UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

# Mearka

Architecting and evaluation of a Sports Video Tagging Software Toolkit

Alexander Torkelsen

INF-3981: Master's thesis in Computer Science, June 2023

“Sometimes life is going to hit you in the head with a brick. Don’t lose faith..”
–Steve Jobs

# Abstract

In the past decade, substantial advancements have been achieved in effectively utilizing video surveillance and associated analysis technologies within the realm of sports. This progress has been particularly noteworthy in elite sports, where the exploitation of athletes' digital footprints for sports analytics has emerged as a catalytic factor, ushering in a paradigm shift in comprehending and formulating strategic approaches to the game.

The architecture of sports video analytics systems can be broadly categorized into (1) tagging and (2) analysis. Tagging involves annotating metadata to specific video sequences and events, and this tagged metadata is subsequently utilized in the causal analysis process.

Multiple enterprise solutions are available today for recording videos, and positions and producing tagged data for the top teams. The issue is that they are often expensive, time-delayed metadata, and the sports organizations do not control where the data is stored or how the analytics company uses it. The alternative to enterprise solutions is manually generating the soccer metadata, which is time-consuming and possibly impossible if, for example, one wants to tag every player's position throughout a game.

This thesis presents Mearka, a distributed soccer tagging system based on cheap common-off-the-shelf components. It allows for tagging events LIVE during a soccer game through the Mearka-app, as well as generating player position metadata with time offsets into a user-uploaded video through the Mearka web-interface, automatically detected using machine learning. After detection, it is possible to download the soccer metadata as a JSON file through the web-interface.

The experiment results demonstrate that Mearka can complete the detection of players' positions from a 90 minutes soccer game within 12 hours after detection is started, with a video resolution of 1920x1080 at 25FPS. Expanding Mearka to only detect on every 10th frame could potentially make Mearka a viable real-time tagging option, as it is able to detect on $\approx$3 frames per second, and a turnaround of 12 hours detects on every single video frame.

# Acknowledgements

I want to thank my supervisor, Dag Johansen, and co-supervisor, Pål Halvorsen, for their unwavering support, valuable guidance, and continuous availability throughout my thesis. Their passion for the subject matter is genuinely infectious and has been a constant source of inspiration.

I also want to thank Martin Rypdal, the trainer of Hamna IL, for his incredible inspiration, enthusiasm, and invaluable cooperation. His engaging discussions and insight have been instrumental in shaping the development of Mearka into what it is today.

Lastly, I want to thank Ingrid for her love and support during this final year. Your support has been incredibly invaluable.

Thank you.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CLI | Command Line Interface |
| CNN | Convolutional Neural Network |
| COTS | Common-Of-The-Shelf |
| CPU | Central Processing Unit |
| CSG | Cyber Security Group |
| DNN | Deep Neural Network |
| FOV | Field Of View |
| FPS | Frames Per Second |
| GB | Gigabytes |
| GOP | Group Of Pictures |
| GPS | Global Positioning System |
| GPU | Graphical Processing Unit |
| HTML | HyperText Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| MAE | Mean Absolute Error |

| | |
|---|---|
| **MB** | Megabytes |
| **ML** | Machine Learning |
| **MSE** | Mean Squared Error |
| **NLP** | Natural Language Processing |
| **PB** | Petabytes |
| **POC** | Proof of concept |
| **POE** | Power over Ethernet |
| **REST** | Representational state transfer |
| **SGD** | Stochastic Gradient Descent |
| **UHD** | Ultra High Definition |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **UUID** | Universally Unique Identifier |
| **XML** | Extensible Markup Language |

# /1 Introduction

In the past decade, significant progress has been made in our capacity to utilize video surveillance and associated analysis technologies within sports effectively. Particularly in elite sports, the ability to harness the digital footprints of athletes for sports analytics has already emerged as a transformative factor, revolutionizing how the game is understood and strategized.

Sports organizations strive to maximize their potential for success, which might entail investing time and resources in sports technologies. These technologies aid in analyzing areas for improvement for the team and may also provide insights into the possible weaknesses of their opponents.

The architecture of sports video analytics systems can be broadly categorized into (1) tagging and (2) analysis. Tagging involves annotating meta-data to specific video sequences and events, and this tagged meta-data is subsequently utilized in the causal analysis process. The word "Mearka" means "mark" in northern Sami and is chosen as a name due to the nature of the system to be developed, that will annotate "marks" on video in the form of soccer metadata. While this thesis primarily focuses on the tagging aspect, it is developed in connection with the analytical part, particularly aligning and collaborating with the ongoing research on the analysis software, Dárkon[1] [1].

This chapter introduces some background and motivation, starting with how

1. conducted by fellow student Sebastian Lyng Johansen

data became popular in soccer. Following is a section introducing “State of the art” systems available to sporting organizations today and why they might be flawed. A section outlining the problem definition for this thesis follows, defining what is different about the proposed system compared to alternatives. Further is a section defining the scope and limitation of Mearka, before explaining in short what research method is used during the development. Lastly, a section explaining the context in which this thesis is written is introduced before the thesis outline is proposed.

## 1.1 Background and Motivation

It is becoming increasingly important to sporting organizations to quantify the performance of a team or individual player through numbers. An example of this is shown in the book “Expected Goals” by Rory Smith [2]. The book explains how Chris Anderson, an academic, saw data as an opportunity to fundamentally change soccer by obtaining data and analyzing it to determine how a team could gain a competitive advantage. Anderson was inspired by “Moneyball”, written by Michael Lewis [3], which follows Billy Beane et al., a baseball theorist searching for knowledge and insights in numbers that could give the “underdog” team an edge over teams with big money.

An early company, Prozone [2], tried to convince multiple soccer teams to give them data to analyze, and in return, they would help the teams see their strengths and weaknesses based on numbers. After a difficult start, where no clubs were interested, Prozone became a big business, analyzing and reporting games for many top-tier clubs. Many companies have followed Prozone in the business of soccer analytics since. In May 2015, Stats Perform acquired Prozone [4] to add advanced technologies to its tracking and analytics products suite.

Commercial entities already provide detailed tagging data for soccer games in top leagues. However, these services come with drawbacks such as high costs, delays in response, external dependence on third-party providers, debatable quality of tags, and unavailability in lower leagues. Numerous young players develop their talents in these lower leagues, making the lack of accessible and affordable tagging data services a significant challenge.

Hudl [5] is one such commercial entity for video capturing that offers some tagging functionality. They are a widely-used system for capturing a panoramic view of the soccer field. This system enables teams to record videos and simultaneously tag events during gameplay. However, tagging each event requires manual input, which can be costly if extra personnel is needed. This is often not a feasible option for smaller organizations that lack the necessary resources. As an alternative, recording the video footage and then tagging events retrospectively could be a more viable solution.

The utilization of video technology to record and evaluate the performance of a team or an individual player is on the rise. If a sports organization desires a more comprehensive analysis of how to team is positioned on the field, a positioning system can be a supplement to video technology.

One option to monitor individual players' location on the field could be to use ZXY [6], or similar systems. ZXY equips each player with a belt that emits a signal to radio towers positioned throughout the stadium. These signals are triangulated to determine the exact location of each player before being recorded and stored.

Another increasingly popular system to track players on the field is to use a Global Positioning System (GPS) vest like the one Statsport provides [7]. Because the vest is equipped with GPS, there is no need to set up radio transmitters and receivers around the stadium like with ZXY [6]. This means the vest can be used anywhere, as long as there is GPS signal. At the time of writing this thesis (April 2023), the vest from Statsport retails for around €235 [7] on sale to individuals.

Today, sporting organizations have two main options regarding video systems to record and tag matches and training sessions:

1. Use a camcorder or similar to manually record and tag events afterward.
2. Buy an expensive system like Hudl [5], or any of their competitors [8, 9], that records videos and tags them manually for the team.

For positional data of where the players are on the field, sports organizations have the following options today:

1. Buy ≈25 GPS vests from someone like Statsport [7], as there are between 20-25 players in each workout. Totaling between €4700 - €5875 to equip a single team with GPS vests.
2. Invest in a system like Hudl [5] and get them to tag the positions from the video, resulting in higher prices.
3. Record the video and tag the position of every player on the field manually afterward.

The problems with the above-mentioned solutions include that they are usually expensive to buy and operate, require too much manual labor, delay on delivering data, or a combination. The application domain will be soccer related, although the final product could be sports agnostics with minor tweaks. It is intended for bigger and smaller teams to extract data from cheap sources like a small video camera.

## 1.2 Mearka Problem Definition

As sporting organizations are increasingly interested in extracting the maximum potential from their teams, analyzing data is an essential tool to do so. There is software available for automated tagging, which is currently in operational use. However, this software is costly, restricted to specific events, lacks sufficient precision, and may not fully satisfy the requirements of coaches and analysts. This thesis focuses on how to use video to create tags and metadata as efficiently, user-friendly, and cheaply as possible, compared to the alternatives that exists. The thesis statement is defined as:

> *It is possible to develop a soccer tagging system based on cheap, common-of-the-shelf components. This will contrast to the state-of-the-art systems using expensive and specialized hardware and software that depend on external storing and analytics of data, where you have no control over where the data is stored, how it is used, or the quality of the tagged data, in addition to a significant time-delay.*

The statement above is investigated through the following steps:

1. Outline specifications and requirements based on the abovementioned problem definition, related work, and domain.

2. Develop a proof of concept (POC) system and demonstrator based on the outlined specifications and requirements.

3. Evaluate the system through experiments and conclude to what degree the system satisfies the requirements.

## 1.3 Methods

To identify the common core in Computer Science subject matter, the ACM task force presents the following in their final report. This report presents an approach to divide the discipline of Computer Science into three paradigms: *Theory*, *Abstraction*, and *Design* [10].

***Theory:*** consists of four steps to aid in the development of a valid, coherent theory, and is rooted in mathematics.

1. **Definition:** Characterize the objects of study.

2. **Theorem** Hypothesize possible relationships among the objects.

3. **Proof** Determine if the hypothesized relationship between objects are true.

4. **Results** Interpret the result.

A theoretician is expected to re-iterate these steps, for example, if errors or inconsistencies are discovered.

***Abstraction:*** are rooted in the experimental scientific method and consists of four stages for investigating a phenomenon.

1. **Hypothesis:** Form a hyphotesis.

2. **Model:** Construct a model and make a prediction for the result.

3. **Design:** Design and run expriments to collect data.

4. **Analyze:** From the collected data, analyze the result.

A scientis is expected to re-iterate over these steps, for example when a predction from a model disagrees with experimental evidence.

***Design:*** is rooted in engineering and consists of four steps when a system or device, that solves a given problem, is constructed.

1. **Requirements:** State the requirements that the system/device needs to fulfill.

2. **Specification:** Specify how the system will fulfill the requirements.

3. **Design and implementation:** Design and implement the system.

4. **Test:** Test the system and make sure it reaches the requirements.

It is expected from an engineer to re-iterate these steps, for example, when tests reveal that the system does not satisfy the requirements stated.

This thesis roots in the *design* paradigm, as the requirement and specification are derived from the problem definition and the application domain. A system prototype will be designed and implemented based on the defined requirement and specifications. Lastly, the system will be evaluated through experiments to see if the POC meets said requirements.

## 1.4 Scope and Limitations

In order to limit the scope of the problem definition, it is necessary to make some assumptions with regard to the problem domain. The assumptions made are documented here.

- To simplify the offset calculations of tags and positions, the video provided by the user needs to start at kick-off. This enables the potential merging of metadata files to be more easily sorted, as any "offset" refers to an offset from the same point of time in the game. Another benefit is easier synchronization of positional data to tags, as the offset noted is the same.

- Mearka should ideally be so easy to use that anyone, technically savvy or not, could use it. However, implementing the POC of Mearka assumes the user knows some basic video editing, specifically to make sure the videos start at kick-off. [2]

- With regards to Machine Learning (ML), Mearka will rely on a pre-trained model, YOLOv4 [14], to detect where people are on the field. Instead of training a specific model for this application, a pre-trained model is chosen based on time constraints.

- Asume that Mearka will run in a trusted environment. The emphasis of the POC will be on the system functionality rather than the security of the system.

- Mearka will not provide the video recording device, as it aims to be video-source agnostic and able to utilize any video that the user provides.

2. Davinci resolve is a free, but very capable, video editing software for macOS, Windows, and Linux (download *Davinci-Resolve-18* [11]). Some sources to get started using Davinci Resolve can be found *here (In-depth-course [12])* and *here (shorter overview course [13])*.

- Mearka assumes it will be a small contribution to the ongoing research within the field of soccer analytics. Developing a system that aims to be easily expanded with more functionality in the future and used in combination with other analytical systems, like Dárkon.

## 1.5 Context

This thesis is written in the context of the Corpore Sano Center [3], which is affiliated with the Cyber Security Research Group (CSG) at UiT, The Arctic University of Norway.

Their research has primarily revolved around designing and developing distributed systems that are scalable, efficient, fault-tolerant, compliant, and secure. The methodology primarily adopted by CSG is an approach that develops experimental systems, experimentally evaluating a prototype middleware system to address different research problems.

Back in 1990, the group researched the co-operation issues in StormCast, a distributed artificial intelligence application for severe storm forecasting [15]. The design uses the employment of multiple "expert modules" that conducted predictive forecasting in their respective geographical area. Once a forecast has been made, it is broadcasted to other "expert modules" to be used in their future predictions, potentially. Deadlocks are prevented by each "expert module" only using an incoming prediction from a different area (and thereby another "expert module"), if it is possible. They found that a modular design with a loosely-coupled distributed approach, de facto industrial standards have been sufficient for this application.

Another area of research done by CSG is "Low Overhead Container Format for Adaptive Streaming", investigated in 2010 [16]. This paper proposes an adaptive video player that codes video segments as closed groups of pictures (GOP) when streaming. It codes the video as H.264/AVC [17] and the audio as mp3, both standard Audio/Video domain components. The audio and video are packaged in a custom-made low overhead container optimized for streaming and easily translatable to different containers if needed. The player can pick a quality level that uses most of the available network bandwidth and utilizes the CPU while still having smooth and uninterrupted playback and close to instant seek and startup times.

In 2018, CSG, together with the School of Sport Sciences, UIT The Arctic

3. `https://corporesano.no`

University of Norway, ForzaSys AS and Simula Research Laboratory in Oslo, Norway, did a case study on quantifying soccer using positional data [18]. There is an increasing availability of athlete quantification data, and more data is being collected automatically, especially after FIFA approved the use of wearable electronic performance and tracking systems. This research presents their experience using radio-based wearable positioning data systems in elite soccer clubs (ZXY [6]). The difference between a GPS system and a radio-based system is investigated in the paper. They demonstrate that this data can be used to detect and find anomalies and trends and give important insights for individual players and soccer team performance development.

This research is just a fraction of what the CSG at UiT, The Arctic University of Norway, has conducted over the years. Mearka will be designed and developed in the context of the research already conducted by CSG about distributed systems, working with video as a source, and potensially detecting positions to quantize soccer performance which can be utilized in analytics.

## 1.6 Outline

**Chapter 2** covers relevant background information and related work within the soccer video and tagging systems domain.

**Chapter 3** outlines the requirement and spesifications that the POC aims to reach.

**Chapter 4** describes the design of the proposed system.

**Chapter 5** covers details on more implementation-specific subjects.

**Chapter 6** describes how the system is evaluated, which experiments are conducted, and why, as well as evaluates the results.

**Chapter 7** discusses the choices made during the implementation of the proposed POC system, and possible alternatives.

**Chapter 8** concludes the thesis by summarizing the POC system, and proposing future work for continuous system development.

# /2 Background

This chapter outlines relevant fundamental concepts grounded in the thesis and related work. Section 2.1 explains a design principle used to communicate by multiple components in Mearka. Section 2.2 describes the soccer-metadata formant that Mearka creates. Section 2.3 explains a framework that is key for this thesis to work with video files. Section 2.4 gives a brief introduction to machine learning, as well as touches on more specific libraries and models utilized. Mearka utilizes ML to detect the position of people in the video. Section 2.5 summarizes some related work relevant to this thesis. The mentioned work is important, as Mearka takes inspiration and builds on the ideas presented in these.

The goal of Mearka is to develop a sports video tagging system that can detect players' positions only based on video recorded with any device. To enable Mearka to try and solve the problem, it needs a way of communicating between distributed components, work video, do ML object detection, as well as output the resulting soccer metadata in a way that is understandable by both humans and other systems.

## 2.1 Representational State Transfer (REST)

A REST API, or Representational State Transfer Application Programming Interface, is a standardized architectural style for designing networked applications

[19]. It provides a set of rules and constraints that enable communication and interaction between different software systems over the internet. REST APIs are commonly used in web development to facilitate data exchange between clients (web browsers) and servers.

At its core, a REST API operates on the principles of statelessness and resource-based interactions. Statelessness means that each request from a client to a server should contain all the necessary information to understand and process the request without relying on previous requests or shared session state. This simplifies the server's job and allows for better scalability and reliability.

In a REST API, resources are the fundamental entities the API exposes and manipulates. These resources can be represented using Uniform Resource Identifiers (URIs), unique addresses used to identify them. Clients can interact with these resources using standard HTTP methods like GET, POST, PUT, and DELETE. For example, a client can retrieve data from a server by sending an HTTP GET request to a specific URI representing the desired resource, and the server responds with the requested data in a format like JSON or XML. Overall, a REST API provides a flexible and standardized approach to building web services that different clients can easily consume. It promotes loose coupling between the client and server, allowing them to evolve independently. REST APIs enable interoperability, scalability, and simplicity in distributed systems by adhering to the principles of statelessness, resource-based interactions, and standard HTTP methods.

## 2.2 JavaScript Object Notation (JSON)

JSON or JavaScript Object Notation is a lightweight data-interchange format that is easy for humans to read and write and for machines to parse and generate [20, 21]. JSON stores data in key-value pairs and arrays, making it a flexible way to represent complex data structures. It is a text format entirely language-independent, making it a popular choice for data exchange between different programming languages.

This thesis uses JSON to create and store metadata because it is easy for humans to read and for machines to parse and generate. JSON is widely used to store and send data between servers and clients, to store configurations and logs, etc. Because of this, there are also multiple libraries to parse and generate JSON. The most important ones for Mearka is a library for python [22], React Native [23], and Golang [24].

Below, figure 2.1 shows an example of how JSON data can look. This example shows a JSON “objects” with key-value pairs containing information about tagged events. Because of how JSON is structured, it is possible to parse this data and access each field based on the keys. An example, loop over each element in the list and print out the “eventType” key. As shown in figure 2.2, Python knows how to parse JSON and handles it like an object. Because it is handled as an object, it is just a matter of accessing the “eventType” attribute on that object to get out the attached value, in this case, “Pass”.

```
[
    {
        "timeId":3,
        "eventType":"Pass",
        "startPlayer":{
            "team":"our",
            "player":"Player 1"
        },
        "endPlayer":{
            "team":"opponent",
            "player":"Player 2"
        },
        "result":"medspiller",
        "startCoordinates":{
            "x":7.94e1,
            "y":3.45e1
        },
        "endCoordinates":{
            "x":6.47e1,
            "y":5.36e1
        },
        "startTime":2.2240000000000002e1,
        "endTime":2.232e1,
        "playingDirection":"RightToLeft",
        "ballWin":false,
        "breakThrough":false,
        "dangerousMistake":false
    },
    ⋮
]
```

**Figure 2.1:** Example JSON

```
for event in eventlist:
    print(f"Event type: {event.eventType}")
```

**Figure 2.2:** For-loop that prints the event type

## 2.3 FFMPEG

FFmpeg is a free and open-source multimedia framework widely used to process and manipulate audio and video files [25]. It consists of a set of command-line tools that allow users to perform a variety of operations on multimedia files, including cutting, encoding, decoding, and modifying. FFmpeg works by reading and writing multimedia data from various sources to one or multiple destinations, such as files, network streams, and devices. FFmpeg provides a powerful and flexible set of tools for working with multimedia files, making it a popular choice for many applications, including video editing, streaming, and compression.

This thesis uses FFmpeg in several ways, most predominantly to concatenate video files. Some cameras split longer video recordings into smaller video files, making it more difficult to calculate a time offset when doing object detection on the video. Since the user might not know how to merge multiple videos into one, the system enables the upload of multiple files to do merging automatically. Then the system figures out what is a video and what is not and concatenates the files into one longer video.

## 2.4 Machine Learning

Machine learning is a subset of artificial intelligence that involves the development of algorithms and statistical models to enable computer systems to learn from data and make predictions, decisions, or suggestions without being explicitly programmed [26].

There are three main methods of machine learning, supervised learning (section 2.4.1), unsupervised learning (section 2.4.2), and reinforcement learning (section 2.4.3). Some typical machine learning applications include image recognition, natural language processing, fraud detection, recommendation systems, and autonomous vehicles.

This thesis uses machine learning to detect where players are positioned in the image automatically. CVlib is used with the pre-trained machine learning model YOLOv4 (section 2.4.6) for object detection. Below is a brief overview of the three main methods for doing machine learning. However, since YOLOv4 is trained on the labeled Microsoft COCO dataset [27], using a supervised learning method, this is where the main focus of this section will be.

### 2.4.1 Supervised Learning

In supervised machine learning, "training a model" equals training the algorithm from data. The idea is that systems can automatically learn from data, identify patterns, and improve their performance over time. Supervised learning algorithms are designed to learn from large amounts of historical data and use this knowledge to make predictions or decisions about new data Supervised [26, ch. 2].

Mearka uses CVlib with the pre-trained model YOLOv4 [14] to detect the position of people in the video. This model was trained using a supervised learning method on the COCO image dataset created by Microsoft [27].

A model usually consists of multiple layers, and each layer is there to help the model to give a correct output. Each layer can consist of one or more "neurons", where each neuron works on its input values before passing the output to the next layer. Each input signal is given a weight to determine the importance of a given input signal from the previous layer. Figure 2.3 shows a simple example model with multiple layers containing neurons and weights between each layer. Models with multiple layers are also called Deep Neural Networks (DNN), as they are multiple layers "deep" [28]. The most popular type of machine learning model is the Convolutional Neural Network (CNN).

