() # Initialize FPS reader
        self.distance_and_position_publisher = self.create_publisher(Int32MultiArray, 'distance_and_pos', 10)


    def is_circle(self, cnt, threshold=0.75):
            area = cv2.contourArea(cnt)
            perimeter = cv2.arcLength(cnt, True)
            if perimeter == 0:
                return False
            circularity = 4 * np.pi * area / (perimeter * perimeter)
            return circularity >= threshold



    def process_image(self, msg): # callback function main processing function

        opencv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        h, w, _ = opencv_image.shape

        myColorFinder: ColorFinder = ColorFinder(False)
        hsvValsBlue = {'hmin': 104, 'smin': 128, 'vmin': 0, 'hmax': 120, 'smax': 255, 'vmax': 152} #blue
        hsvValsGreen = {'hmin': 26, 'smin': 54, 'vmin': 48, 'hmax': 90, 'smax': 137, 'vmax': 255} #green
        hsvValsRed = {'hmin': 0, 'smin': 120, 'vmin': 120, 'hmax': 20, 'smax': 255, 'vmax': 255} #red

        fps, img = self.fpsreader.update(opencv_image)

        _, maskBlue = myColorFinder.update(img, hsvValsBlue)
        _, maskGreen = myColorFinder.update(img, hsvValsGreen)
        _, maskRed = myColorFinder.update(img, hsvValsRed)


        _, contoursBlue = cvzone.findContours(img, maskBlue)
        _, contoursGreen = cvzone.findContours(img, maskGreen)
        _, contoursRed = cvzone.findContours(img, maskRed)

        # Filter contours that are circles
        circular_contours_blue = [cnt for cnt in contoursBlue if self.is_circle(cnt['cnt'])]
        circular_contours_green = [cnt for cnt in contoursGreen if self.is_circle(cnt['cnt'])]
        circular_contours_red = [cnt for cnt in contoursRed if self.is_circle(cnt['cnt'])]

        # Process and display depth, x, and y position for each ball
        for color, circular_contours_list in zip(['blue', 'green', 'red'],
                                                 [circular_contours_blue, circular_contours_green, circular_contours_red]):
            if circular_contours_list:

                cnt = circular_contours_list[0]
                x, y = cnt['center']

                f = 889
                W= 6.5
                w = cnt['bbox'][3]
                d = (W * f) / w
                self.get_logger().info(f"{color}: {d}")
                self.get_logger().info(f" fps: {fps}")
                self.get_logger().info(f" x: {x} y: {y}")

                self.publish_dist_and_pos(x , y, d)

    def publish_dist_and_pos(self, x, y, distance):
        msg = Int32MultiArray()
        msg.data = [int(x), int(y), int(distance)]
        self.distance_and_position_publisher.publish(msg)

def main(args=None):
    rclpy.init(args=args)

    node = ObjectDetection()

    rclpy.spin(node)
    node.destroy_node()

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Object Estimation

```

#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu
from std_msgs.msg import Int32MultiArray
from pymavlink import mavutil
from pymavlink_msgs.msg import DronePose
import math
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np


class FollowAlgorithm(Node):
    def __init__(self):
        super().__init__('follow_algorithm')

        self.create_subscription(Int32MultiArray, 'distance_and_pos', self.position_and_distance_callback, 10)

        self.create_subscription(DronePose, 'drone_pose', self.qualisys_callback, 10)
        self.distance = 0.0

        arrsize = 1000
        self.drone_x_arr = np.zeros(arrsize)
        self.drone_y_arr = np.zeros(arrsize)
        self.drone_z_arr = np.zeros(arrsize)
        self.qualisys_x_arr = np.zeros(arrsize)
        self.qualisys_y_arr = np.zeros(arrsize)
        self.qualisys_z_arr = np.zeros(arrsize)
        self.counter = 0


    def qualisys_callback(self, msg: DronePose):

        object_x = 0.0  # meters
        object_y = 0.0  # meters
        object_z = 0.0
        camera_angle = 45  # degrees
        yaw = msg.yaw.data # radians -pi to pi


        true_x = msg.pos.x # x position in meters from qualisys
        true_y = msg.pos.y # y position in meters from qualisys
        true_z = msg.pos.z
        distance_to_object = self.distance# meters

        drone_x, drone_y, drone_z = self.find_drone_position(yaw, distance_to_object, object_x, object_y, 0, camera_angle)


        self.plotter((drone_x, drone_y, drone_z), (true_x, true_y, true_z))


    def find_drone_position(self, yaw, distance_to_object, object_x, object_y, object_z, camera_angle):
    def find_drone_position(self, yaw, distance_to_object, object_x, object_y, object_z, camera_angle):

        # Calculate drone position
        drone_x = object_x - distance_to_object * math.sin(yaw)
        drone_y = object_y + distance_to_object * math.cos(yaw)
        drone_z = 0
        return drone_x, drone_y, drone_z

    def position_and_distance_callback(self, msg: Int32MultiArray):
```

```python

        self.distance = (msg.data[2] + 12) / 100
        self.get_logger().info(f"Distance to object {self.distance} ")


    def plotter(self, drone_pos, qualisys_pos):


        if (self.counter < len(self.drone_x_arr)):
            # Scatter plots
            x_drone = drone_pos[0]
            y_drone = drone_pos[1]
            z_drone = drone_pos[2]
            x_qualisys = qualisys_pos[0]
            y_qualisys = qualisys_pos[1]
            z_qualisys = qualisys_pos[2]
            ndx = self.counter
            self.counter += 1

            self.drone_x_arr[ndx] = x_drone
            self.drone_y_arr[ndx] = y_drone
            self.drone_z_arr[ndx] = z_drone
            self.qualisys_x_arr[ndx] = x_qualisys
            self.qualisys_y_arr[ndx] = y_qualisys
            self.qualisys_z_arr[ndx] = 0
        else:
            self.save_plot()

    def save_plot(self):
        plt.figure()
        fig = plt.axes()
        fig.scatter(self.drone_x_arr, self.drone_y_arr, s=len(self.drone_x_arr), c='Blue', marker='.')
        fig.scatter(self.qualisys_x_arr, self.qualisys_y_arr, s=len(self.qualisys_x_arr), c='Green', marker='.')
        plt.savefig("/home/vaffe/ros/log/plot.png")

def main(args=None):
    rclpy.init(args=args)
    follow_algorithm = FollowAlgorithm()
    rclpy.spin(follow_algorithm)
    follow_algorithm.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Appendix H

Tensorflow Lite Model Maker script

H.1 Tensorflow Lite Model Maker script