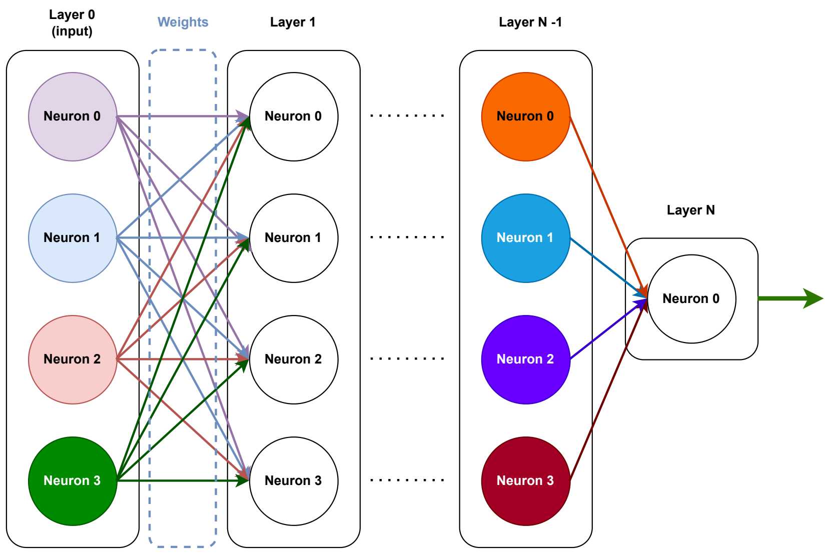


**Figure 2.3:** Example machine learning model with steps and neurons

For example, a model takes an image as input and can determine if the image contains a dog or a cat. Step one in the model could cut up the image into smaller chunks. Another might take the chunks and determine if they contain edges in different orientations. The next step tries to combine the inputs and see if the resulting sub-image contains parts of an ear, nose, paws, etc. Following a step might determine if the ear and nose are more likely to belong to a cat, dog, or something else. Based on this estimation, if the model is confident enough that a cat or dog is in the image, it outputs which one it thinks is present. "Confident enough" is determined by the developer. It is possible to set a probability threshold that states how certain a model has to be before stating that the image contains something.

## Training a model

Training a model means estimating a value for a given data point against the label, and adjusting the weights to better estimate the correct classification or prediction. Labeled data contains an observed truth for every data point in the set, often called "ground truth". An iterative method is usually used to optimize the weights as training progresses. One such iterative approach can be Stochastic Gradient Descent (SGD) as seen in equation 2.1 [29, eq. 4]. Here $w_{t+1}$ is the updated weight after the optimization, $w_t$ the current weight, $\gamma_t$ a stepsize function, and $\nabla_w Q\,(z_t, w_t)$ is the chosen optimization function that takes the labeled value $z_t$ and estimated value $w_t$ and returns the direction $w_{t+1}$ needs to change. The stepsize function $\gamma$ scales how much $w_{t+1}$ changes per iteration $t$ while keeping the direction it changes the same.

The supervised learning model splits the data into training and testing sets. The model is trained on the training data, adjusting its weights gradually with something like equation 2.1, based on how far off the labeled truth is. After the model is trained, it is tested with test data to see how well it performs on data it has never seen before. During testing, the model does not adjust any weights; it is tested as is.

$$SGD = w_{t+1} = w_t - \gamma_t \nabla_w Q\left(z_t, w_t\right) \tag{2.1}$$

### Evaluating model

How well a model is trained can be determined using different algorithms; one of these is the mean square error (MSE) algorithm. This looks at the mean of squared accumulated distances between estimated and labeled values [30] in the testing phase. As seen in equation 2.2, for $n$ data objects, $Y_i$ denotes the labeled value $Y$ on data point $i$, and $\widehat{Y_i}$ is the estimated value $\widehat{Y}$ of the same data point $i$. MSE is a good algorithm to use if it is important that the model does not contain any outlier predictions with significant errors. This is because the error, defined as the distance between the estimated and actual values for each data point, is squared and therefore made more significant in the overall MSE score.

$$MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(Y_i - \widehat{Y_i}\right)^2 \tag{2.2}$$

Unlike MSE in equation 2.2, mean absolute error (MAE) takes the absolute value of the errors without squaring them, as equation 2.3 shows [31]. The result is that it is less affected by some outlier predictions with bigger errors, making MAE a better choice if it does not matter that the model has some outlier predictions with higher errors. Both MSE and MAE will always have a positive score, as MSE Squares the errors while MAE takes the absolute value. No matter which of these two is used to evaluate a machine learning model, the closer to zero the score is, the more accurate the model is predicting

$$MAE = \frac{1}{n} \cdot \sum_{i=1}^{n} \left|Y_i - \widehat{Y_i}\right| \tag{2.3}$$

If the model needs to take the outliers into consideration more than what MAE does, but still to a lesser degree than what MSE does, the Huber loss function seen in equation 2.4 could be a good option [32, eq. 2]. Here it is possible to get “the best of both worlds”, as MSE is only used as long as the error is less than $\delta$, and MAE otherwise. When tweaking the model, $\delta$ can be adjusted to include outliers as close to or as far from the labeled truth as wanted while giving outliers beyond this less prominence in the score.

$$L_\delta(Y, \widehat{Y}) = \begin{cases} \frac{1}{\delta}\left(Y - \widehat{Y}\right)^2 & \text{, if } |Y - \widehat{Y}| \leq \delta \\ |Y - \widehat{Y}| & \text{, Otherwise} \end{cases} \tag{2.4}$$

### 2.4.2 Unsupervised Learning

Unsupervised learning is a fundamental concept in machine learning that involves the analysis of data without prior knowledge of the target variable or explicit supervision from a human expert. Unlike supervised learning, which relies on labeled data to guide learning, unsupervised learning seeks to discover underlying patterns and structures in unlabeled data [26, ch. 1.2.4]. The goal of unsupervised learning is to identify meaningful relationships and regularities in the data, which can be used to inform downstream tasks such as classification, clustering, and anomaly detection.

Several common approaches to unsupervised learning include clustering, dimensionality reduction, and generative modeling. These methods are not used in this thesis, therefore they will not be explained further. However, below is a brief overview of clustering using k-means as an example of how an unsupervised method might work.

Clustering involves grouping similar data points based on their distance or similarity. One algorithm to do just that is k-means clustering, which aims to partition an $N$-dimensional data population into $k$ sets [33]. Equation 2.5 shows a clustering algorithm that tries to find $k$ set of clusters $s$, such that the sum of squared distances between each data point $x$ and the mean $\mu_i$ of data points in $s_i$ is as short as possible.

$$\underset{S}{\arg\min} \sum_{i=1}^{k} \sum_{x \epsilon S_i} ||x - \mu_i||^2 \tag{2.5}$$

### 2.4.3 Reinforcement Learning

Reinforcement learning (RL) is a method of machine learning that involves an agent interacting with an environment to learn how to make optimal decisions. The agent receives feedback through rewards or penalties based on its actions, which guides its learning process. The goal is to create a policy that contains a sequence of actions that reaches the destination [26, ch. 1.2.5]. Since each intermediate step is on a path toward the destination, an action is "good" if it is part of a policy that leads there. RL is particularly useful when the optimal solution is unknown beforehand or difficult to calculate using traditional algorithms.

The RL process involves several key components, including a discrete set of *environment states*, *agent actions*, and a set of *reinforcement signals* [34, page. 3]. The *environment states* represent the possible states that the environment can be, while the *agent actions* is a set of all actions the agent can take. A *reinforcement signal* function assigns a numerical value to each action the agent takes, which the agent seeks to maximize, a low value can be considered a penalty. Through trial and error, the agent learns to optimize its policy and cumulative reward over time, resulting in a more effective decision-making process.

### 2.4.4 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library [35]. OpenCV provides various video and image processing algorithms, such as feature detection, object recognition, image segmentation, and several others.

Initially, Intel developed OpenCV in the late 1990s, and it has been supported by several organizations and individuals since. This thesis uses OpenCV version 4.6.x, distributed under the Apache License 2.0 [36], which allows OpenCV to be used, modified, and distributed without restriction.

This thesis uses OpenCV to load frames from a video into memory, which is further worked on by CVlib (section 2.4.5) for object detection.

### 2.4.5 CVlib

CVlib is a computer vision library built on top of TensorFlow [37] and Keras [38]. TensorFlow is a low-level numerical computation library that provides a flexible platform for building and deploying machine learning models. Keras is a high-level neural networks API written in Python that runs on top of TensorFlow. The main difference between TensorFlow and Keras is that TensorFlow is more low-level and requires more coding to build and train models, while Keras is more high-level and potentially provides a more straightforward, intuitive interface for building and training neural networks.

CVllib builds on these libraries but has a higher level of abstraction than Keras, meaning neither library is accessed directly by the developer. Because of this, there will not be a section going in-depth on how they work, just a mention that CVlib uses TensorFlow and Keras underneath.

CVlib provides easy-to-use functions for different computer-vision tasks such as object detection, face detection, and gender detection. It is mainly designed for Python and provides pre-trained models for easy use.

This thesis uses CVlib for its object detection functionality with the default pre-trained model YOLOv4 [14]. CVlib takes a frame from the video, loaded by OpenCV (section 2.4.4), and returns a list of pixel positions if YOLOv4 recognizes an object.

### 2.4.6 YOLOv4

YOLOv4 (You Only Look Once, version 4) is an advanced real-time object detection algorithm initially developed by Joseph Redmon and Ali Farhadi [39], and further improved by Alexey Bochkovskiy [14]. It is trained using the COCO dataset from Microsoft [27], and the improvements in version 4 over version 3 are improved accuracy, speed, and generalization capabilities.

YOLOv4 uses a deep neural network to identify objects in an image or video and predicts their location and class probabilities in a single forward pass of the network. This differs from other algorithms that need several passes over an image to detect multiple types/classes of objects. Overall, YOLOv4 is an efficient and powerful algorithm used for object detection that has been widely adopted in multiple computer vision applications.

CVlib is a high-level abstraction for object detection machine learning to make it easier to start doing object detection at all. There is a function that only needs an image as input and returns a list of positions for any objects detected. Referencing CVlib in section 2.4.5, YOLOv4 is the pre-trained model chosen by CVlib, and therefore used by this thesis to detect people and where they are in the image.

## 2.5 Related Work

Previously, the CSG team at UIT researched using video recording for soccer analysis. The selected research aimed to challenge the domination of expensive tools that only benefit financially strong sporting organizations. This thesis aims to build on the ideas and essence of the previous research and create an easy-to-use, semi-automatic, cheap, and efficient system for tagging. The goal is to simplify the setup and usage process, automating as much as possible while providing valuable analytical data.

### 2.5.1 Muithu

Muithu sought to provide an alternative to the excessive use of resources required by expensive tagging systems [40]. Traditional systems require multiple people to tag events during matches or training sessions, Moreover, using personnel can be costly. Additionally, since someone other than the head coach typically performed the tagging, some of the events that are tagged may not even be relevant. In contrast, Muithu uses affordable consumer cameras (GoPro HD Hero 2) and a phone app to move the tagging process into the hands of the head coach, enabling the coach to tag events in real-time as they unfold during a game or practice.

By involving the head coach in the tagging process, Muithu ensures that only relevant events are recorded. Moreover, with the ability to see a situation before tagging it, precision and recall rates approach 100%. However, the system must be non-intrusive and fast to use, allowing the coach to focus on the team. The intuitive user interface (UI) of the Muithu app enables the coach to tag a situation containing "who" and "what" within five seconds.

After recording, a timeline displays all the events noted, and the specific video segment from the camera(s) for each event can also be extracted. Each tag contains the offset into the recording where the event occurred, streamlining the workflow. While the final step is a manual process, it is still significantly faster than manually scrubbing through the entire recording in retrospect.

### 2.5.2 Bagadus

Bagadus builds on the research done by Muithu (section 2.5.1) and improves, and expands on it. It consists of three key sub-systems, Video, position, and analytics system [41]. The video sub-system is comprised of a multi-camera array that together covers the entire field. The system allows the user to view the output from a single camera or see the entire playing field in a panorama.

In addition, Bagadus has a sensor sub-system that tracks the location of each player on the field using ZXY tracking sensors [6]. The analytical sub-system enables users to zoom in and out, navigate the image, and record events in real-time. All events are automatically stored in a video and sensor data database for easy retrieval.

Additionally, Bagadus can query video clips based on specific sensor data. For instance, it is possible to query every clip where a particular player runs faster than a certain speed or multiple players are located in a specific field area.

Unlike Muithu's use of consumer-grade cameras that can be positioned anywhere, Bagadus relies on a static camera array, one of which is installed at Alfheim Stadium in Tromsø, and it has also been installed at Ullevaal Stadium for the national team. The reason for a static camera array is that the cameras must be synced and calibrated to provide a seamlessly stitched panorama, which is not a concern for Muithu's cameras.

Muihtu has an intuitive app that lets the user tag events as they happen during a game, as well as record the game with consumer cameras to be synced with the tags after the game. Bagadus expands on this with a static camera array delivering a panorama and adds positional data through ZXY tracking sensors [6], as well as an analytical sub-system that enables the user to pan around the panorama. Mearka aims to have a similarly intuitive app that the user can use to tag events as they happen during the game. Unlike the beforementioned systems, Mearka should be camera agnostic, utilizing any camera, from a phone to an action camera or up to professional-level broadcast cameras, as well as hopefully being able to extract the positions automatically from the video without the need for tracking sensors on the players.

## 2.6 Summary

This chapter introduced FFMPEG (section 2.3), which is a framework that is instrumental in how Mearka works with video files. Next was a section outlining some machine learning methods, like supervised, unsupervised, and reinforcement learning. Supervised learning (section 2.4.1, [42]), described how a machine learning "model" can be trained with labeled data (section 2.4.1), as well as how to evaluate the trained model (section 2.4.1). Furthermore, other machine learning libraries used in this thesis were introduced together with the particular model (YOLOv4 [14]), which will be used for object detection. Lastly, some relevant work like, Muithu (section 2.5.1) and Bagadus (section 2.5.2), were introduced.

The next chapter outlines and defines the functional and non-functional requirement specifications of Mearka.

# 3 Requirement Specification

Based on the goals specified in section 1.2, we have specified a set of requirements the system needs to meet to reach the goal. The requirements are divided into two separate components, functional and non-functional. Section 3.1 describes the functional requirements set for Mearka, while section 3.2 outlines the non-functional requirements.

## 3.1 Functional

Functional requirements are defined as clear statements outlining the services that the system should offer, specifying how the system should respond to specific inputs, dictate its behavior in particular situations, or highlight any actions the system should avoid [43, page 2-4].

1. **Input:** Mearka should support different inputs from the user. It must support live tagging of events during a game and handle video uploads of the game or training sessions afterward.

2. **One-click live tagging:** It should be possible to tag an event that happens live, with at most one click or press of a button.

3. **Automated:** Mearka should be able to extract positional data from the video automatically. The user should be able to upload any video or

multiple videos (if they are segments of the same video), to the system for analysis and get useful positional data back.

4. **One-click export:** Once the system has detected positions, it should be no more than one click to get the generated soccer metadata downloaded.

5. **Output:** The output from Mearka after detecting positions should be soccer metadata that enables the user to have information on the positions of players at any given time in the video. At a minimum, the output should be pixels positions in the frame as illustrated by figure 3.1, with x,y coordinates for player positions at time t. Ideally, the positions will be translated to real-world coordinates as illustrated by figure 3.2, with a simpler 2D-view.

6. **Data deletion:** The user owns their data, so Mearka needs to make it easy to delete what user data the system temporarily has stored. Ideally, the user only needs to click on one button to remove everything.

## 3.2 Non-functional

Non-functional requirements encompass limitations placed on the services or functionalities provided by a system. These constraints involve aspects such as timing, development process, and adherence to standards. Unlike functional requirements, which pertain to specific system features and services, non-functional requirements typically apply to the system as a whole[43, page 2-6].

1. **Ease-of-use:** Any UI of Mearka should be intuitive and easy to use. The user should not be unsure of what to expect when interacting with the system.

2. **Soft real-time:** Data resulting from using Mearka shall strive to be available in real-time in the long run, such that it is able to give feedback LIVE during a game. However, since Mearka shall be developed as a POC prototype in this thesis, a first iteration *soft real-time* goal will be to have data available within 12 hours of the system starting to process, such that a coach can analyze the data the following day, even if the performance of Mearka is not optimized.

3. **Common Of The Shelf (COTS) components:** Mearka needs to be as cheap as possible, both to set up and operate, which is why the system shall rely on only using COTS components.

4. **Data ownership:** The user owns the data, so Mearka shall only store the data for as long as needed and not store anything long-term.

5. **Privacy Compliance:** The user decides what data they upload and when they want it removed. The system shall make it easy for a user to delete their data on the system if they choose to.

6. **Security:** The system shall restrict the user-uploaded data and the generated data based on that input to only be available to the user that uploaded it. Mearka shall store data such that one user cannot access another user's data by accident or otherwise.

## 3.3 End user interactions

Someone on the team must set up a camera to record the match or training session. What kind of camera shall not be important as Mearka shall be able to work with the video no matter what camera it comes from. The coach shall be able to start recording events from the kickoff using a Mearka-app as the match or training session is about to start. Tagging events shall be able to be done while the session is still in progress, and the metadata shall be sent from the backend to the app once the session is over. After the session, the web interface could be used to upload the video. Uploading the video shall start the system to detect where players are positioned on the field. Merging the positional data with the events tagged through the app shall be done by uploading the file from the app through the web interface. Once Mearka is done detecting positions, soccer metadata shall be able to be downloaded through a single button click on the web interface. This file should then be able to be uploaded to another analyzing software, like Dárkon.

## 3.4 Summary

To summarize the proposed functional aspects of Mearka, it shall support different inputs from the user. This can be, at least, live tagging during a game and handling video uploads of the game or training session. Tagging an event shall ideally be as simple as pushing a button. Mearka shall be able to extract players' positions from the video uploaded by the user. Once the system has processed the video, it shall be simple for the user to export the data generated with the click of one button. The output from the system shall be positional data correlated with timestamps that can be used to create a positional map for each time "t", as illustrated by figure 3.1 and 3.2. If the user chooses to

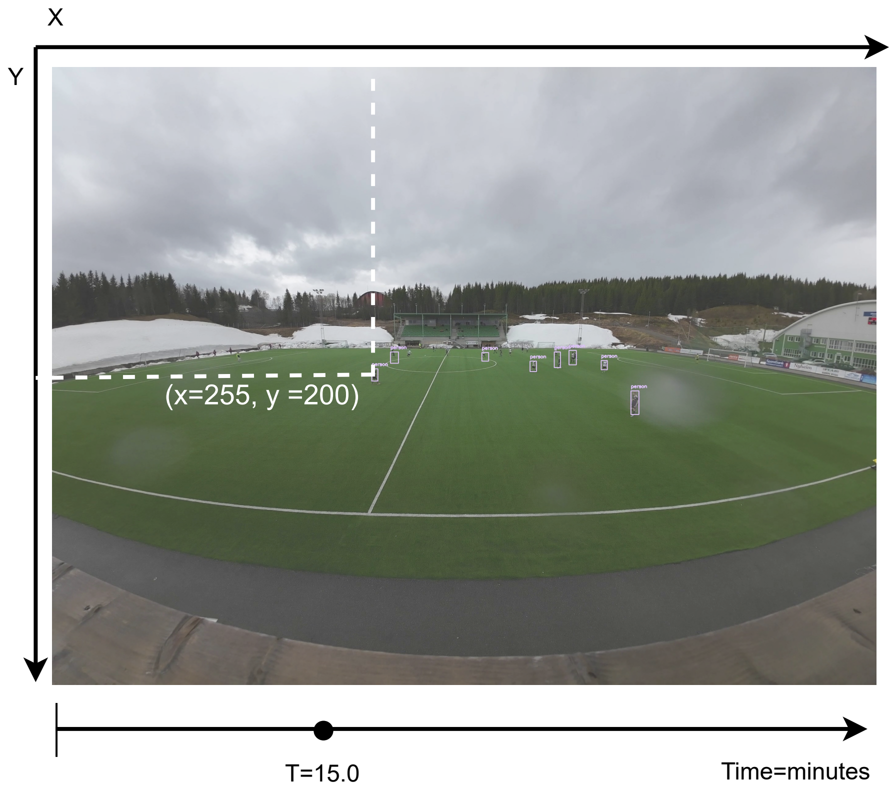


**Figure 3.1:** Example position detection x,y at time t.

delete their data, the functionality should be readily available to do so.

Regarding the non-functional aspects, Mearka shall be easy and intuitive to use, no matter which UI the user utilizes. Since Mearka shall be as cheap as possible to set up and operate, it shall be restricted to only choosing COTS components. When data is uploaded to the system, it shall be able to process and generate positional data within 12 hours. This means that within 12 hours, it should be possible to download useful soccer metadata from the system. As the user owns their own data, Mearka shall only store their data for as long as it is actually needed. It shall not be designed to keep data long-term. Mearka takes privacy compliance seriously and shall make it easy to upload and delete a user's data in the system at the user's will. In addition, the system shall implement some security protocol that restricts the data to only being accessible by the user that uploaded it.

The following chapter outlines and describes the design of Mearka and all its components.

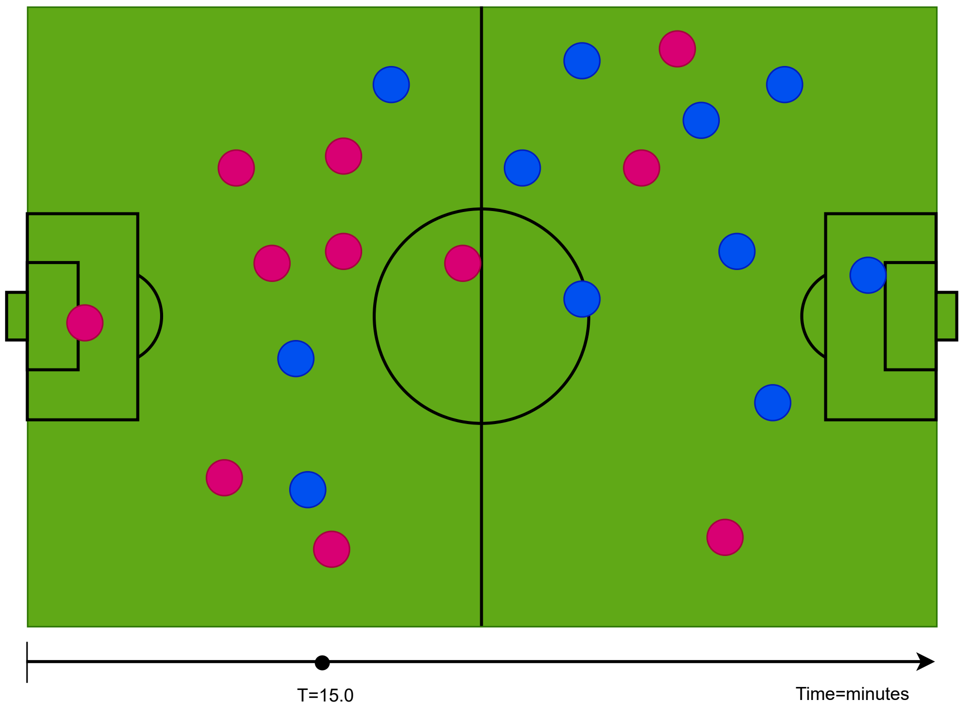


**Figure 3.2:** Example of positions at time t.

# 4 Design

This chapter outlines the design and architecture of Mearka. Section 4.1 provides an overview description of the system. Section 4.2 explains the requirements and findings when researching which camera to use during development. Section 4.3 outlines the design of the Mearka-app, while section 4.4 and 4.5 describe the web implementation and backend, respectively. Lastly, section 4.6 describes how the position detection component is set up and used in this thesis.

## 4.1 System Overview

Mearka is a distributed system that communicates using REST APIs for simplicity and convenience. REST APIs was chosen since all the information needed to fulfill a request is included in the request. A con of this is that all the data needs to be sent for each request, which means the same data might be sent in multiple requests. The Mearka-app offers an interface that enables the user, a main coach or similar, to record tags while the match or training session progresses. Since the web interface, backend, and position detection component all use REST APIs to communicate, it allows for flexible development of each component. Each component can be expanded with more functionality independently of the others as long as the API interface stays the same. The interfaces may be expanded, but it needs to keep the original interface endpoints, not to break the functionality of the other components that might use these. An overview of how the different components in Mearka communicate is illustrated in figure 4.1.

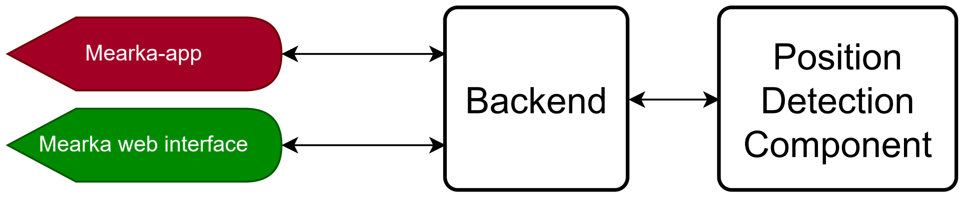


**Figure 4.1:** Mearka data and communication illustration.

For Mearka to meet the requirements set in chapter 3, the system design needs to fulfill the following:

1. **Easy-to-use:** Low effort to use the system, no matter if it is tagging through the Mearka-app or uploading a video to the website. The system needs to be as close to “one-click” as possible.

2. **One-click export:** Coupled with the system being easy to use, the resulting soccer metadata that Mearka generates should be easy to download once it is generated.

3. **Privacy and Security:** System users should only have access to their data. Uploaded data should be inaccessible to others than the one who uploaded it.

## 4.2 Choosing A Camera For Development

Mearka is designed such that the user, sports organization, or private person does not need a special camera to utilize its functionality. However, a COTS camera was chosen to help develop the development of Mearka by recording football matches in different resolutions, later used for evaluation of the system.

### 4.2.1 Requirements

Choosing a camera for this purpose must meet some requirements to be eligible. The following requirements are defined on the basis that a game is two times 45 minutes, with a 15 minute break.

1. **Battery life:** The battery (if any), needs to last for at least 105 minutes. 90 minutes match and 15 minutes break.

   - It should be possible to set up the camera, start recording, and not worry about it until the session ends.

2. **Field of view (FOV):** It needs a wide enough field of view to cover the entire football field.

   - FOV is often denoted as x° between 0° and 360°. The higher the number, the wider the FOV.

3. **Ease-of-use:**

   - **Operation:** The camera has to be easy to use and operate, no matter the technical level of the user.

   - **Movable:** Taking the camera to different arenas must be easy. The teams should be able to record wherever they play. The camera can not be constrained to only being installed and usable in one arena.

4. **Rugged:** It has to handle any weather the players can play in.

   - Waterproof.

   - Withstand hot and cold temperatures while still functioning.

   - Reliable.

5. **Common-of-the-shelf:** It should be a common-of-the-shelf camera.

### 4.2.2 Options

There are different types of cameras, many of which meet parts, if not all, of the requirements listed above. The first hurdle was to find a camera with a wide enough FOV to cover the entire field of view while being rugged and having good enough battery life.

#### Enterprice Camera Solutions

There are many different solutions for 180°and 360°cameras in the enterprise surveillance business. These are often installed at construction sites, warehouses, or other areas where a company wants video surveillance covering a wide field of view.

Examples of enterprise solutions that solve some of the defined requirements (section 4.2.1) is *Axis P3807-PVE network camera* [44], or *Oncam C-08 outdoor camera* [45].

**Axis P3807-PVE network camera** is a 180°camera the can shoot in resolutions as high as 4320x1920 at 25 or 30 frames per second (FPS). This camera has an IP rating of 66/67, which means it is protected from water, can be submerged up to 1m, under pressure, for a short period, and is completely protected against dust [46].

Unlike *Axis P3807*, **Oncam C-08 outdoor camera** is a 360°camera, but it can also output 180°video. *Oncam C-08* can record video with a resolution of 2160x2160 and has an IP rating of IP66, IP68, IP69K, which implies it is just as durable as *Axis P3807-PVE*.

Both **Axis P3807** and **Oncam C-08** are high-resolution cameras with a wide enough FOV to cover the entire field. They are also IP certified [46], which means they are built to be outdoors and to operate in various conditions. This means these cameras meet the **Field of View** and **Rugged** requirements defined in the list above. However, since they are both enterprise solutions, they have their own software to use them, which means they do not meet the first of the **Ease-of-use** requirement. Another issue is that both use Power over Ethernet (PoE) or need to be tethered somehow, making it much more challenging to move the cameras to other arenas if required. This means both cameras meet the **Battery life** requirement, as there is no battery, but neither camera meets the second requirement of **Ease-of-use**, in that the cameras are not that easy to use in different arenas. In addition, neither camera meets the **Common-of-the-shelf** requirement, simply by being enterprise for a different market.

### Consumer Cameras: >180°FOV

Like with section 4.2.2 above, there are many different options on the consumer market for cameras capable of one or more of the requirements. On the spectrum's widest FOV, there are multiple 360°cameras.

One of which is **Insta360 x3** [47], which is a 360°action camera that can shoot video in 5760x2880 resolution, is waterproof down to 10m without a dive housing, and operating range from -20°C to 40°C. The conditions in which players are out on the field will likely not be below or above these temperatures, so *Insta360 x3* meets the **Rugged** requirement, and being a 360°camera makes it meet the **FOV** requirement as well. From the 360°video, it is also possible to crop in and choose a narrower FOV. As the 5760x2880 resolution is the full 360°image, the quality will degrade the more the video is cropped in to get a narrower FOV. Because it is an action camera, it is meant to be simple to use, increasing the likelihood of it being **Easy-to-use** in operation. It runs on batteries, which, combined with mounting it to a tripod, reaches the **Ease-of-use** requirement by being easily movable to different arenas. Unfortunately, because it runs on battery, and the advertised battery life is set to be 81 minutes under perfect conditions, it does not meet the **Battery life** requirement.

Another camera from Insta360 is the **Insta360 One RS** [48]. It is also a 360°camera, and like the *Insta360 x3, one RS* can record video in 5760x2880 resolution; it can also be submerged down to 5m and have similar operating temperatures, which means it meets the **Rugged** as well as **FOV** requirements. *Insta360 One RS* is a modular[1] action camera that is supposedly easy to operate; therefore, it meets both of the **Ease-of-use** requirements in terms of operability and being movable. However, an advertised battery life of 82 minutes falls short of the required minimum 105 minutes of **Battery life**.

The last 360°camera that will be considered is the **Gopro MAX** [49]. It can record 360°video in 5376x2688 resolution, and like the beforementioned 360°cameras, it is possible to pan around to crop out the video as wanted, meaning it makes the **FOV** requirement. *Gopro MAX* can be submerged in 4.8m of water and advertised as rugged, which means it meets the **Rugged** requirement. The battery life is advertised to be 78 minutes [50]; therefore, it does not have enough **battery life** to reach the required minimum of 105 minutes.

These 360°cameras all meet most of the requirements set in section 4.2.1, because they can record everything around themself, run on battery, and are easy to set up and use. However, since they all use two or more camera sensors to achieve this 360°field of view, they all have the same issue with less than the required **Battery life**. This makes sense, as two sensors require more power than one, so the battery will deplete quicker than a one-sensor equivalent camera with the same battery.

1. "Modular" means this camera can change out the camera sensor while keeping the battery and processing module. There exists a 360°and a 4k regular camera-sensor module available. The regular 4K sensor has an advertised battery life of 75 minutes, which is why it is not considered.

### Consumer Cameras: <180°FOV

As cameras capable of shooting 360°use more than one sensor, which depletes the battery quicker, the next natural option is cameras with only one sensor but still a wide FOV. There are plenty of options in this category, in every price bracket, from action cameras to professional cameras, where it is possible to change lenses. However, focusing on the **Ease-of-use**, **Rugged**, **FOV**, and **Battery life** requirements in section 4.2.1, action cameras tick the first two by just being "action cameras". As these types of cameras are built to capture the action, they usually have a very wide **FOV** as well. Therefore, it depends on the **Battery life** (and the **Rugged** requirement, but we will get back to this) for which camera to choose.

A well known brand in the "action camera" segment is **GoPro**; therefore their **GoPro Hero 11** [51] is considered for this thesis. This camera can record video up to 5312x4648 resolution and record for 61 minutes, or an advertised 137 minutes if the resolution is lowered to 1920x1080 [50]. As *GoPro hero 11* has an advertised battery life of 135 minutes when recording in 1920x1080 resolution, it meets the **Battery life** requirement. Even though it is not a 360°camera, it can record video with a 122°horizontal FOV [52]. To summarize *GoPro hero 11* meets all the **Ease-of-use**, **FOV**, **Rugged**, and **Battery life** requirements, which would make it a superb camera for Mearka, if it was not for the overheating issues. Unfortunately, the last few generations of the GoPro Hero line of cameras have had issues with overheating [53, 54] when recording over a longer period of time. GoPro has issued a proposed fix for the overheating problems some customers have [55], which is, in short, to turn off features and record at a lower resolution.

### The Camera Of Choice

From the cameras above, an action camera with its **Ease-of-use**, wide **FOV** and **Rugged** (and reliable) design, with a good enough **Battery life**, is what would work best during the development of Mearka.

The camera chosen to be used during the development of Mearka is the **DJI Osmo Action 3** [56]. This camera is advertised to be rugged and reliable[2],

2. There are reports that the first batch of cameras, produced in September 2022, has a focusing issue on objects very close to the lens [57, 58]. However, even though the cameras purchased for the development of Mearka are from the September 2022 batch, it is not considered an issue in this thesis. The cameras were bought to record sessions from more than 50cm (estimation of how close is "too close") away, things further away from the lens are still in focus. *DJI Osmo Action 3* focusing issue seems to have been corrected beginning with the November 2022 batch and onwards, although there are still unconcluded reports

never to overheat, withstand cold temperatures, and record for longer than the required 105 minutes. It can withstand being submerged down to 16m without a dive housing. Additionally, it has operating temperatures from -20°C to 40°C, which meets the **Rugged** requirement. Record options include recording in 4096x3072 resolution, with a 155°-**FOV**, which is a wider FOV than the GoPro alternative. The camera is presumedly easy to use, based on the premise that "action cameras" are easy to use and very movable, which means the **Ease-of-use** requirement is met. The last requirement **Battery life** is met by *DJI Osmo Action 3* having an advertised battery life of 160 minutes. This thesis thoroughly tests this camera's battery life in section 6.1.1, but in short, the tests show that in all but the most draining setting, the battery performs well over 105 minutes. In summary, this camera meets every requirement set in section 4.2.1 and more. Another trick that the *DJI Osmo Action 3* can do is to stream the video to a server, which will be explained further in section 8.3.1.

## 4.3 Mearka-App

The Mearka-app is inspired by Muithu [40] and how Muithu used the app as part of their system. Muithu has an app that lets the head coach tag an event after it happened and add the player involved. After the match, retrieving the events from a video recording, accomplished by consumer-grade cameras (GoPro Hero 2), and playing them back, is possible. Like Muithu, Mearka has a simple app that is meant to be used by the head coach in real-time, and the Mearka-app only needs to do one thing: Tag events as they happen on the field. For the Mearka-app to be used, the following requirements need to be met:

1. **Tag Events:** The Mearka-app needs to be able to tag events.

2. **Ease-Of-Use:** The Mearka-app should be self-explanatory and intuitive to use.

3. **Non-Intrusive:** It is designed to be used by the head coach during a match or training, so it must require as little focus as possible.

4. **Low Effort - High Reward:** For a coach to use the Mearka-app, it should require low effort while still giving a lot of value.

5. **Rugged UI:** The Mearka-app is developed in northern Norway, where

on this. Some users have also fixed the issue themself [59] by opening the cameras and physically adjusted the lens.

there is a lot of weather. The Mearka-app needs to have a functional UI even in rain, snow, and frosty conditions, as well as when sunny and dry.

### 4.3.1 System Design

The Mearka-app is built on the premise that it can connect to the backend when used; without this connectivity, the Mearka-app cannot be used. In short, the Mearka-app is a UI that allows users to interact with the backend to create tags aligned with time; however, it does not record any video or store tags in the app. On Mearka-app startup, it is possible to choose to start a recording. This sets the point in time from where the tag-offsets are calculated for that session. For this POC, Mearka assumes the record time starts at kick-off, as stated in section 1.4.

When the Mearka-app starts recording, it sends a notification to the backend stating it wants to start a recording. If the backend is up and responds as expected, the Mearka-app allows the user to start tagging. Tagging an event is done by sending a request to the backend with the click of a button from the user. The request lets the backend know something of interest happened, and the time offset is stored. From then on, the Mearka-app will enable the tagging of events as long as the session is ongoing. More details about how a tag is created and stored are described in section 4.5. Since the Mearka-app can tag events, as described, requirement to **Tag Events** is met.

Once the recording (session) is stopped, the backend gets a notification that the session is over. Upon the arrival of the request, any soccer metadata associated with the user is collected and returned as a JSON file that contains a list of tags. The soccer metadata file can be used to more easily see where in the video (recorded separately) something of interest happened, or to be used with something like Dárkon for further analysis. Figure 4.2 illustrates the complete data flow in the Mearka-app.

### 4.3.2 User Interface

To comply with the **Easy-To-Use** requirement listed above, the Mearka-app is designed with a simple, intuitive user interface (UI). An overview can be seen in figure 4.3. On startup, the Mearka-app displays a single button that lets the user choose when the video recording starts, figure 4.3a illustrates this. This is important because the timestamp of a tag is calculated as the offset from when "Start recording" is pressed, as described in section 4.3.1.

Once the Mearka-app and the backend agree to start recording, meaning a

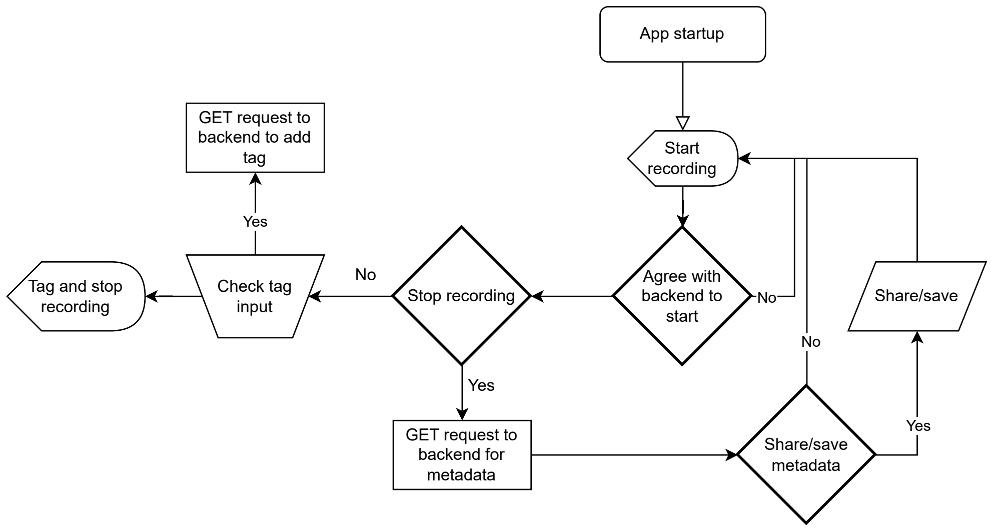
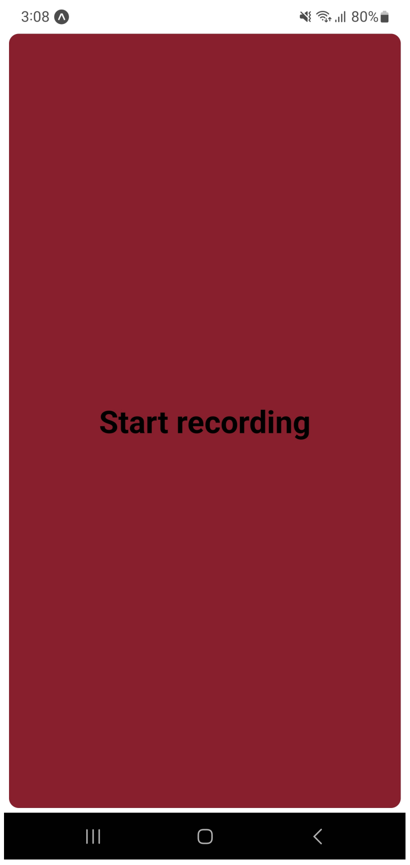


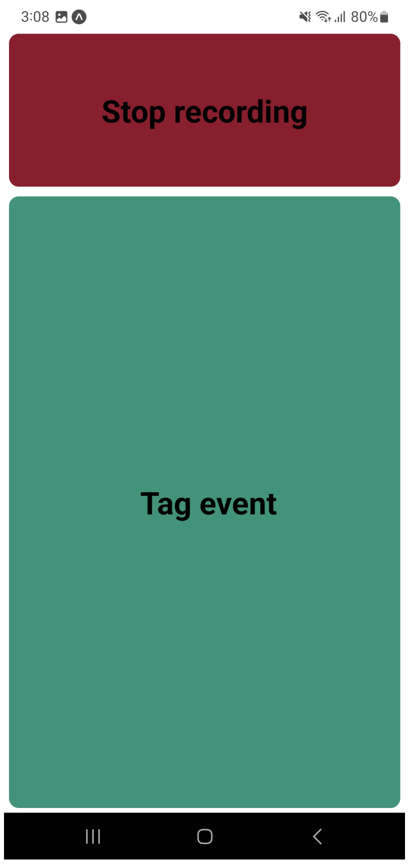
**Figure 4.2:** App flowchart

session has begun, it is possible to start tagging events. This is accomplished with the “Tag event” button displayed while recording, as illustrated by figure 4.3b. On a button press, the button changes for a split second to give visual confirmation that the button actuated on the press. How the button gives a visual indication that it has been pressed is illustrated in figure 4.3c.

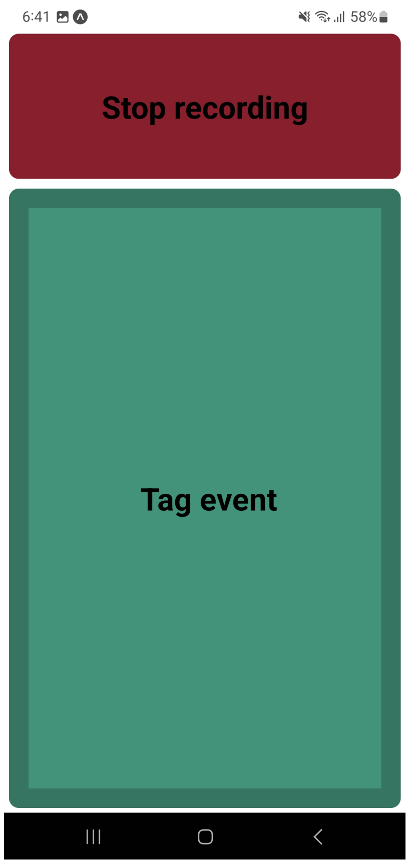
The “Stop recording” button, displayed at the top of figure 4.3b, can be pressed to stop the session. Pressing “Stop recording” notifies the backend that the session is over and prompts the backend to send any soccer metadata it has, tags, and otherwise, back to the Mearka-app. Upon receiving the metadata, the Mearka-app prompts the user to share it wherever needed. Some sharing options can be sending it as an email attachment or using any possibility available through the default sharing screen on the given phone operating system. On Android [60], sharing the received metadata is as depicted in figure 4.3d.



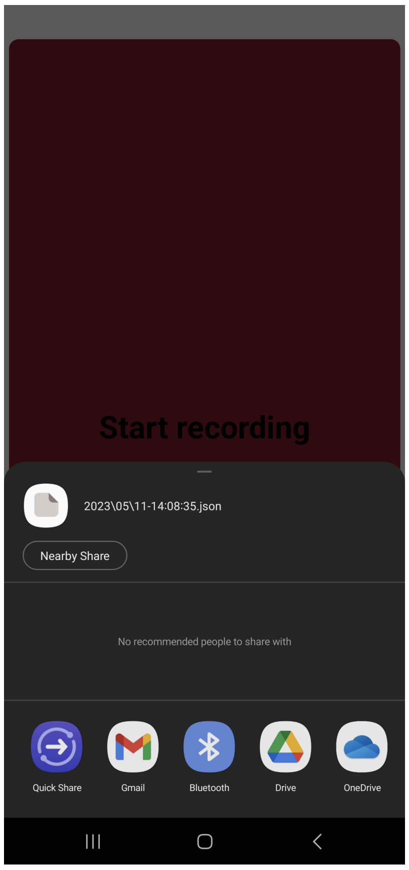
**(a)** Mearka-App Startup.