```python

from tflite_model_maker import model_spec
from tflite_model_maker import object_detector
import tensorflow as tf

# Only log errors
tf.get_logger().setLevel('ERROR')

# Load train, validation and test data. Arguments 1 and 2 are relative paths
# to image and annotation file directory, 3rd argument is a label map.
training_data = object_detector.DataLoader.from_pascal_voc(
    'training_images2', 'training_images2', ['Greenball']
)

validation_data = object_detector.DataLoader.from_pascal_voc(
    'validation_images2', 'validation_images2', ['Greenball']
)

test_data = object_detector.DataLoader.from_pascal_voc(
    'test_images', 'test_images', ['Greenball']
)

# Select pretrained model and directory for storing temp data
spec = model_spec.get('efficientdet_lite1', model_dir='D:/tfTemp/edl1_1k')
# Number of passes through full dataset
epochs = 1000
# Number of batches the full dataset is divided into
# Higher batch size = faster training
# Max batch size depends on available video memory
batch_size = 16

# Start training/create model
model = object_detector.create(training_data, model_spec=spec,
batch_size=batch_size, train_whole_model=True,
epochs=epochs, validation_data=validation_data)
# Evaluate Tensorflow model
eval_result = model.evaluate(test_data)
# Convert to Tensorflow Lite model
model.export(export_dir='export', tflite_filename='edl1_1k.tflite')
# Evaluate Tensorflow Lite model
tflite_eval_result = model.evaluate_tflite('export/edl1_1k.tflite', test_data)

print(eval_result)
print("--------------------")

# This returns the final precision and recall scores achieved on the test data set
print(tflite_eval_result)
```

Appendix I

Team

I.1 Team Members

- Martin Børte Liestøl – Software and Object Detection: As our software lead, Martin made sure that we were on the right track when it come to picking and developing our code. He was also responsible for our accelerated configurations.
- Even Jørgensen - Group Leader and Nvidia Jetson: As a group leader he made sure that each group member was seen and heard, and was the one communicating with the school. He was also responsible for the Nvidia Jetson Nano configuration of our project.
- Sindre Nes – Report, Ros2 and Qualisys: Sindre was responsible for the unaccelerated configuration, and making Qualisys operational.
- Ådne Kvåle – Hardware and Drone: Ådne focused on the hardware and flightcontroller. From building the drone to making it fly.
- Jon Jahren – Process and DevOps: Jon was in charge our group's methodology and DevOps practices. He established an environment conducive to efficient collaboration by leveraging tools such as Taiga for agile project management and GitHub for version control. Additionally, he took charge of managing our Docker images, ensuring smooth deployment and containerization processes.
- Abdul Majeed Alizai – Documentation, Blob Detection, and website: Abdul contributed to the project by creating the blob algorithm for one of the configurations, being responsible for the risk analysis, and creating the website for the team.

Group 6-2022 – Aerial Edge

Our thesis is to research the possibilities edge computing gives in an embedded system.

Even Jørgensen
Even1993@hotmail.com
Computer engineer

Main responsibility
Group leader

Martin Børte Liestøl
Martin.liest@gmail.com
Computer engineer

Main responsibility
Programming

Sindre Nes
Sin_pin@hotmail.com
Computer engineer

Main responsibility
Report

Ådne kvåle
237113@usn.no
Tlf: 47242193
Computer engineer

Main responsibility
Hardware

Abdul Majeed Alizai
Majeed.alizai4@gmail.com
Computer engineer

Main responsibility
Documentation

Jon Jahren
jon.jahren@gmail.com
Computer engineer

Main responsibility
Project model/Agile

Internal supervisor: **Henning Gundersen**
E-mail: **Henning.gundersen@usn.no**

External sensor: **Tord Fauskanger**
E-mail: **tord.fauskanger@gmail.com**

External supervisor: **Jan Dyre Bjerknes**
E-mail: **Jan.dyre.bjerknes@kongsberg.com**