**(b)** Session is in progress.



**(c)** Session is in progress - Tag event.



**(d)** Session is over and it is possible to export soccer metadata.

**Figure 4.3:** Mearka-app UI overview

.

## 4.4 Mearka Web-Interface

Mearka is camera agnostic for video recordings of a match or training session. This allows the user to use the system with any available video source. A video source can be anything from recording with a phone to buying a professional camera or something in between, for example, choosing one of the options described in section 4.2.

The web interface illustrated in figure 4.4 is the user's "portal" to use Mearka. Through this portal, Mearka could allow the user to expand on the tags and other metadata created. More about metadata and its content is described in section 4.5.2. Mearka allows for uploading a video (or multiple video sections) through the web interface for position detection.

Another option could be to add players and actions that can be used for more detailed tagging. This would allow the tagging process to be tailored to the needs of the sport and coach as needed. More on expanding the web interface can be read in section 8.3.3.

However, this thesis has chosen to focus on implementing functionality that finds players' positions on the field. Finding the positions of every player on the field is very time and labor-intensive as it has to be done manually today. To find the positions manually, the video must be inspected frame by frame, and note the positions of each player one by one. How the positions are found in Mearka is described in section 4.6.

With this positional data, it is possible to analyze player patterns and how players position themself on the field, including when they do not possess the ball. These patterns and player positions could be an excellent asset to the coach for refining the team playing style or better positioning players. Intuitively it is a lot of work to tag this data, even if it could be of great value. Therefore, Mearka strives to make this process as easy and automated as possible while giving the same valuable data.

### 4.4.1 User Interface

The web interface is how users interact with the Mearka system. To make the interface as useful as possible, the web interface needs to meet the following requirements:

1. **Easy-to-use:** Intuitive UI that makes it easy to use the system no matter the technical level of the user.

2. **Automatic:** Automated process to remove as much manual labor as possible.

The user interface consists of a video player where it could be possible to replay video, as well as a template for where the positional data could be visualized. This thesis focuses on the positional detection functionality, and so the essential part of the UI displayed in figure 4.4 is the button to upload a video.

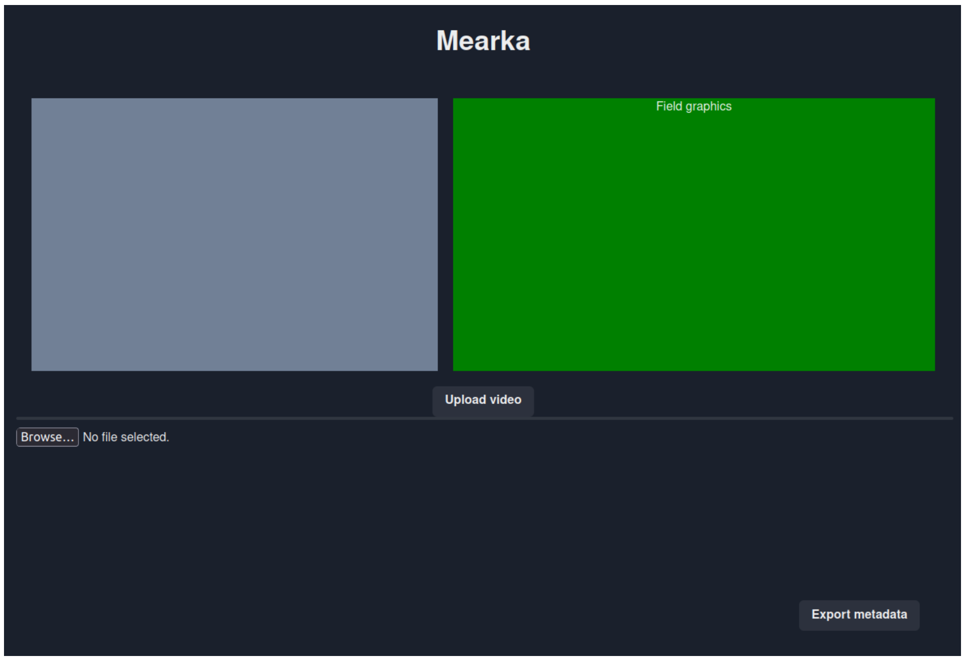
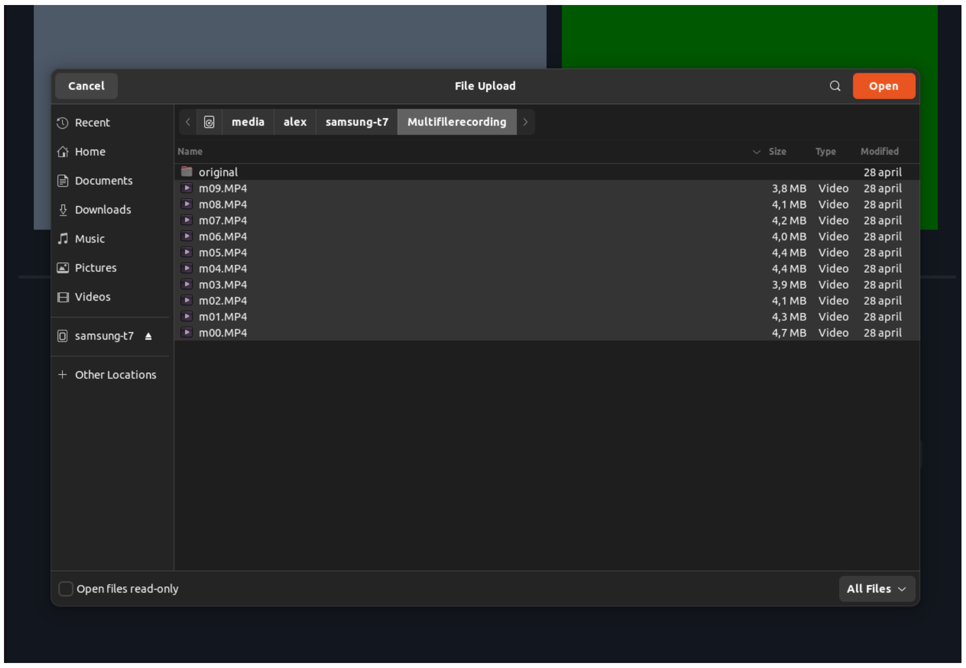


**Figure 4.4:** Web UI.

When the button is pressed, a dialog box opens, and the user can choose one or more videos that will be uploaded, as figure 4.5 illustrates. The backend only works on one video at a time. This means that the system considers a multiple file upload as being multiple parts of one video. Uploading multiple unrelated videos will work, but the resulting data will not make sense unless the same video clips are concatenated, or uploaded in separate sessions for position detection.

Once the video files are selected, the user has to confirm (illustrated in figure 4.6) that the videos are sendt to the backend. If the user accepts this, the videos are transferred to the backend before it starts detecting positions. The user is notified that the position detection has started, through a message, shown in figure 4.7.

From the position extraction has begun, there is an option to remove the data



| Name | Size | Type | Modified |
|---|---|---|---|
| original | | | 28 april |
| m09.MP4 | 3,8 MB | Video | 28 april |
| m08.MP4 | 4,1 MB | Video | 28 april |
| m07.MP4 | 4,2 MB | Video | 28 april |
| m06.MP4 | 4,0 MB | Video | 28 april |
| m05.MP4 | 4,4 MB | Video | 28 april |
| m04.MP4 | 4,4 MB | Video | 28 april |
| m03.MP4 | 3,9 MB | Video | 28 april |
| m02.MP4 | 4,1 MB | Video | 28 april |
| m01.MP4 | 4,3 MB | Video | 28 april |
| m00.MP4 | 4,7 MB | Video | 28 april |

**Figure 4.5:** Web UI - upload multiple files.

uploaded to the backend through the button displayed in figure 4.8. That figure also shows the option to extract the soccer metadata once the position extraction is complete.

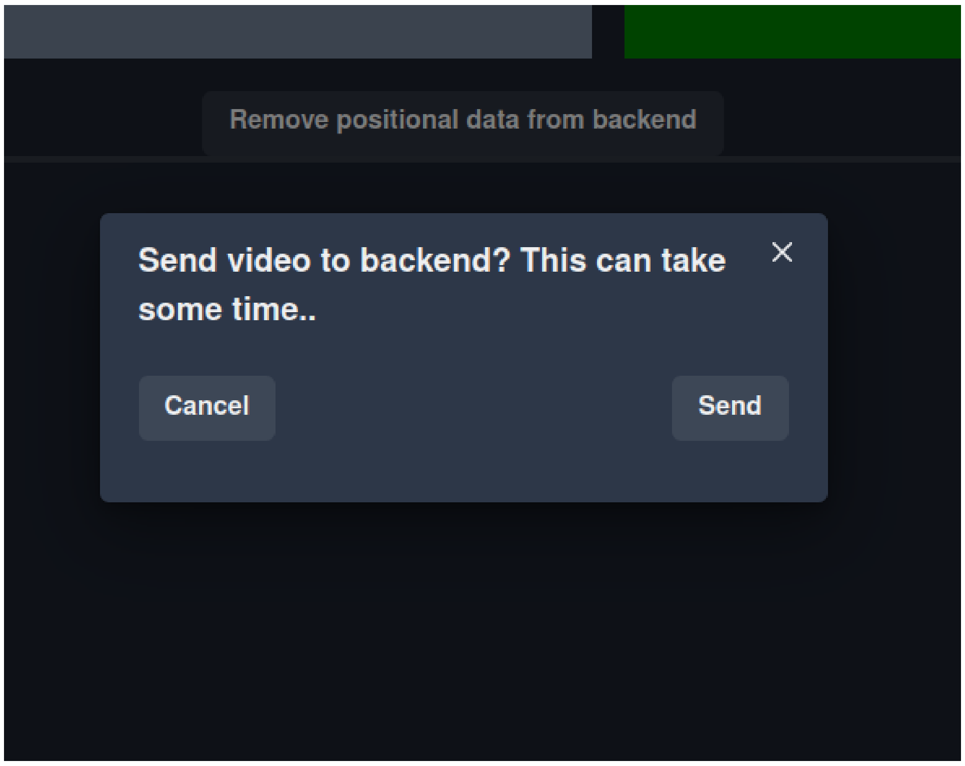

Remove positional data from backend
Send video to backend? This can take some time..
Cancel
Send

**Figure 4.6:** Web UI - confirm send to backend.

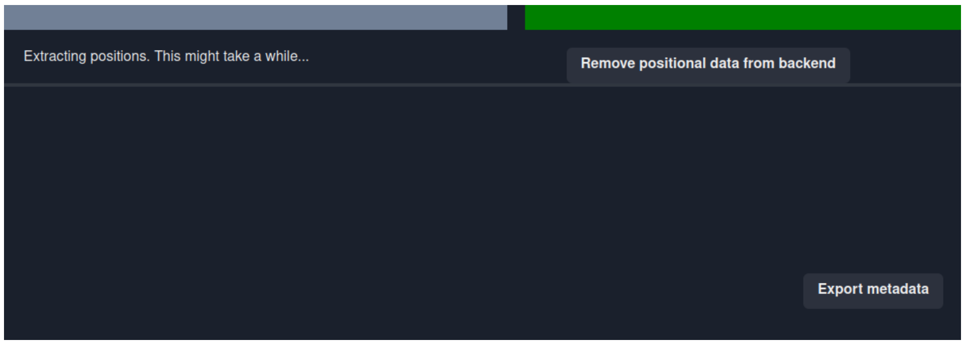
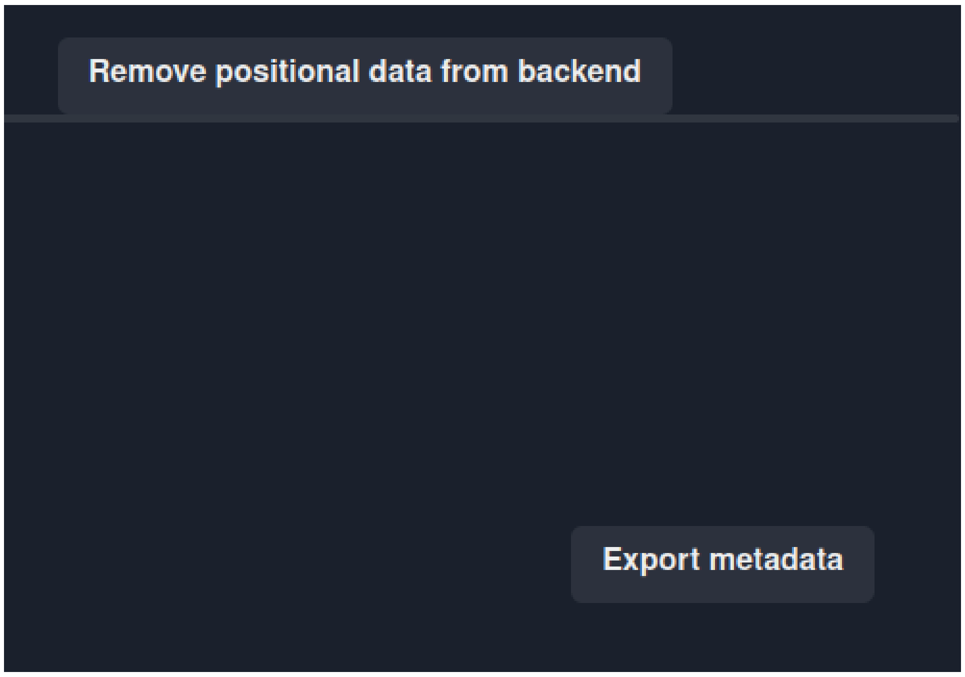

Extracting positions. This might take a while...

**Figure 4.7:** Web UI - extracting-positions.



Following is a summarization of the Mearka web-interface and the interface requirements defined for it in section 4.4.1. The Mearka web UI gives the user only one option to upload a video, as illustrated by figure 4.4. This, together with no further input, qualifies the web interface to have reached the **Easy-To-Use** requirement in section 4.4.1 since there is only one option. The system detects positions automatically, without additional input from the user other than the uploaded video, thererfore, the **Automatic** requirement is also met.

## 4.5 Backend

The backend plays a critical role in the distributed system, Mearka, acting as a conductor, orchestrating all the requests and inputs from different components. It ensures that the Mearka-app and web interface can deliver the promised functionality. The data flow between the Mearka-app, web interface, backend, and position detection components is illustrated in figure 4.1.

Mearka does not require the user to specifically "log in" to use the system, but it does require that requests have an attached universally unique identifier (UUID). Because there is no way for the backend to know if one UUID is the same user as another UUID, it does not retain any data longer than it has to, nor does it allow sharing of data between UUIDs. Mearka lets the user upload a video to get positional data and use the Mearka-app to tag events relative to the start of the video. However, since there is no login, the user must export the data from the Mearka-app and import it into the Mearka web-interface to merge the tags and positional data. The resulting soccer metadata can then be exported before importing it into something like Dárkon for further use and analysis [1].

### 4.5.1 REST API

In Mearka, the backend is built as a REST API server with endpoints that the system components can use. Figure 4.9 lists an overview of the endpoints available. Each item in the list is part of a URL, where the sub-list items build on the URL they are nested beneath.

The backend utilizes the REST design principle for developing its API because it simplifies the development process as it can assume that it will get all necessary information through the request itself. This is different than keeping a state between requests from a user since it then would need to check whether it has the necessary information to fulfill the request.

All the endpoints end with a ":uuid", which is a Universally Unique Identifier (UUID). This UUID is created by assigning a big and unique number to each "user". In Mearka, this is more associated with a specific communication session than a user through the Mearka-app or the web interface. Because of this, a user can use the web interface and the Mearka-app simultaneously, but both devices will be assigned a different UUID. It is possible to merge the two metadata files by uploading the file from the Mearka-app to the web interface. Given that the web interface has just detected the position on a video, the uploaded metadata will be merged with the positional data already on the backend. Once merging is complete, the newly updated metadata can be exported through the "export" button in the lower right corner, illustrated in figure 4.8.

An example endpoint is "/app", and its nested "/record/:uuid". This describes the API URL "/app/record/:uuid" used by the Mearka-app. Adding ":uuid" to the end of the URL means that the endpoint expects an assigned UUID for each request. This is so the backend knows to which user to add or create data.

Almost every endpoint will reject a request if it does not contain a UUID. If the interface does not have a UUID assigned, it calls the "/utils/get-uuid" endpoint to get an assigned UUID. The newly generated UUID is then added to the response such that the client interface, Mearka-app, or web can use the same UUID to add additional data. This, and the "/app/record" endpoint, are the two endpoints that do not refuse a request without a UUID.

When it comes to building APIs, representing Representational State Transfer (REST) and GraphQL are two popular options that developers often consider. REST is a well-established standard for building web services that use HTTP as the communication protocol [19]. It is resource-oriented and relies on pre-defined HTTP methods to manipulate resources, such as GET, POST, PUT, and DELETE. On the other hand, GraphQL is a relatively new technology developed by Facebook to address some of the limitations of REST [61]. GraphQL is a query language that allows clients to specify exactly what data they need and receive only that data in response, making it more efficient and flexible than REST. While REST works best for simple, straightforward APIs, GraphQL is better suited for complex, data-intensive applications where performance and flexibility are critical.

An evaluation between GraphQL and REST API shows that REST is up to 50.50% faster response time [62], which means it is better when multiple requests access some data. This also means it is better suited to be used in time-sensitive applications, like when the Mearka-app tags an event. Choosing to implement the backend as a REST API was mainly chosen for its stateless property. Because REST is stateless, the backend can assume that each request has all the information needed to fulfill that request. This simplifies the development as each endpoint only needs to require what that endpoint needs. Another reason is that the backend only requires simple HTTP requests to fulfill most of its duty besides uploading videos, therefore, a simple REST API is sufficient.

```
/app
        /record/:uuid
        /tag/:uuid
/web
        /upload-video/:uuid
        /metadata/:uuid
/utils
        /get-uuid
        /get-user-metadata/:uuid
        /remove-user/:uuid
        /remove-user-data/:uuid
        /check-user-files/:uuid
        /begin-processing/:uuid/:numfiles
```

**Figure 4.9:** Backend API endpoints.

### 4.5.2 Soccer Metadata

The premise of Mearka is to help the user get useful metadata from the video footage they have provided, and so the backend is developed with this in mind. To store soccer metadata, the backend uses a map where UUID are keys, and the value mapped to each key is an object containing all the possible soccer metadata for a user. A more detailed description of how the data is stored on the backend is described in section 5.4.

When the Mearka-app wants to record, the backend generates a UUID, and the soccer metadata is set up for that UUID in the local map of UUIDs on the backend. This UUID is also returned to the source of the request, so it can be used with subsequent calls to map new tags to the correct UUID.

As figure 4.9 lists, the endpoints for the web also need a valid UUID. If none is present in the request, the request is invalid. This means that the web client needs to ask the backend for a UUID, which will then be used when uploading video or soccer metadata.

### 4.5.3 Mearka-app

As described in section 4.3, the Mearka-app needs to agree with the backend before starting a session to tag events. This is done so the backend can control how metadata is created and formatted. The Mearka-app sends a request to the backend, which stores the time of day when the request arrives.

From then on, the Mearka-app can tag events by sending a request that includes the assigned UUID, and the backend will add a tag to that UUIDs metadata. The tags created with the Mearka-app will not contain any additional information other than the offset into a separately recorded video. This is done to make the Mearka-app non-intrusive to use by a coach, while still giving value with time offsets into a video that can show what happened.

Once the match or training session is over, the Mearka-app sends a request (with the stored UUID) to get the resulting soccer metadata stored on the backend. When the backend retrieves the soccer metadata from memory and sends it back to the Mearka-app, the user and the metadata are removed. This is done because a request to start a new recording from the Mearka-app will always generate a new UUID. The old data is irretrievable after the Mearka-app is done with a session since the Mearka-app forces the user to start a new session once a session is complete. Because of this, removing a user and data after the Mearka-app completes a session is done to mitigate storing unused data.

### 4.5.4 Mearka Web-interface

The web interface interacts with the backend whenever the user uploads or requests a file. As described in section 4.4, the user can upload a video, or multiple videos, with just a few clicks. Once the user has chosen which video files to run position detection on, the front end lists the files and sends them individually to the backend. The backend receives these files one by one and stores them in a shared folder named the same as the UUID of the request. This folder is shared between the backend server and the position detection component of Mearka.

Once the files are transferred, the frontend notifies the backend that every file transfer is complete to start position detection on the video(s). The backend is notified by sending a request to a different endpoint with the same UUID, with information on how many files have been uploaded.

Depending on how many files were uploaded, the backend acts accordingly. If multiple files are transferred, then there is a need to concatenate the videos into one longer video. The functionality to concatenate videos is present in the position detection component of Mearka. Therefore the backend requests that the files present in the shared folder, inside the subfolder named the same as the UUID, be concatenated. More on how this is done is described in section 4.6. Once the video is concatenated, a new request to the position detection component is sent to start position detection on this new file.

After the files are sent, the user is notified that the system is working on detecting people's positions in the video. The Mearka web-interface enables the soccer metadata to be downloaded after the system is done processing the video. To get the soccer metadata, a request for the most up-to-date soccer metadata from the backend is sent to "/utils/get-user-metadata/:uuid", where the actual UUID is appended at the end. The backend uses the UUID from the request to retrieve the soccer metadata from its local map of data and respond with the metadata file, which is automatically downloaded to the client's machine.

### 4.5.5 Position Detection Component Communication

Communication between the backend and the position detection component is very important in Mearka, as these components are what enable the system to generate the most valuable data. The backend and the position detection component both have a reachable REST API that other components in the system can reach.

The user can upload a video through the web interface as described in section 4.5.4, and the backend stores this video in a shared folder between the backend and the position detection component. Once the file transfer is complete, the backend requests the position detection component to detect positions in the video through an API endpoint. After the position detection component is done detecting positions, the file containing positions is sent to the backend. The backend appends this information to the soccer metadata for the UUID that requested it.

In the case where the user uploads multiple files, the backend requests that the position detection component concatenate the videos. The position detection component then responds with the filename of the concatenated video. The backend then requests the position detection component to start detecting positions on the video, in the same shared folder, with a request containing the same UUID of the user. This also refers back to using REST APIs which are stateless, meaning they expect all relevant information to be present in the incoming request.

## 4.6 Position Detection Component

The position detection component enables Mearka to detect positions in the video. This component consists of a video concatenation script, a program detecting positions in the video, and a server. The script concatenating videos consists of libraries that utilize system calls to list files in folders and start the program FFMPEG with the appropriate command line arguments. The server is a middle layer, receiving requests from the backend server and responding with the results.

No matter which script or program is activated, the data flow is the same, as illustrated in figure 4.10. The user uploads the video to the backend, which stores the video in a shared volume between the backend component and the position detection component. Next, the backend sends a request to the position detection server (section 4.6.3) to start concatenating video or detect positions. The appropriate script or program is activated, and the result is returned to the backend. From there on, the Mearka web-interface (section 4.4) enables the download of the updated soccer metadata from the backend, as illustrated by point 6 in figure 4.10.

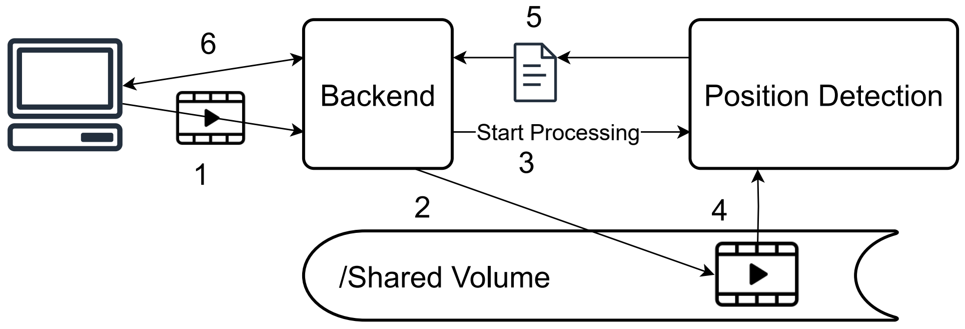


**Figure 4.10:** Data flow when using the Position Detection component.

### 4.6.1 Concatinate Video

As some cameras split a longer recording into smaller video files, there is a need to concatenate those files back into a full-length video[3]. The script used in Mearka takes a folder path as input and returns a concatenated video in the same folder as the input files. It starts by listing all files present in the

3. *DJI Action 3* is a camera that splits a longer recording into smaller 4GB files. Although it is not confirmed, and the camera formats the memory card to exFAT format (max file size of 128PB [63]), the choice to split a recording every 4GB likely stems from the older FAT32 filesystem days. On the FAT32 filesystem, one file can not be larger than 4GB [64].

folder and filtering out anything that is not a video[4]. This script uses FFMPEG (section 2.3) to concatenate a list of videos into one longer video. Once the list of filenames to concatenate is obtained, the script creates a file containing each filename with specific keywords so FFMPEG recognizes it. Once this file is created, FFMPEG is run with the appropriate command on the operating system, concatenating the videos listed in the file into a new video stored in the UUID folder.

FFMPEG concatenates the videos in the same line order that the videos are listed in this file, which means that the videos listed are appended after each other in the same order as the file lists them. For this script to be useful, the videos must be named ascendingly, alphabetically, or numerically. The list of video filenames is sorted alphabetically before being written to the file. Once done, the script has a command line interface option to clean out all the smaller video files that make up the more extensive full-length video. This could potentially be helpful if storage is an issue and the user does not want to store the same video twice.

### 4.6.2 Position Detection

Detecting positions based only on video input is part of Mearka functionality that can potentially give the most value. The position detection functionality is further evaluated in section 6.2. This program combines the OpenCV with the CVlib library to read images from a video file and detect pixel positions in the frame. Mearka do not convert from pixel coordinates in the frame to field positions in the current implementation. A possible solution to this could be to implement something similar to "Pixel2Field" by Liang Peng, which is able to automatically transform every pixel coordinate to their scaled 2D field position [65]. Implementing this is left for future work due to time constraints.

OpenCV interprets the video as a "webcam", which means it is possible to load one frame at a time in memory by manually choosing to load the next frame in the sequence. For each frame, CVlib is used to do the object detection on the loaded frame.

As explained in section 2.4.5, this thesis uses the default model that CVlib has set up to do object detection, and that is YOLOv4. This model is trained using a supervised method [42] on the COCO dataset from Microsoft [27]. YOLOv4 not only detects people, but it can also detect phones, monitors, balls, different fruits, etc.

4. In this thesis, "video" refers to either .mp4 or .mov files.

Before frames from the video are loaded into memory, the total number of frames in the video and the frame rate of the video are noted. OpenCV has methods to figure out both once the video is loaded. This information is used for calculating the offset in the video where the positions are found, using equation 4.1. For each frame, the time offset into the video is calculated and appended alongside a list of pixel positions.

$$\text{Time offset} = \frac{\text{frame number}}{\text{FPS}} \tag{4.1}$$

Since YOLOv4 detects more than just people, the output of object detection has to be filtered before being stored. CVlib returns a list of labels describing what it has detected, a list of pixel positions, and a list of confidence levels. The labels are iterated over, and if it has detected people, those positions are stored in a position object together with the time offset into the video.

After the program has tried to detect positions on the entire video, the result is a list of position objects containing a time offset and a list of pixel positions for each offset. This data is stored in a file, and the server responds to the backend by sending over the positional data.

### 4.6.3 Position Detection Component Server

To connect the position detection component to the rest of Mearka, it has its own server. This is a simple REST API server that opens up endpoints to enable concatenation and position detection through some API calls. These endpoints make it possible for the backend to request the concatenation of videos in a folder or to do position detection on a video. The server also helps return the results from the abovementioned script and program to the backend so that the soccer metadata can be updated accordingly.

Like with the backend API, the position detection server endpoints also require that the UUID is included in the URL. The UUID is extracted from the URL and used to create the path to the folder for that user, which is shared between the backend and this server.

Upon a request to concatenate video or start position detection, the server extracts the UUID, creates the necessary paths, and does an operating system call to start the relevant script or program. Once the processing is done, the server returns the associated result to the backend.

## 4.7 Summary

This chapter has presented the design of Mearka. The system is designed with a distributed architecture, using REST API interfaces to communicate between components. REST APIs were chosen for their simplicity during development since each endpoint can assume that any data they need will be provided with the request. The core component in the system is the backend, which controls how metadata is formatted and bridges the communication between client devices and the system's capabilities.

As described in section 4.3, the Mearka-app consists of two buttons once it is recording tags. The Mearka web-interface (section 4.4) consists of a few buttons, one for uploading video and the other for exporting soccer metadata from the backend. The interfaces for the Mearka-app and web interfaces, depicted in figure 4.3 and 4.4, strives to meet the **Easy-to-use** requirement, set in section 4.1. As the web interface lets the user download the newest version of the soccer metadata through a click of a button, depicted in figure 4.8, the **One-click export** requirement in section 3 is met.

To meet the **Privacy and Security** requirement, that states the system should separate users' data from each other, both the backend and the different interfaces ensure unique UUIDs are used when generating metadata. The UUID is generated by the backend on each reload of the web interface and on each new recording session by the Mearka-app. Since the system uses the UUID to map metadata to a user, meaning one user only has access to their own metadata, requirement **Privacy and Security** is met.

The following chapter builds on the design of Mearka by detailing the implementational details of the system.

# 5 Implementation

This chapter presents the implementational details of Mearka. In particular, section 5.1 gives an overview of the system implementation, such as programming languages, special libraries, etc. Section 5.2 covers the implementational details of the web interface. Section 5.3 describes the app implementation. Section 5.4 details the implementation of the backend, while section 5.5 covers how video is concatenated, and position detection is accomplished.

## 5.1 System overview

Figure 4.1 illustrates a brief overview of how Mearka is set up. However, figure 5.1 illustrates the system architecture in more detail. This figure depicts the four components that Mearka comprises: The frontend (1), backend (2), position detection component (3), and a shared volume (4) between the last two. Each sub-component of the figure is denoted with (<component number>.<sub-component number>), e.g., the Mearka-app is denoted (1.2) within figure 5.1.

The Mearka frontend comprises a phone app (Android) and a web interface, sub-components (1.2) and (1.1) in figure 5.1 respectively. The app is developed with React Native and the Expo library to make development easier. The Expo library enables the developer to observe changes in the Mearka-app on a device, live, during the development process. This is contrary to building the app to observe changes or utilize a virtual device that emulates the Mearka-app on the computer. React Typescript with the Chakra design library is used to develop the web interface.

The backend component is wholly written in Golang [66], with the Gin web framework [67] to help ease the implementation of a REST API. Golang was primarily chosen for its small language syntax, good support for HTTP servers, and speed. It was also selected because I wanted to learn Golang for this thesis. There are two separate REST APIs on the backend, one that communicates with the frontend (2.1), as well as one for internal communication with the position detection component (2.2).

Python [68] was chosen for the position detection component because it is widely used in scientific and machine learning applications, which means multiple libraries are accessible to help ease the implementation of this prototype. Flask [69] is used to run the position detection server, implementing a small REST API (3.1) that enables communication with the backend. In addition, the server can start the concatenation of a video (3.2) as well as begin position detection (3.3) upon request. The shared volume (4) between the backend and position detection component is located on the filesystem, where both components have access to it.

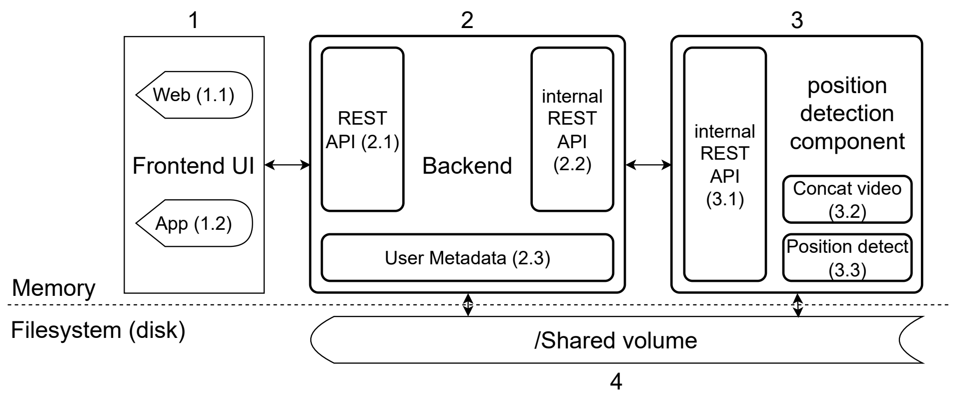


**Figure 5.1:** Component system overview.

## 5.2 Web

The web interface is implemented using React typescript [70] in combination with Chakra UI [71] to simplify the design. React typescript is a library that wraps around HTML. It lets the developer create and reuse custom components to simplify the development of a website and enable the use of pre-made components implemented by the library. A component can have input that will change the information in the generated HTML based on that input. This allows the component to be dynamic and potentially reused in multiple places on the website, which minimizes the need to copy-paste HTML. Figure 5.2, depicts a simple example of a component created with React that uses a component argument. Once the component is created, it can be used as depicted in figure 5.3, passing the appropriate arguments.

Chakra is a design library used on top of React Typescript to simplify the design process while developing the website. It works by having pre-made components in React that have styling already applied. Using the Chakra components, the design is set “out of the box”, meaning less time is spent on visual design.

```
import { Flex, Text } from "@chakra-ui/react";

type args = {
    numberArgument: number,
}

const exampleComponent = ({ numberArgument }: args) => {
    return (
        <Flex>
            <Text> "This is a number: ",{numberArgument} </Text>
            <Text> "This is twice the size of the number: ", {numberArgument * 2}</Text>
        </Flex>
    );
};
export default FieldPlayer;
```

**Figure 5.2:** React typescript component example

```
import exampleComponent from "../components/exampleComponent";

const website = () => {
    return (
        <exampleComponent numberArgument={10}} />
    )
}
```

**Figure 5.3:** Use a React Typescript component.

### 5.2.1 Api calls

For the frontend to be useful, it needs to communicate with the backend. This is done through the REST API available on the backend, illustrated as component 2.1 in figure 5.1. Before doing any API calls, the web interface makes sure it has a valid UUID. If it does, then it can contact the appropriate API endpoint. If not, it acquires a UUID by requesting one from the backend via a GET request to the "/utils/get-uuid" endpoint.

The most important endpoint for the web interface is the "/web/upload-video:uuid". This endpoint is used to upload videos from the client frontend, to the backend, to be used for position detection. When uploading a video or multiple video segments to the backend, the frontend counts how many files are being sent. It then creates a multiform that is sent to the backend such that it can transfer each video. Once the last video (if multiple) is sent, the frontend sends a notification to the backend to let it know the frontend is done sending videos and how many videos were sent. Once the videos are on the backend, the backend ensures multiple video segments are concatenated into one and starts position detection.

After position detection is complete, it is possible to download the available soccer metadata from the backend. To download the newly generated soccer metadata, the web interface has an "export metadata" button in the interface's lower right corner, as depicted in figure 4.8. This sends a request to "/utils/get-user-metadata" API endpoint on the backend, which returns and downloads the soccer metadata as a JSON file.

## 5.3 Mearka-App

As described in section 5.1, the Mearka-app is developed using React Native with the Expo framework [72]. React native is used to develop multiplatform apps rather than developing for one operating system specifically. This eases the development process by being developed once and choosing a platform later, increasing the portability of the app.

Figure 5.4 illustrates how React Native works when developing an app with a button and a text input field in a portable way. In this example developer environment, the implementation states a button or input field should be present. Once the app is ready to be built, React Native swaps out these components with the platform-specific equivalent. The platform-specific components are illustrated in the "Android" and "IOS" boxes in figure 5.4. The portability is the main reason why this thesis developed the Mearka-app with React Native.

Expo is a framework that enables the developer to see the app's development live as changes are being introduced. For this to work, Expo needs to run a server on the development computer, and both the computer and the device (phone, tablet or otherwise) needs to be connected to the same network. The

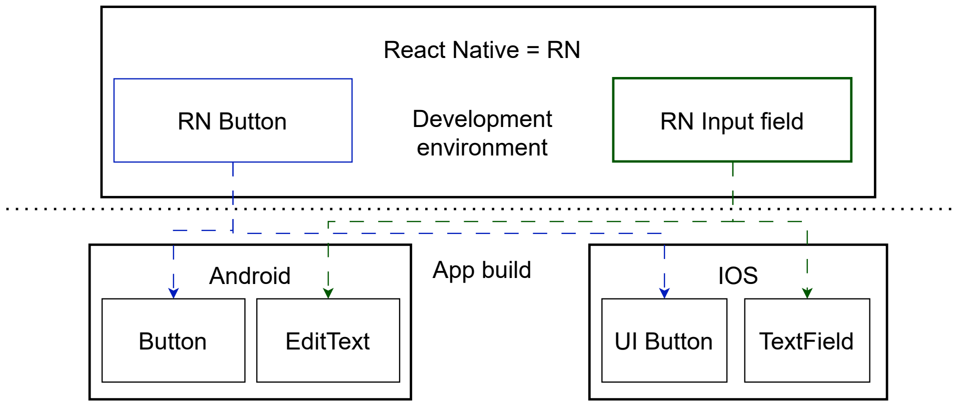


**Figure 5.4:** React Native example.

device needs to run the "Expo Go" app, which lets Expo stream the app to the device live. This allows the developer to observe changes done, in real-time, instead of using an emulator or building the app and transferring it over to the device for each change.

Communication with the backend is done through API calls to the appropriate endpoints (listed in figure 4.9) at the backend. Unlike other endpoints on the backend, the app does not need to specifically request a new UUID if it does not have one on app startup. This is because starting a recording prompts the backend to generate a new UUID that the app gets in response. The UUID is stored locally on the device and used in every later request to append soccer metadata to the correct UUID.

Since the backend uses a REST API, all information needed must be included in each request. Because of this, requesting to start a recording is done with an HTTP [73] POST request, where the body contains a boolean value informing the backend if the app wants to record or stop recording. Without this, the backend cannot know if it is a request to start or stop recording.

## 5.4 Backend

The backend is implemented using Golang [66] and Gin web framework [67] to simplify setting up a REST API server. Golang was chosen because of its built-in concurrency and robust standard library. The backend ties Mearka together, so it must respond to requests, even when waiting for work, such as position detection from the position detection component. As a request comes in, the current thread starts working on it, and a new thread is started to listen for new incoming requests. Golang is a compiled language, which means everything it needs to run is compiled into the executable file.

Another reason Golang is chosen to implement the backend is that it is strongly typed. This helps to keep types consistent, as there is no confusion about what types a function or API endpoint requires. Python and Typescript [68, 74] have syntax for types. This means specifying which types a function requires and returns is possible. However, neither is strongly typed, which means passing variables without declaring or specifying what type is possible.

To set up the API endpoints, the Gin web framework for Golang was used. Gin makes it relatively easy to nest endpoints in groups, as depicted in figure 5.5. Here there are three main groups, "app", "web", and "utils", that all have nested endpoints. Each nested endpoint adds to the URL it is nested within. This means the endpoint "/record/:uuid" adds to the URL "/app", resulting in the endpoint being "/app/record/:uuid". Each endpoint also has an appropriate HTTP method, which means it will only accept a request if both the URL and method is correct. This removes the need to manually check the method used when implementing the endpoint handler[1].

1. A "handler" is a function responsible for handling the request. Handling a request can include reading the body, doing some work, and responding with the appropriate response and/or data.

```go
func setEndpoints(r *gin.Engine, users *map[string]UserDataLocal) {
    appGroup := r.Group("/app")
    {
        appGroup.POST("/record/:uuid", record(users))
        appGroup.GET("/tag/:uuid", tag(users))
    }
    webGroup := r.Group("/web")
    {
        // Uploading video or metadata
        webGroup.POST("/upload-video/:uuid", recieveVideo(users))
        webGroup.POST("/metadata/:uuid", recieveMetadata(users))
    }
    utilGroup := r.Group("/utils")
    {
        utilGroup.GET("/get-uuid", createUuid())
        utilGroup.GET("/get-user-metadata/:uuid", getUserMetadata(users))
        utilGroup.GET("/remove-user/:uuid", removeUser(users))
        utilGroup.GET("/remove-user-data/:uuid", removeUserData())
        utilGroup.GET("/check-user-files/:uuid", checkUserFiles())
        utilGroup.GET("/begin-processing/:uuid/:numfiles",
↪   beginProccessing(users))
    }
}
```

**Figure 5.5:** Setup backend endpoints using Gin for Golang.

Figure 5.5 depicts the endpoints used by the app and web interface to communicate with the backend. To communicate with the position detection component, the backend uses a different API that a server on the position detection component is hosting. These endpoints let the backend start the video concatenation or a position detection process.

Concatenate video is done whenever the user uploads multiple videos from the frontend to the backend. The backend receives the video and stores it on a storage volume that is shared between the backend and the position detection component. This component (4) is illustrated in figure 5.1. Storing the videos on a shared volume (4) reduces network traffic since the video does not need to be re-transmitted from the backend to be worked on by the position detection component. Once the video is concatenated, the backend requests that the new video is used for position detection.

The backend is the “orchestrator” of Mearka, it controls the input from the frontend, as well as the response from the position detection component. These input requests can create metadata or add to existing soccer metadata connected to a specific UUID.

UUID is created on the backend using Golangs UUID package [75]. This generates a UUID as a 16-byte array, which is converted to a string, illustrated in figure 5.7. It is converted to a string, so it is easier to work with by the frontend and the position detection component in this POC prototype of Mearka.

What soccer metadata is stored depends on the data available. Figure 5.8 illustrates that metadata is structured on the backend as a map (similar to dictionaries in Python [76, section 5.5]). In this example, the keys, "uuid0" and "uuid1", are UUID's on the form described in figure 5.7. All metadata is stored in memory on the backend to make it easy to retrieve and add new metadata to a UUID. Another option was to store it on disk, but this option will be further discussed in section 7.3.

To store the soccer metadata as a map, the data structure of which is illustrated in figure 5.8 and 5.12, on the backend was chosen because it is humanly readable and easily convertible to JSON format when exporting. JSON is often used by REST APIs as many programming languages widely support it [20]. This is important as the frontend of Mearka is implemented using React Typescript as well as React Native. The backend is implemented with Golang, and the position detection component uses Python. Even if the different components of Mearka uses different languages, they should all be able to handle JSON [22, 23, 24].

An example of how the positional data can be extracted from the soccer metadata to display players' positions at a time "t", is illustrated in figure 5.6. For each time "t=n" to "t=n+5", there are certain known positions, as illustrated by the soccer field illustrations, and the dots depict player positions at time "t". Time "t" is based on the "time" key, value stored in the positional object, which is illustrated in figure 5.11.

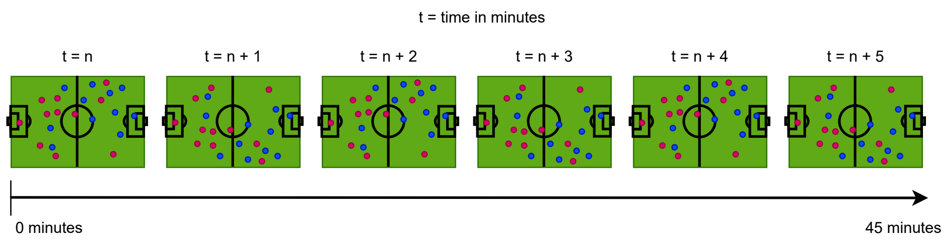


**Figure 5.6:** Example: soccer metadata used to know positions over time.

When the app wants to start recording a tag session, a new UUID is generated and sent in return to the app. In addition, an empty metadata object is created and mapped to the UUID in memory on the backend. When the user tags an event, a request is sent to the backend, and a new tag event is appended to the list of "tags" in the metadata, illustrated from the list starting on line 4 in figure 5.8.

**UUID:** b650161d-ca3d-4781-a9ac-5aa462d734d1

**Figure 5.7:** Example UUID

## 5.5 Position Detection Component

For the position detection component, everything is implemented using Python [68]. Python is chosen because of its simple, easy-to-read syntax and the vast number of machine learning libraries available. It is also ranked the number 1 programming language for machine learning by the website `www.codecademy.com` [77], for the same reasons.

Figure 5.9 lists the Python libraries used in this thesis to do position detection on the video provided by the user. In addition to these libraries, FFMPEG [25] is used to concatenate video through an operating system call. Some of the libraries listed will be described in section 5.5.1 and 5.5.2.

1. **OpenCV:** Used to load the video and read one frame at a time.
2. **Cvlib:** Used to detect objects, including people.
3. **Flask:** Used to implement server with a simple REST API for the backend.
4. **Os:** Used by Server to run scripts and commands on operating system.
5. **Pathlib:** Used to extract subpaths from a path or filename from a path.
6. **Argpars:** Used to setup parser for the scripts Command Line Interfaces (CLI)

**Figure 5.9:** List of Python libraries used.

Running the script that concatenates the video and starting the position detection program is done through a server. The server is set up using Flask [69], implementing a simple REST API that enables the backend to request concatenation or position detection. The position detection server endpoints are listed in figure 5.10. As a request comes in, the UUID is extracted from the URL to create a path to the shared folder between the backend and the position detection component. Once the correct path to use is created, the OS library runs a terminal command to start the appropriate script or program with the correct arguments. Once the script is done, the result is sent back to the backend. The new filename to the concatenated video is returned after video concatenation, and for position detection, the list of positional data is returned.

```
/merge-videos/<uuid>
/position-detection/<uuid>
```

**Figure 5.10:** Machine Learning endpoints.

### 5.5.1 Concatenate video

The videos uploaded to the system can be recorded on any **COTS** (section 3) camera, phone, action camera, or other devices. If the backend receives multiple videos, they must be concatenated for the position detection to be correct. This script takes a path to the folder containing the video files as a command line argument from the server. The OS library gets a list of every file in the path, which is later used when creating a list containing only videos. Filtering is done by iterating over the list returned by the OS library and adding every file with the ".mov" or ".mp4" extension to a new list. Pathlib library is used to extract which file extension each file has. The resulting list is then written to a file inside the path.

FFMPEG is called with the abovementioned files to concatenate the videos together. FFMPEG is started through its CLI to create a new video with the same extension (".mp4" or ".mov"), which contains the content of the videos sent by the user. When creating a new video using FFMPEG, there is an option to copy the codec from the input videos or re-encode it into a new codec. The decision fell on copying the codec as that is the most efficient way to get a concatenated video from the list of videos[2]. Once the video concatenation is complete, the new file name is returned back to the backend.

### 5.5.2 Position detection

This program is the component in Mearka that detects players' positions in the frame. This is partly accomplished with the help of OpenCV [35] and Cvlib [78]. It is partly accomplished because this thesis detects the pixel positions in the frame. Translating pixel coordinates to real-world field coordinates is left for future work. However, this thesis does have **Automatic position detection**

2. H.264 [17] is encoded by tracking the changes between frames. Only certain frames (i.e., every 10th frame) have all the information needed to display that frame. The "in-between" frames only store pixel values that have changed since the last "full-frame". Re-encoding means FFMPEG would need to, for each frame, go back and look at the last "full-frame" to get the full image, and then re-encode it to the new codec. This is more processor intensive than essentially copying the bytes directly from one video into another.

(section 3), even if the positions are pixel coordinates in the frame.

As the user uploads a video, the backend ensures that multiple videos are concatenated, if multiple are uploaded, before any position detection begins. This is done because the system assumes multiple videos are just multiple parts of a longer video.

Once the video is ready, the position detection server starts the position detection program. This program takes a video path as an argument and writes the resulting positional data to a file in the same folder that the video resides.

The script uses OpenCV to load the video like a “stream” so that it is possible to read one frame at a time. A frame is loaded into memory, and CVlib’s “detect_common_objects” method is used on the loaded frame. This method uses the pre-trained YOLOv4 model [14] to do object detection and utilizes the Central Processing Unit (CPU) to the max. It is possible to build the library to utilize the Graphical Processing Unit (GPU) instead, which would likely give a significant speed increase. The reason is that GPUs are more suited to handle highly parallel tasks such as these, compared to a CPU [79]. However, this is not done since this is a POC prototype, and implementing the functionality was prioritized higher than performance.

The object detection method returns three lists; one containing the labels of the objects it found, another list with the pixel positions of all objects with the different labels, and lastly, a list of how confident it was that the positions are of a given label. Labels are names of machine learning classes, like “person”, “monitor”, “apple”, etc. The lists have an equal number of elements, such that element on an index “i” in one corresponds to element on index “i” in the other two.

Mearka is only interested in finding the position of people relative to a soccer field, so those positions need to be filtered out. This is done by iterating over the list of labels and keeping track of which index in the label list the current label is. If the label is “person”, then the corresponding positions are added to a position object before that object is appended to a position list. Figure 5.11 illustrates a position object’s appearance. Here the time offset is mapped to the “time” key, and a list of pixel positions is mapped to the “positions” key. The time offset is calculated using equation 4.1. Once position detection is done, a list of position objects, illustrated by figure 5.12, is written to a JSON file in the same folder as the video.

```
        {
            "time":0.76,
            "positions":[
                [18,217,44,300],
                [400, 36, 222,56],
            ]
        }
```

**Figure 5.11:** Example positional object for one offset.

```
[
    {
        "time":0.76,
        "positions":[
            [18,217,44,300],
            [400, 36, 222,56],
        ]
    },
    {
        "time":3.12,
        "positions":[
            [44,100,300,543],
            [400, 36, 222,56],
        ]
    },
    ⋮
]
```

**Figure 5.12:** Example list of positional objects.

The server then reads the position file, returning its content to the backend. On the backend, the positional data is mapped as the value to the "positions" key of the metadata for the UUID, as depicted on line 25 in figure 5.8.

## 5.6 Summary

This chapter outlined the implementational details of Mearka. The system is comprised of three main components, the frontend, backend, and the position detection component. React Typescript with Chakra, and React Native, is used to develop the web interface and app, respectively. React Typescript was chosen for its simplicity to get components displayed on the screen with some pre-applied styling through Chakra. Developing the app through React Native and Expo was done to implement once and choose a platform to build the app for later.

The backend is developed with Golang and the Gin web framework. Golang is chosen because it has minimal syntax, easily concurrent code, and libraries like Gin to help ease the implementation of the REST API needed to connect everything.

The position detection component is developed using Python because of its high-level syntax that makes it humanly readable and its libraries that makes it easier to do machine learning tasks. Concatenating video is done using a combination of OS libraries in Python together with FFMPEG that runs on the machine. Position detection is done with the help of OpenCV and CVlib. It loads a video from the volume shared with the backend and runs one frame at a time through YOLOv4 to detect the pixel position of people in the frame. Translating the pixel coordinates to real-world field coordinates is left for future work. However, the "Pixel2Field" system by Peng, Liang, can be a good place to get inspiration [65]. The result is a file containing a list of positional objects (illustrated in figure 5.11 and 5.12) stored in the same folder as the video, on the shared volume.

The next chapter evaluates Mearka through experiments and results. Both functional and non-functional requirements will be evaluated.

```
[
    "uuid0":{
        "metadata": {
            "tags":[
                {"player":{},
                "event":{},
                "startCoordinates": {},
                "endCoordinates":{},
                "startTime":float64,
                "endTime":float64,
                "playingDirection":string
                },
                {"player":{},
                "event":{},
                "startCoordinates": {},
                "endCoordinates":{},
                "startTime":float64,
                "endTime":float64,
                "playingDirection":string
                },
                ⋮
            ],
            "players":[]
            "globalStartTime": time.Time,
            "positions":[
                {
                    "time": float64,
                    "positions":[][]int
                },
                {
                    "time": float64,
                    "positions":[][]int
                },
               ⋮
            ]
        }
    },
    "uuid1":{···},
    ⋮
]
```

**Figure 5.8:** Example JSON metadata.

# 6 Evaluation

This chapter outlines an evaluation of Mearka. Camera setup and experimentation are presented in section 6.1. Section 6.2 outlines the experiment setup and measurement of position detection speed and accuracy. Section 6.3 describes the experiment and results of testing JSON and XML as options to store metadata. The potential speedup Mearka has over manually tagging is measured and presented in section 6.4.

## 6.1 Choosing a Camera

When choosing a camera to help the development of Mearka, specific record time requirements were defined in section 4.2.1. A football match is at a minimum of $2x45$ minutes plus a 15-minute break, which adds up to 105 minutes of minimum record time. It would be possible to stop the recording and turn the camera off during the break, extending the battery life further. However, setting up a camera, press record just before the match start, and not think about the camera until the match is over require fewer resources by the team. Therefore, 105 minutes is the minimum record time required for the camera to meet the stated requirement.

The camera was set to record mp4 files to test the battery life, using H.264 compression and 16:9 aspect ratio. H.264 compresses video in the following way: At a set interval, there are frames called “keyframes”, where every pixel has the information needed to display that pixel. For every frame between these "keyframes", only the changed pixels contain their complete information. The pixels that have not changed refer back to the last "keyframe" instead of storing the information for that pixel, meaning the filesizes are lower than raw (where no information is discarded). However, it also means that the previous "keyframe" needs to be consulted to display every frame in the video that is not a "keyframe".

A combination of different camera and recording settings is tested in this thesis. Resolutions are denoted on the form Widt *x* Height in pixels. Framerate at each resolution is denoted as a number XX frames per second (FPS). Table 6.1 presents the different settings combinations tested.

| | | Resolutions | | |
|---|---|---|---|---|
| | FPS | 3840x2160 | 2688x1512 | 1920x1080 |
| Screen on while recording | 60 | ✓ | ✓ | ✓ |
| | 25 | ✓ | ✓ | ✓ |
| Screen turns off 3s after recording is started | 60 | ✓ | ✓ | ✓ |
| | 25 | ✓ | ✓ | ✓ |

**Table 6.1:** Settings combination for camera and record options.

### 6.1.1 Battery life

One requirement for the camera is that it has to be able to record for more than 105 minutes. Another requirement is that the camera should be easy-to-use. Therefore, the camera should be able to record for long enough (minimum 105 minutes) with various combinations of the settings used. It is not good enough if the camera can only record for long enough in one or two specific combinations of settings. Therefore, the camera has been evaluated to observe the recording limit for different combinations of settings.

The battery life was evaluated by recording continuously, starting with the battery at 100% and depleting it completely. The recording was conducted in a room-tempered room, filming with visible skies and a road in the lower part of the video, out of a window. One downside to benchmarking the camera by recording this scene is that there is less movement in the frame throughout the recording. This can be an issue for the data, as the cameras record in H.264 [17], which compresses each image based on the pixels that have changed values. Because there are more pixels that are similar between frames, the cameras might perform overly well since the camera needs to store less data per frame than a recording with more movement in the frame.

Each setting combination described in table 6.1 is benchmarked four times, and the average value is used to create the resulting graphs in figure 6.1, 6.2 and 6.3.

Using the average is chosen because it gives a good indication, rather than using an overly optimistic or pessimistic value. Benchmarking each setting combination four times was done to save time, compared to five. Only the average values would get used, compared to graphing the avg, 25th and 75th percentile, etc., as is done when evaluating the position detection component in section 6.2. A result of this is that the reported values are higher than the worst observations but also lower than the best-performing benchmarks.

The graphs have resolution and frame rates along the x-axis, denoted on the form "1080p25", for example. 1080 refers to the vertical pixel resolution of the video, and 25 refers to how many FPS the video is. The "p" refers to the scan mode of the video, which indicates how the frames are displayed on the screen. In this case "p" is for "progressive" rather than "interlaced". More information about the two different scan modes is available in appendix A.1.

Screens draw power and heat up when turned on for some time, negatively impacting the recording time. The first set of exeriments was conducted with the screen turned on throughout the recording to test the "worst-case scenario" for the battery. The result is displayed in figure 6.1. From this graph, it is possible to conclude that in almost every combination, except when recording with a resolution of 3840x2160 (UHD) at 60 FPS, the camera can record longer than the minimum of 105 minutes.

The same set of experiments is conducted with the screen turning off three seconds after the recording starts. Figure 6.2 displays the results of these benchmarks. The findings are that every resolution and FPS combination can record for more than 105 minutes with the screen turning off. This is backed up by every point being above the blue horizontal line, which illustrates the lower limit of 105 minutes of record time.

Figure 6.3 displays the differences when comparing the results from recording video with the screen turned on versus off after three seconds. Orange dots are the results from recording with the screen on at different resolutions and with the different framerates, and blue dots for recording with the screen off. The blue line illustrates the minimum requirement of 105 minutes minimum recording time. The plot indicates that the user can choose almost any combination of settings and still get the required 105 minutes of recording time. As depicted by the plot, only 3840x2160 resolution, recording 60 FPS and keeping the screen on during recording, does not meet the 105 minutes required recording time. These tests are only valid for considering recording times possible with the **DJI**

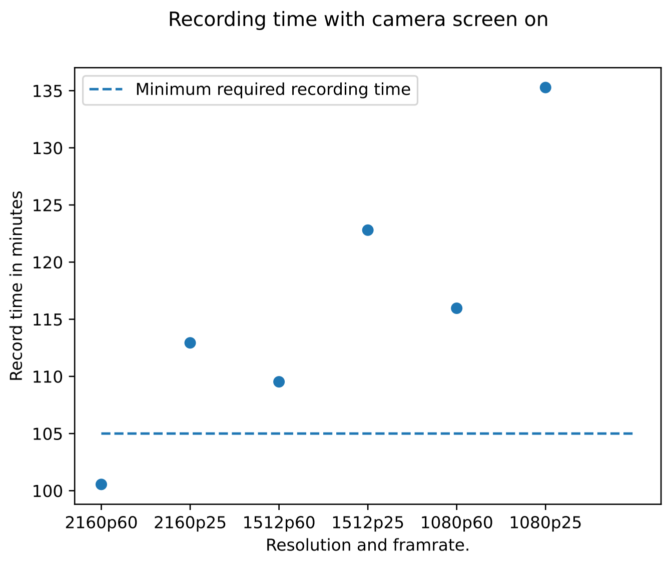


**Figure 6.1:** Recording times with screen on.

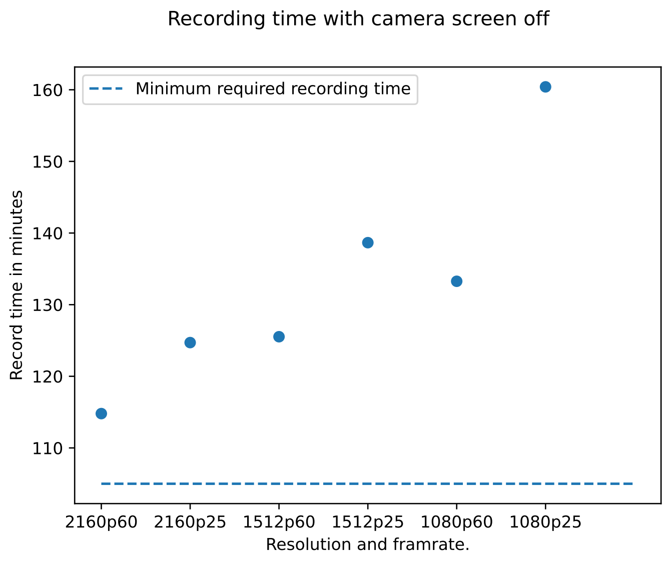


**Figure 6.2:** Recording times with screen off.

**Action 3.**

From the plot in figure 6.3, it is possible to conclude that recording with the screen off will increase the overall recording time of the camera. Decreasing the resolution is another contributing factor to increase recording time. Resolution is a measurement of how many pixels need to be stored for each frame in the video. Table 6.2 displays the resolution differences and the number of pixels

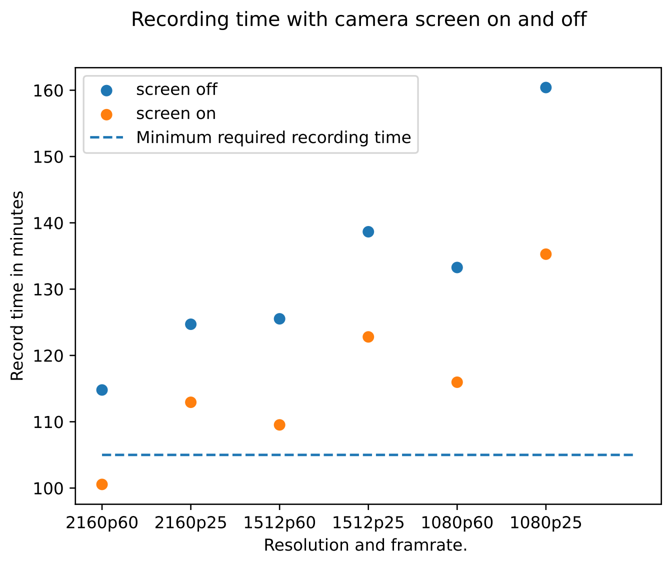


**Figure 6.3:** Compare recording times - blue=screen off, orange=screen on.

stored for each frame at each resolution. If only considering the vertical number of pixels noted on the x-axis in plot 6.3, it might give the impression that video with 1512 and 2160 vertical pixels is only 1.5 and 2 times higher resolution than 1080 pixels. However, this is not the case, as table 6.2 describes, the higher resolutions have closer to double and quadruple the number of pixels per frame. Each pixel contains data, so storing more pixels per frame means the camera must work harder to store everything. If the camera needs to work harder, it consumes more power, which lowers the overall battery life and recording time.

| Resolution | 1920x1080 | 2688x1512 | 3840x2160 |
|---|---|---|---|
| Number of pixels | 2073600 | 4064256 | 8294400 |
| Multiple of previous resolution | 1 | 1.96 | 2.0408 |
| Multiple of lowest resolution | 1 | 1.96 | 4 |

**Table 6.2:** Resolution pixel differences.

In addition to the screen and resolution influencing the power consumption and, thereby, the recording time, the number of frames per second recorded affects this as well. A good illustration of this is comparing “1512p25” to “1080p60” in figure 6.3. As explained, higher resolution means more pixels to store per video frame, which decreases battery life and record time. In this example, the lower resolution with a higher framerate has a shorter recording than the higher resolution with a lower framerate. The reason for this is illustrated by table 6.3. Video with a resolution of 1920x1080 pixels and 60 FPS has to store 124,416 million pixels every second. That is 22, 8 more than the video with twice the resolution at 2688x1512 pixels but half the framerate with 25 FPS. Overall, this results in the camera doing less work per second and can therefore record for longer with a higher resolution but lower framerate.

| Resolution | FPS | Pixels per second | Difference |
|---|---|---|---|
| 1920x1080 | 60 | 124416000 | 22809600 |
| 2688x1512 | 25 | 101606400 | |

**Table 6.3:** Example: compare total pixels/second for 1512p25 and 1080p60.

### 6.1.2 File size

As described in section 6.1.1, higher resolution means more pixels per frame in the video. This, in combination with the framerate and the length of the recording, determines how large the video file will be. File sizes are evaluated to indicate the potential storage requirements of recording in various resolutions and framerates.

The data used to test file sizes is the same as what section 6.1.1 uses. The number of minutes recorded and file size were noted for each battery life test. These numbers are averaged out within each combination of settings and used to plot figure 6.4 and 6.5. Only file sizes from the evaluation with the screen turned off are used for figure 6.4. Only using data from when the screen turned off was chosen because these recordings lasted longer and had bigger file sizes. This can better indicate the actual storage requirements for different settings-combinations, compared to underestimating based on shorter recording times.

From figure 6.4, it is possible to extract that the higher the resolution and framerate, the higher the file size. This is not wrong, as illustrated by the linear regression over the filesizes resulting after recording trending downwards towards the lowest resolution. However, as table 6.3 presents, and as underlined by figure 6.4, it is possible to record at a higher resolution while still getting smaller filesizes than a lower resolution-framerate combo.

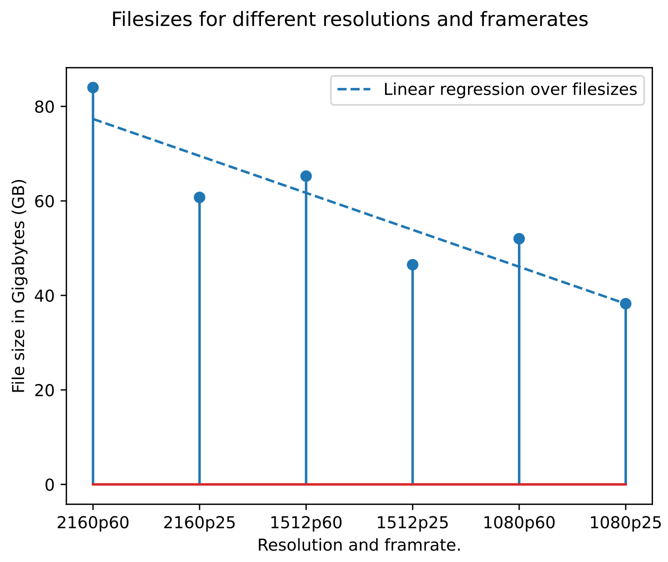


**Figure 6.4:** File sizes between resolutions and framerates.

To explain how a higher resolution video can record for longer and have smaller file sizes than a lower resolution one, only based on different framerates, equation 6.1 can be used. It is established that pixels equal data, so more data equals larger files. From equation 6.1, it is possible to calculate that "2160p25"and "1512p25" needs to store 15% and 18.4% fewer pixels every second than the 60 FPS version of the next step down in resolution.

$$\text{Pixels per second} = \text{Width-pixels} \cdot \text{Height-pixels} \cdot \text{FPS} \tag{6.1}$$

To further figure out which resolution and framerate combination gives the most record time per GB, equation 6.2 is used to create figure 6.5. From this figure, "1080p25" gives the longest recording per GB, but the ratio does not decline linearly for the higher resolutions. As noted, recording in "1512p25" rather than "1080p60" results in smaller files and a more extended recording. This is supported by figure 6.5 displaying that "1512p25" has a higher recording time per GB ratio than "1080p60". It is also possible to note the same discrepancy between "2160p25" and "1512p60", but the difference is less than the beforementioned example.

$$\text{Ratio} = \frac{\text{Recording time in minutes}}{\text{File size in gigabytes}} \tag{6.2}$$

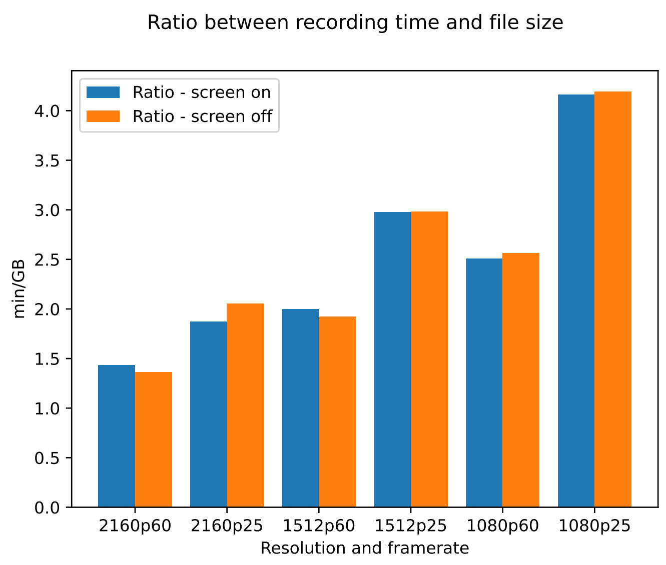


**Figure 6.5:** Recording time to file size ratio.

If it is important to have the longest possible record time, then “1080p25” is the best option. However, “1512p25” might also be a good choice. It doubles the resolution while still providing more than adequate battery life and has the second-best ratio between record time and file size. Which option is best for position detection is explored in the next section.

## 6.2 Position detection

Mearka is designed to make position detection as easy as possible. This is why the web interface described in section 4.4.1 enables the user to upload a video, or multiple video segments of the same video, to the system for position detection without additional manual input. How Mearka accomplishes to do position detection is described in section 4.6.2 and 5.5.2. But, the system must have completed detection with the specified requirement of **soft real-time**, which is defined as within 12h for a full match. This section investigates if Mearka is able to meet the system requirement defined in chapter 3.

### 6.2.1 Test system

All the experiments are run on localhost, as the system is a POC and not deployed on a remote server. Specific experiments, like upload speed, latency tests, etc., are not conducted because the system is not deployed.

Localhost runs on an HP EliteDesk 800 G6 Desktop Mini PC with the following specifications:

- **OS:** Ubuntu 22.04.2 LTS x86_64
- **Kernel:** 5.19.0-42-generic
- **CPU:** Intel i7-10700 (16) @ 4.800GHz
- **GPU:** Intel CometLake-S GT2 [UHD Graphics 630]
- **Memory (RAM):** 15776MiB (16GB)

### 6.2.2 Resolution speed

The first experiment evaluates how resolution impacts position detection time. The experiment is based on a match recorded in 3840x2160 resolution at 25 FPS. A thirty-second section from this video was extracted, as well as scaled down to separate 2688x1512 and 1920x1080 resolution files. The choice to use a thirty-second clip is made to have time to run each test multiple times within a reasonable time frame. The resolutions that the original video is sized down to are chosen to mimic actual recording options in the camera of choice.

To prepare Mearka for the experiments, the system was started, and a video was uploaded to detect positions. The reason is that Mearka needs to download several libraries and models, like YOLOv4 [14], for the first video being uploaded after a restart.

Each resolution was uploaded five times, and the system was cleared (but not restarted) to remove old data. Uploading and doing position detection on each resolution five times was done to get more representative data and an idea of the variance. The result of this experiment is displayed in figure 6.6, and more detailed numbers are listed in table 6.4.

The boxplot in figure 6.6 illustrates where the median, mean, and 25th and 75th percentile are. The orange line in each box notes the median, the green line is the mean for each resolution, and the lower and upper bounds of the box are the 25th and 75th percentile, respectively. Table 6.4 describes the mean time for position detection as well as the average FPS, in more detail.

As depicted in figure 6.6, the mean and average timings of both “1512p25” and

"1080p25" are not that different, even if "1080p25" in general is a little bit faster. However, "2160p25" needs a bit more time to complete position detection on the same number of frames. One explanation for why the two lower resolutions use roughly the same time is that modern computers are efficient. CVlib [78] and YOLOv4 [14] use all available computing power on the machine. Since "1080p25" and "1512p25" are "just" doubling in resolution, the resolutions might be similar enough so that the computer can complete detection almost equally as fast by utilizing more resources. However, the next step up is a quadrupling of resolution compared to "1080p25", and this resolution generally takes more time, as depicted in figure 6.6. One reason could be that the computer has exhausted its resources and cannot work faster. It is observed that the computer used between 80-100% of its CPU, as well as up to 14.7GB of RAM, which indicates the computer is maxing out the resources.

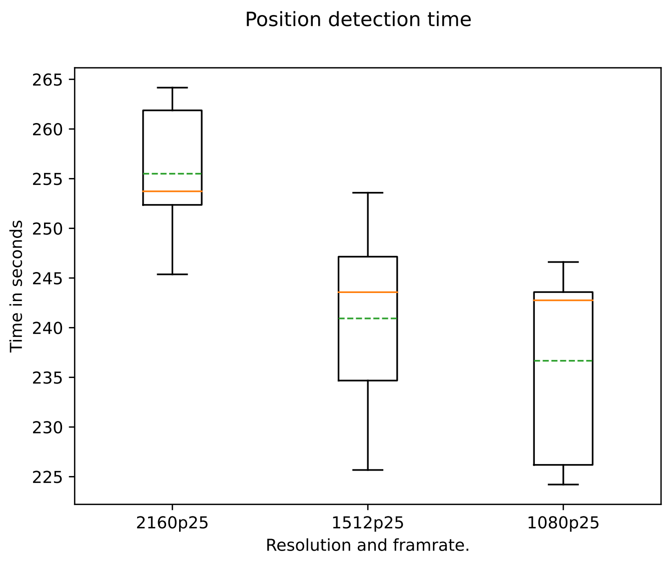


**Figure 6.6:** Position detection time on 30s video

Table 6.4 describes concrete numbers for the average time to detect the position in a 30s video, as well as an estimation for how long it would take to detect positions in a 90 min video. Referring to figure 6.6, table 6.4 concretizes the differences in detection time between the three resolutions. The two lower resolutions differ by ≈4 seconds on a 30-second clip, while the highest resolution used ≈14 seconds more than the next step down in resolution.

Calculating how many frames the system can detect per second is done with the help of equation 6.3. This equation needs the framerate of the video, the length of the video, as well as the time it took to detect the positions in that clip. The result from using this equation is described in the "avg. detected FPS" column in table 6.4. Each resolution has an average detect FPS around 3, which also builds on the theory that the CPU of the computer is the bottleneck. As mentioned in section 4.6.2, position detection with CVlib and YOLOv4 can be done with the help of GPU. This would likely speed up the number of frames Mearka can detect each second, significantly [79]. Since Mearka is a POC, and due to time and hardware constraints, this option was not explored any further in this thesis but left for future work.

Another possible way to decrease the time it takes to run detection is to filter out pixels that are not part of the soccer field. Since the cameras should be static during recording (but movable otherwise), segmenting out parts of the image that does not contain the soccer field is only needed to be detected on one frame to create a filter [80]. This would reduce the number of pixels needed to detect over, which could help the system detect more frames per second.

$$\text{Avg. detect FPS} = \frac{\text{Video FPS} \cdot \text{Record time in seconds}}{\text{Avg. total time in seconds}} \tag{6.3}$$

| Resolution | Avg. Total time (in seconds) | Avg. detected FPS | Est. time (in minutes) for 90min video |
|---|---|---|---|
| 3840x2160 | 255.499 | 2.935 | 766,609 (12,776 hours) |
| 2688x1512 | 240.930 | 3,112 | 723,007 (12,050 hours) |
| 1920x1080 | 236.669 | 3,168 | 710,227 (11,837 hours) |

**Table 6.4:** Position detection of 30s video

After calculating the average FPS that the system can detect, it is possible to calculate how long it would take to run detection on a full football match. To give the system the best chance to detect a full match within 12 hours, a football match is set to be 90 minutes of video. The results of using equation 6.4 are listed in table 6.4. While the differences for detecting positions in a 30-second clip were ≈4 and ≈15 seconds, that difference is now ≈13 and 43 minutes when estimating detection time for an entire football match. The estimated detection times are 11.8, 12.0 and 12.7 hours, respectively.

$$\text{Est. detect time (in seconds) for XX min video} = \frac{(\text{video FPS} \cdot 60) \cdot \text{XX min}}{\text{avg. detect FPS}} \tag{6.4}$$

These numbers illustrate that Mearka is only able to meet the requirement of *soft-realtime*, with recordings in the resolution “1920x1080” at 25FPS. *soft-realtime* is defined to be 12 hours after the game is complete and the video is uploaded. The next step up in resolution misses the mark by ≈ 3 minutes, rather than 46 minutes for“3840x2160”, both resolutions at 25 FPS.

### 6.2.3 Framerate Speed

Exploring how different framerates affect the time it takes to do position detection is done by recording a one-minute video in 2688x1512 resolution, both in 60 and 25FPS. The clips are trimmed to exactly one minute while keeping the other recorded attributes the same. Mearka is started, and each video is uploaded three times to get the average timing for each framerate.

Table 6.5 displays the difference in how long it takes to run position detection on 60FPS versus 25FPS[1]. The time difference between the two framerates to detect positions are around 10 minutes. This means that, for every minute of video, Mearka needs roughly 10 minutes longer to detect positions when recording at 60FPS instead of 25FPS. Calculating an estimation of how long it would take to process an entire match (90 minutes of video), table 6.5 illustrates that the 60 FPS video would take around 2.3 times as long at nearly 27 hours.

As table 6.5 illustrates, there is a time penalty to recording video in 60FPS when detecting positions with Mearka. Implementing support for GPU can help speed up the system's processing capabilities [79]. However, the current implementation of Mearka runs on the CPU, and it is at the limit of what the hardware can do, maxing out at processing roughly ≈3 frames every second. Therefore, to get the most value out of Mearka within a reasonable timeframe, the proposed best framerate is to record at 25FPS. This enables the system to meet the *soft-realtime* requirement, set in chapter 3, defined as delivering positional data within 12 hours.

| Video length | Video FPS | Total time in seconds/minutes | Avg. detect FPS | Est. time 90min match (minutes/hours) |
|---|---|---|---|---|
| 1 minute | 60 | 1073.09/17,88 | 3,35 | 1611,94/26,87 |
| 1 minute | 25 | 462.35/7,71 | 3,24 | 694,44/11,57 |

**Table 6.5:** Time difference between 25 and 60 FPS

1. Support for GPU will drastically speed up the number of frames Mearka is able to detect each second [79]. However, the current implementation uses the CPU for position detection and therefore needs to consider a max detection speed of around ≈3 FPS.

### 6.2.4 Detection Accuracy

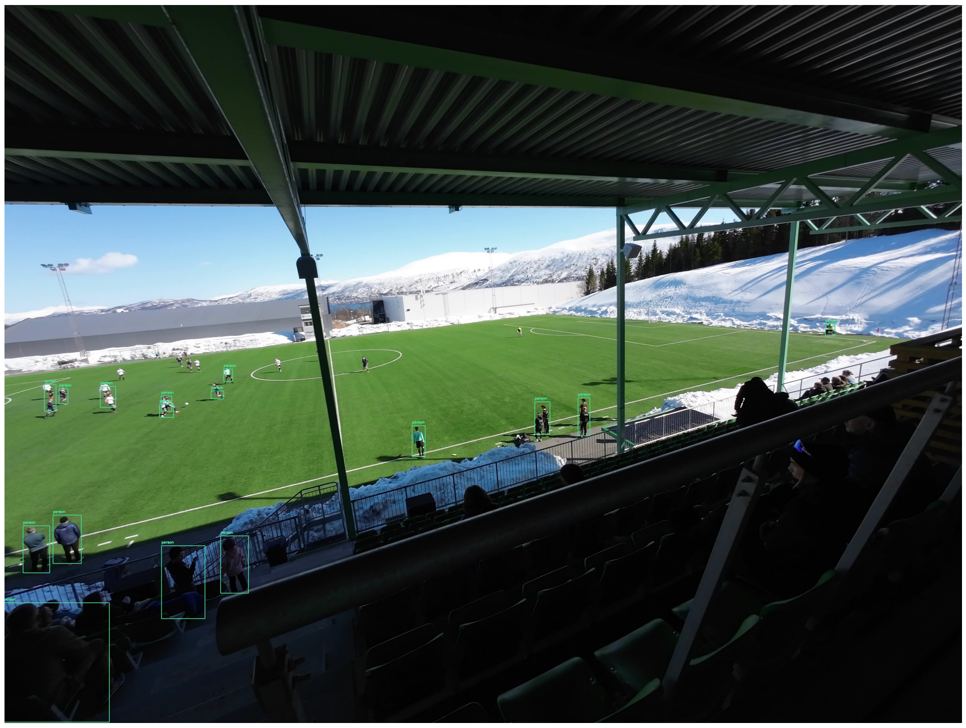
The accuracy of Mearka is evaluated to observe how well the position detection is, to inform about the quality of data that it is able to provide. To evaluate the accuracy, three resolutions of the same video are uploaded and verified. These resolutions are "4096x3072", "2688x2016" and "1920x1440", which are all "4:3" aspect ratios[2]. For each video, Mearka exports a frame every 20th frame, containing a bounding box around every detected person. Six exported frames from each resolution are copied from Mearka to be verified manually. The positions detected are pixel coordinations in the frame rather than actual field positions. This translation is not developed in this thesis but is left for future work.

Verifying the frames is done by counting every player the system detects and noting any player it does not. For a detection to be valid, the bounding box must clearly show that it has detected an entire player. A bounding box with only a player's shoes inside will not be counted as a success. People that are detected but that are not players are not counted towards the system's accuracy. If Mearka detects something as a person that is not a person, that counts towards the misclassification statistics.

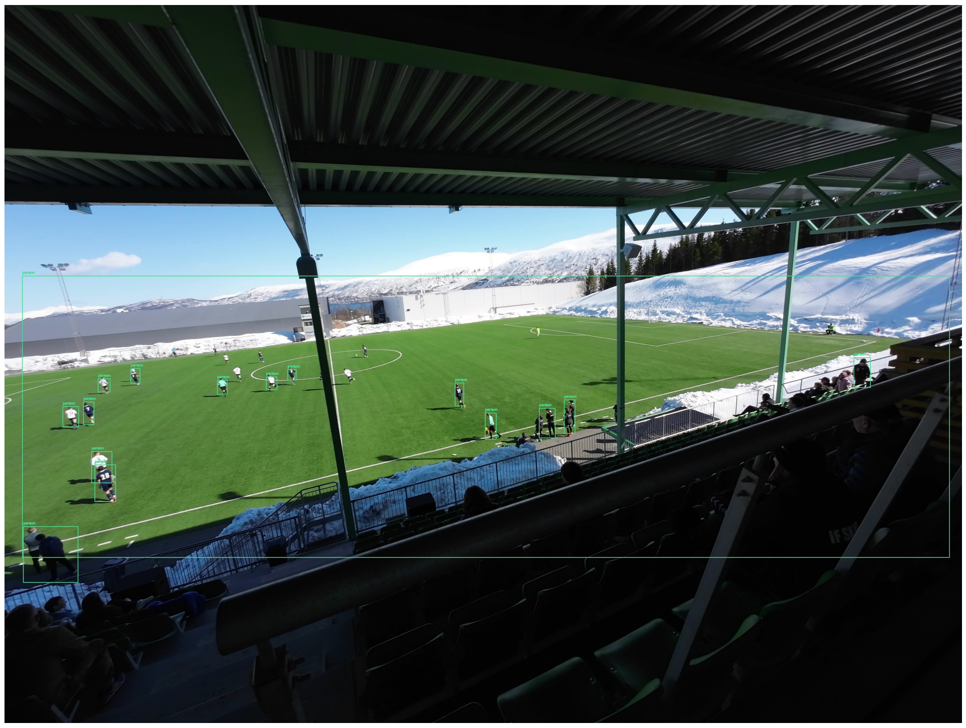
Figure 6.7 displays two frames with bounding boxes after position detection. Subfigure 6.7b illustrates that Mearka sometimes detects people where there are none.

Ideally, every player should have a box illustrating where they are in the frame. However, as figure 6.8 displays, the accuracy of Mearka rarely exceeds 50%. The percentages are calculated with equation 6.5. The numbers used to create this figure are collected from the same video as the frames in figure 6.7. One possible reason why the detection percentage is so low is that the camera was mounted in the stands with multiple columns and stances in the frame (depicted in figure 6.7). The camera is also far away from the field and very wide-angled, resulting in people appearing smaller in the frame, which makes it more difficult to detect them. It is noted that Mearka is better at detecting people closer to the camera than further away. This can be explained by people closer to the camera taking up more of the frame, which equals more pixels that can then be detected.

2. These resolutions and aspect ratios were chosen because they could record more of the environment than their "16:9" aspect ratio counterparts.



**(a)** Accuracy with bounding boxes



**(b)** Accuracy fail

**Figure 6.7:** Accuracy example

Figure 6.8 illustrates that the resolution has less impact on the detection accuracy. And, as backed up by the numbers in table 6.6, the median and average of “1080p25” are higher than the higher-resolution options from the same angle. It also has less variance as the 25th and 75th percentile are closer together, with between 40% and 48% accuracy. The highest 75th percentile, between the three resolutions from the same angle, goes to the highest resolution, a few percentage points higher than the lower resolutions.

$$\text{Percentage} = \frac{\text{Number of detected people}}{\text{Number of people}} \tag{6.5}$$

Although the resolution does not have the highest impact on the accuracy of detecting people, the placement of the camera does. Figure 6.9 depicts an alternative camera angle with fewer columns and stands in the frame. The same test is run on a one-minute video from this angle, exporting a frame with bounding boxes every 20th frame. From these frames, the same interval of frames was extracted from the system for manual inspection. The data from this angle is placed next to, but separated from the other data, as this is a different angle with only one resolution. As observed by table 6.6, the alternative camera angle gives the highest median and average percentage of 49.24 and 50 %. The 25th percentile is the second highest of all four measurements, and the 75th percentile is 2.69 percent points above the next one down.

| Resolution | Mean | Median | 25th percentile | 75th percentile |
|---|---|---|---|---|
| 3840x2160 | 41.85 | 40.87 | 31.78 | 50.71 |
| 2688x1512 | 40.89 | 39.74 | 30.39 | 46.83 |
| 1920x1080 | 42.80 | 42.02 | 40.29 | 48.21 |
| Alternative camera angle | | | | |
| 2688x2016 | 49.24 | 50.0 | 39.77 | 53.40 |

**Table 6.6:** Resolution accuracy

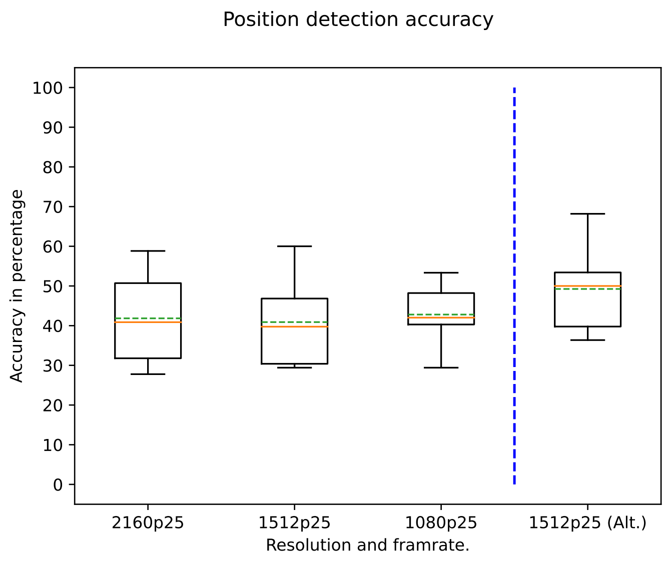


**Figure 6.8:** Position detection accuracy

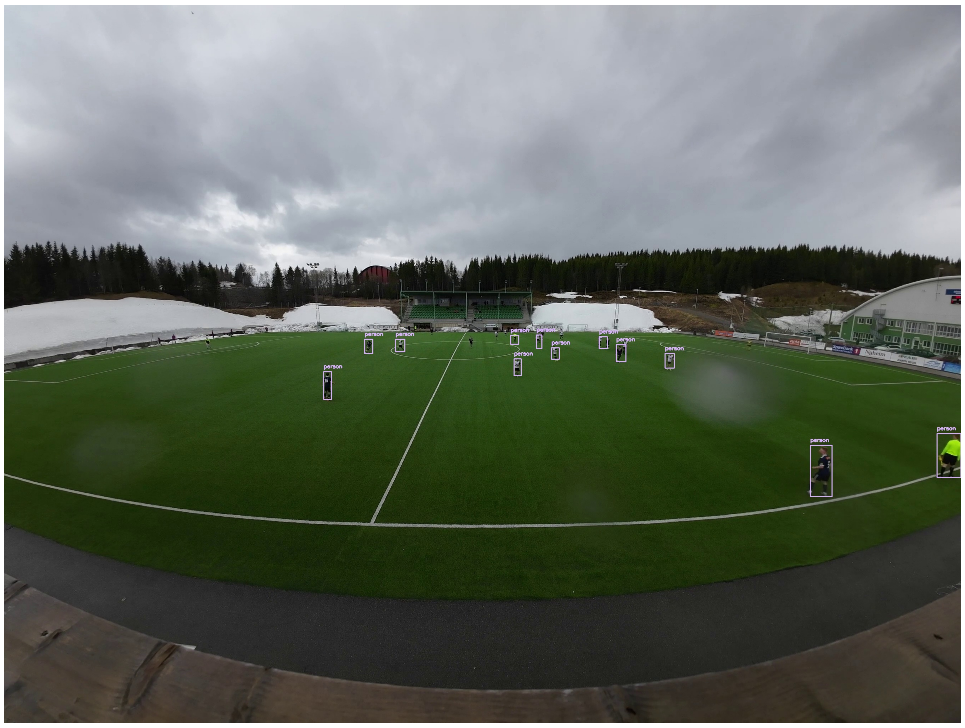


**Figure 6.9:** Alternative camera angle

As described, Mearka is not perfect, and it does not detect 100% of the players on the field. There are multiple reasons why this might be. The camera can have physical obstacles between the lens and the field, like the columns in figure 6.7, or water droplets as in figure 6.9. In addition, players can be overlapping in the frame or be so far away that there are not enough pixels to detect that it is a person. But, Mearka, and by extension YOLOv4 [14], sometimes detects a person where there are none. This is illustrated in figure 6.7b, where a bounding box takes up half the frame. Based on how YOLOv4 notes the position, the bounding box should frame a person with little to no space around it, which illustrates that this is a misclassification.

Figure 6.10 describes the number of miss classifications observed from the frames verified. That is the total number of misclassifications for each resolution's validated frames. As depicted, the number of misclassifications is almost the same, except for the highest resolution, with one less misclassification than the rest.

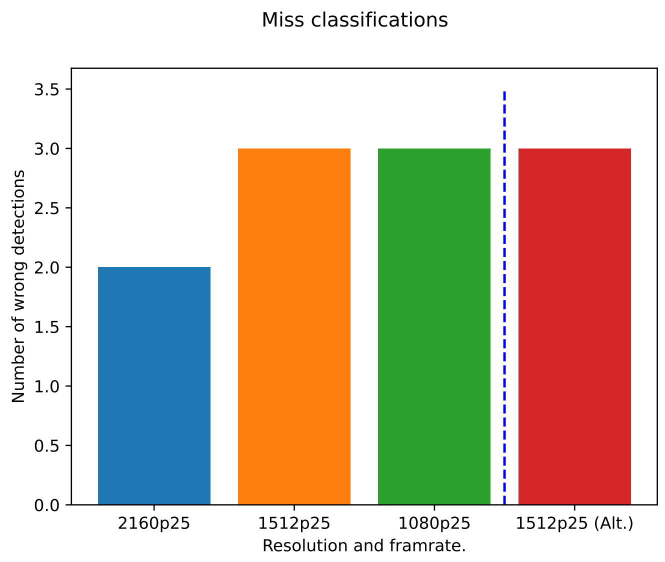


**Figure 6.10:** Number of miss classification within six frames

The data for accuracy suggests that the lower-resolution options are just as accurate, if not more than the higher-resolution options. However, the most significant increase in accuracy comes from choosing a camera angle well suited for position detection. This means an angle with as few visual obstructions as possible, where the players have as much contrast to the background, and take up as much of the frame as possible.

## 6.3 Metadata Size

The data generated by Mearka should be humanly readable and take up as little storage space as possible. Since JSON is very human readable, that is one of the reasons it was the preferred format to store the metadata. In addition, JSON is widely supported by the technologies used in this implementation. However, JSON was not the only option when deciding which format to store metadata. Another alternative to JSON is the “Extensible Markup Language” (XML) [81].

XML is a markup language based on tags, similar to HTML, that differentiate items in the file and enable items to be nested within other objects. Figure 6.11 illustrates how the metadata could look using the XML format instead of JSON. As illustrated, each item needs an opening tag like “<tags>” with a corresponding closing tag “</tags>”. Each item in between also needs an opening and closing tag, as illustrated on line 8 in figure 6.11, “<startTime>3.777054983</startTime>”.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <tags>
        <player/>
        <event/>
        <startCoordinate/>
        <endCoordinate/>
        <startTime>3.777054983</startTime>
    </tags>
    <tags>
        <player/>
        <event/>
        <startCoordinate/>
        <endCoordinate/>
        <startTime>4.992716238</startTime>
    </tags>
    <tags>
        <player/>
        <event/>
        <startCoordinate/>
        <endCoordinate/>
        <startTime>5.24555306</startTime>
    </tags>
    <tags>
        <player/>
        <event/>
        <startCoordinate/>
        <endCoordinate/>
        <startTime>7.215931706</startTime>
    </tags>
    <globalStartTime>2023-05-26T08:30:36.64658778Z</globalStartTime>
    <positions>
        <time>0</time>
        <positions>
            <row>2112</row>
            <row>1107</row>
            <row>2158</row>
            <row>1210</row>
        </positions>
        <positions>
            <row>1807</row>
            <row>1070</row>
            <row>1855</row>
            <row>1156</row>
        </positions>
        <positions>
            <row>1584</row>
            <row>944</row>
            <row>1608</row>
            <row>1012</row>
        </positions>
    </positions>
</root>
```

**Figure 6.11:** Example XML Metadata

Having opening and closing tags for each element in the metadata amounts to more characters needed to be stored to convey the same information, compared to JSON, which is illustrated in 5.8. This observation is also quantified by table 6.7. The table illustrates the file size difference between JSON and XML of metadata created from the same 30 and 60-second video clips. These observations illustrate that the XML takes up six times as much space as the JSON equivalent metadata. Based on these findings, in combination with JSON being widely supported by both Python [22], Golang [24], and React Native [23], this was the clear choice for Mearka.

| Video length | FPS | JSON size | XML size |
|---|---|---|---|
| 1 minute | 25 | 0.403 MB | 2.3 MB |
| 30 seconds | 25 | 0.203 MB | 1.2 MB |

**Table 6.7:** Size difference between JSON and XML

## 6.4 Speedup

To test the speedup that Mearka provides when generating metadata, it is compared to 100% manual tagging of the positional data in the video. As it is previously stated and illustrated that Mearka can detect $\approx$ three frames a second; this is how many frames were tagged manually to compare. It should be noted that the three frames, tagged manually, are the first three frames in the same video that are uploaded to Mearka.

The frames from the video are tagged as similar to how Mearka tags positions as possible. For each person detected in the frame, note the pixel coordinate of a potential bounding box's top left and bottom right corners.

As table 6.8 illustrates, Mearka can detect more frames per second than doing it 100% manually. The difference is substantial, as Mearka is over three orders of magnitude faster than tagging the positions manually. Table 6.8 displays the average frames per second for each tagging method. Calculating the speedup is done with equation 6.6, which finds $x$ speedup compared to $y$. The average detected frames per second numbers were used to calculate the speedup between Mearka and manual tagging. The result concludes that Mearka gives a **1538** times speedup compared to manually tagging the positions.

$$\text{Speedup}(x, y) = \frac{x}{y} \tag{6.6}$$

| Video length in frames | Video FPS | Manual or Mearka | Total time in seconds/minutes | Avg. detect FPS | Est. time 90min match minutes/ hours |
|---|---|---|---|---|---|
| 3 | 25 | Manual | 1442.28/24.038 | 0,0021 | 1071428,57/ 17857,14 (2,04 years) |
| 50 | 25 | Mearka | 15.47/0,26 | 3,23 | 696,59/ 11,60 |

**Table 6.8:** Time difference between Mearka and manual tagging posisions

## 6.5 Summary

This chapter provided an evaluation of Mearka based on different experimental results. The experiments were organized into camera benchmarks, position detection speed, and accuracy, Metadata size as well as system speedup compared to manual tagging.

For the camera tests, the result shows that only the highest resolution and framerate, with the camera screen turned on during recording, is not able to reach the 105 minutes of minimum record time. Apart from this, the camera reaches 105 minutes of record time in every other settings combination.

Position detection is able to finish with the *soft-realtime* requirement of 12h with a video in the lowest resolution, recorded at 25FPS. Recording in 60FPS scales the detection time linearly compared to 25FPS, meaning roughly 2,4 times longer. Although the position detection accuracy rarely exceeded 50% accuracy, the result illustrated that resolution has less to do with the accuracy than the camera angle and position. Camera placement can increase the mean accuracy between 15-20% compared to increasing the resolution. The median accuracy is improved by 18-25%, with better camera placement.

To store metadata, the results describe that JSON takes up six times less space than the XML equivalent. Lastly, experimenting with speedup indicates a 1538 times faster turnaround using Mearka rather than 100% manually tagging.

The next chapter elaborates on a discussion of Mearka.

# 7 Discussion

This chapter outlines a discussion on Mearka. Section 7.1 discusses the Mearka-app and the design choices made. Section 7.2 Outlines the strengths, weaknesses, and possible improvements of the Mearka web-interface. The backend's system design, implementation, and discussion are outlined in section 7.3. Section 7.4 discusses the position detection component and possible improvements to position detection.

## 7.1 Mearka-app

The Mearka-app has four requirements it needs to meet. It needs to **Tag events**, be intuitive and easy to use (**Ease-of-use**), **Non-intrusive**, and in addition, give a lot of value with minimal effort (**Low effort - high reward**).

On startup, the Mearka-app allows to start a new recording. A new recording means a new session where the user can tag events, with the tags storing the offset from when that recording started. During a recording it is possible to tag an event or stop the recording, as illustrated by figure 4.3. This UI only allows do two things, and so the UI should be intuitive, fulfilling the **Ease-of-use** requirement. Since the Mearka-app can tag events as well, by sending an API request to the backend, it is able to meet the requirement to **Tag events**.

As illustrated by figure 4.3b, once a recording is underway, the tag button takes up most of the screen, allowing the user to tag events almost without needing to look down at all. This design philosophy is inspired by the app used in Muithu [40], as it greatly decreases how intrusive the app is while a game is ongoing. Because of this, the Mearka-app is designed to meet the requirement of being **Non-intrusive**, as well as having a **Rugged UI** that could work in harsher environments.

Additionally, even though the Mearka-app itself starts and stops recording by contacting the backend and sends a request to tag events, it does not store anything locally. Tagged soccer metadata is returned as a JSON file from the backend as soon as a recording is stopped. This enables the Mearka-app to be hopefully **Low effort - high reward**, meeting that requirement.

Once a recording is complete and the backend responds with the recorded soccer metadata, it is possible to share that soccer metadata in different ways. This means sending it to an email address, storing it on the phone, or any other options provided by a sharing screen on Android or IOS phone operating systems. However, the Mearka-app does not store anything locally in memory or otherwise if the user does not share it from the Mearka-app once that screen appears. As a result, if the user closes out of the sharing popup without storing or sharing the data somewhere, it is lost forever. The reason it is lost forever is that the backend wipes the user data once a recording is complete, and it has responded with the soccer metadata,

The Mearka-app was designed to be simple and a POC to enhance the usability of Mearka, which is why it is a UI to interact with the system rather than processing locally. A welcome addition to the Mearka-app would be the ability to store a selection of historical recordings locally Then it would be possible to retrieve the soccer metadata, even after the Mearka-app has been restarted, or the sharing screen has been clicked away.

Contrary to the web interface, the Mearka-app does not need to request a UUID from the backend when starting a recording. The Mearka-app has separate endpoints at the backend, and the backend assumes a new recording means a new match or training session. Therefore, it generates a new UUID automatically as a response to a recording start.

A possible option would be to keep the UUID and let the user choose when to clear it, as long as the Mearka-app has not been restarted. This would allow the user to append tags if they stopped the recording prematurely (as an example), as starting a new recording (without app restart) could let the user tag an event with an offset based on the first recording start.

An example could be a training session where they start the video and the

Mearka-app recording simultaneously. The first training set is over, but the camera is still recording, and then a player wants the coach to look at some drills. If the soccer metadata file starts at the same time as the video, then a new recording start would append new tags with an offset into the video that is recording already. Unlike today, where a new recording would mean the tags are offset from the start of that specific session. This option would mean that the tags would line up in the video recording after the entire session is over and the video recording is stopped.

The Mearka-app is designed to be simple and to be easily expandable if future implementations require or desire it. An example would be to have a few different tagging options when tagging, of different severity or types. This could be "Offence", "Defence", "Tackle", or as simple as "Tag", and "Important tag". But as Mearka is currently a POC prototype, the focus has been on creating the option to tag through an Mearka-app at all, which is the reason behind its simplistic design.

## 7.2 Mearka Web-interface

For the Mearka web-interface to be helpful, it has to meet two requirements: to be **Easy-to-use** and **Automatic**. Some solutions, like Hudl [5], provide video recording systems where the customer can tag events manually or pay them to tag the videos for the team. This is either labor-intensive or an expensive option for the team.

The web-interface of Mearka allows to upload a video to be used for position detection, as well as export the resulting soccer metadata once it is done. On the UI, there are two upload buttons as well as a button to export the soccer metadata from the backend. The "Upload video" button opens a popup where one can choose to upload a video to the backend to be processed. The other upload option is for uploading a JSON file containing soccer metadata, i.ex. Resulting from the Mearka-app. If both video and soccer metadata are uploaded, the soccer metadata from the upload, and the positional data created by the position detection component, will be merged on the backend to be one. Since there are only two buttons to upload and a button to export the data from the system, Mearka meets the requirements of **Easy-to-use**. The system does not need additional input other than the video, and optionally the soccer metadata from the Mearka-app, to give positional information in return. This means that the system is automatic in terms of detecting the (pixel) positions of the players in the frame, meeting the **Automatic** requirement.

The Mearka web-interface communicates with the backend through the REST

API it provides. To use any of the endpoints, the web interface needs to get an appointed UUID, which it will send with every request. Requesting a UUID is done as soon as the user tries to upload a video or soccer metadata to the backend before sending the data. Doing this ensures the data is mapped to that UUID, meaning one layer of protection against someone trying to find a users data on the backend. It should be noted that the web interface, upon refresh, also loses its UUID. This means that a user should not refresh the page once a video is uploaded until the position detection is complete and the soccer metadata has been extracted. A good next step would be to implement the use of cookies, so the UUID could be more persistent. However, this is not implemented in the current version of Mearka

As figure 4.4 displays, the UI in terms of visual elements is slightly cluttered and leaves the impression of additional functionality. There are only a few options for the user, as described, uploading a video and soccer metadata, or exporting soccer metadata from the backend. The web interface is easy to use in terms of functionality and options. However, the visual UI could benefit from improvement.

The thought behind the green box was that it should be possible to see a bird's eye view of the field, with player positions illustrated by small circles or similar. Due to time constraints, this translation from pixel position in the frame to physical position on the field is not completed in this POC.

The grey box is put there to be a video player, playing back the video and letting the user interact with the video. There were two options for using the video player on the web interface: Playback video directly from the frontend or send the video from the backend to be played back in the web interface. Option one would work if the video is only one file and always one file. However, the system is developed and used with action cameras (**DJI Action 3**) that record multiple files, which are all part of a more extended recording. Because of this, the system is designed to let the user upload multiple videos as if they are all part of a larger recording, and the backend then concatenates them with the help of functionality on the position detection component. Because this functionality resides on the backend and in the position detection component of Mearka, with the current implementation, the only option would be to transfer the concatenated video back to the frontend for playback after processing. As figure 6.4 illustrates, the filesizes of a video could be between 40 and 80GB. Because of the potentially large files, this thesis chose not to transfer the video back after processing to save on network bandwidth. A third option could be to play back the files as a playlist, but this functionality is not implemented to reduce the web interface complexity.

The web interface is designed to make it easy for future development to add additional functionality and elements to the UI. One such option would be to implement functionality to concatenate the video on the client machine. This option could save on bandwidth, especially if the video gets slightly compressed as well. In addition, it would be easy to let the user playback the concatenated video and add functionality for tagging while the position detection component creates positional data on the backend.

## 7.3 Backend

The backend is the core component in Mearka. As illustrated by figure 5.1, both the Mearka web-interface and Mearka-app are pure interfaces for the user to interact with the backend. The backend acts as a hub of the soccer metadata and all the inputs from the user, and the results returned from the position detection component. To do this, the backend has set up a REST API with endpoints that the web interface and app can utilize. Communication with the ML component is done through a REST API set up on that component, which is monitored by its own server. Video uploaded from the web interface is received by the backend and stored on a storage volume shared between the backend and the position detection component.

When the Mearka-app tries to start a new recording, the backend generates a UUID and maps an empty soccer metadata object to that UUID, in memory, before returning the UUID to the app. Any subsequent request from the Mearka-app appends a tag to the mapped soccer metadata unless it is a request to stop recording. On a request to stop the recording, the soccer metadata is collected from memory and returned to the Mearka-app before being removed from the backend. This is done because a request from the Mearka-app to start a new recording generates a new UUID, which means the old data is not accessible to anyone.

The backend uses UUID to identify who has access to what soccer metadata. Any request, except the Mearka-app starting a recording, to the API that does not include a UUID will be refused. Using UUID was chosen because it adds a layer of security to the data being generated. The user does not need to think about UUID, as the Mearka-app and web-interface do this in the background. If the web-interface tries to upload a video, but discovers that it does not have a UUID, a request to get a UUID is sent first before trying the original request again.

Because handling of the UUIDs is done in the background, the user does not have the option to share it with collaborators or the team. As Mearka does (pixel) position detection automatically and live tagging with the Mearka-app, there is not much need to collaborate through the system, yet. However, this POC could be expanded, and in future revisions of Mearka, it might be possible to tag more extensively and add tags of different types. This could create the need for several people to be able to tag on the same video and soccer metadata simultaneously. In that case, the system would need to allow for sharing of a key, such that tags are merged and appended to the correct file on the backend, and not multiple soccer metadata files, one for each user of the system.

To store this soccer metadata, Mearka chose to use JSON for a few reasons. One is that it is widely supported by the technologies that the system utilizes. Another reason is that evaluating it against XML, JSON has one-sixth the storage requirements as XML. This is also backed up by the results described in table 6.7.

All the soccer metadata residing on the backend is kept in memory and not in persistent storage. This was done to simplify the setup during the development of the POC of Mearka and to make the data quicker to access upon request from the user. However, all the soccer metadata is lost if the backend shuts down or something happens. Therefore, storing the soccer metadata on persistent storage could be a good alternative, at least at some intervals. This does not remove the qualities of having the data in memory, but in the case of a failure, the system would still be able to regain the last stored state the data was in.

As mentioned, the backend communicates through a REST API with the position detection component. This API is run by a server on that component and lets the backend request to concatenate videos and start position detection on a video. Both requests require the backend to pass along the UUID to the user whose video it should work on. Concatenating video returns a filename to the newly concatenated video. Once position detection is complete, the return value lists all the (pixel) positions detected in that video, with the time offset into the video where the positions are detected. Since the backend knows which UUID those positions belongs to, it appends that list of positions to the soccer metadata already mapped to that UUID. The user could then extract This newly updated soccer metadata through the same web interface from which the video was uploaded.

Even though the backend gets all the positional data through the response from the position detection component, that data is also stored in a file on the shared volume. The reason for appending the data in the response is that the data is rarely larger than (observed) $\approx$ 80MB for a 45min video, and reading data from a file on disk would mean more “steps” to get the same information for the backend.

## 7.4 Position Detection Component

The main component, besides the backend, is the position detection component. This component has three parts, one server that serves a REST API, a script concatenating video, and a program that detects positions in the frame. Together, these enable Mearka to detect the positions automatically without additional user input. It also allows the system to accept multiple smaller video files that are part of a full recording and do position detection after the videos have been concatenated into one file.

The main part of this component is the ability to do position detection on a video. This is done by loading the video with OpenCV [35], and for one frame at a time, use CVlib [78] to do object detection on that frame. The object detection function is a high-level function that uses a pre-trained model, YOLOv4 [14] to detect multiple objects in the frame. The resulting list of objects is iterated over, and if it is a person, the positions are added to a list.

YOLOv4 detects multiple things in the frame: people, phones, monitors, and other types of objects. Because of this, the list of objects it returns needs to be iterated over only to save the positions of detected people. An alternative to using YOLOv4 is to train and use a custom model. This model could be trained using supervised ML methods, as explained in section 2.4.1, only to detect people. Using this custom-trained model would eliminate the need to iterate over objects found in every frame, as the positions returned would already be people only.

As the current implementation uses the CPU to do object detection, the system is able to detect roughly three frames per second. Building OpenCV and CVlib to utilize a GPU would drastically increase the throughput per second [79]. The reason that GPU support is not implemented in the current POC of Mearka is that this functionality was added later in the project. This resulted in insufficient time to get a machine with a GPU and optimize the system to use it.

Position detection speeds are influenced mainly by two things, the resolution and framerate of the video. The higher the resolution, the longer it takes to detect, as figure 6.6 illustrates. This is because the model has more pixels to cover to detect people. As figure 6.7 displays, a lot of the pixels in the frame are not part of the field or of the players.

One possible solution is to do image segmentation [80] on the video to filter out pixels that are not part of the field. This would drastically reduce the number of pixels in each frame that the position detection would need to consider. In addition, since the video is static, meaning the camera does not move during recording, the segmentation could be a static “filter” such that it does not need to be computed for every frame, only once for the entire video.

The goal of Mearka is to be able to do tagging and give feedback in real-time, such that it can be used LIVE during a game. To do this would require a more performance-optimized system, especially with regards to the position detection component and video work. A potential big step in the right direction would be the introduction of using GPUs to help detect positions.

If multiple people should be able to tag simultaneously on the same data, then a schema for sharing id as well as how to handle concurrent updates need to be developed.

The next chapter makes concluding remarks, summarizes Mearka as well as provides some future work.

# 8 Conclusion and Future Work

This chapter outlines concluding remarks about the thesis based on the problem statement and our findings. Following that is a summarization of the system before some potential future work is provided.

Revisiting the problem statement defined in section 1.2:

> *It is possible to develop a soccer tagging system based on cheap, common-of-the-shelf components. This will contrast to the state-of-the-art systems using expensive and specialized hardware and software that depend on external storing and analytics of data, where you have no control over where the data is stored, how it is used, or the quality of the tagged data, in addition to a significant time-delay.*

## 8.1 Concluding Remarks

From the problem statement above, section 3 defines the following requirements, briefely summarized here:

Functional:

1. **Input:** Allow for tagging and video input.
2. **One-click live tagging:** Possible to tag an event, live, with one press of a button.
3. **Automatically:** Extract positional data automatically from the provided video.
4. **One-click export:** Export soccer metadata generated by the system with the press of a button.
5. **Output:** Mearka should generate soccer metadata containing tags as well as positional data.
6. **Data deletion:** It should be possible to delete any data uploaded to or generated by the system.

Non-functional:

1. **Easy-to-use:** Any UI should be intuitive and easy-to-use.
2. **Soft real-time:** Strive to get real-time soccer metadata for LIVE feedback during games. With the current POC prototype, this is defined to be: Deliver soccer metadata 12 hours after a video is uploaded, containing positional data of the game.
3. **Common Of The Shelf (COTS) components:** Mearka should be as cheap as possible, therefore using COTS components to be implemented.
4. **Data ownership:** Should only store the data for as long as needed, but not longer.
5. **Privacy compliance:** The user should be able to easily upload data and delete data from the system, at will.
6. **Security:** Only the user that uploads a video or tags a game should have access to that data and any soccer metadata generated by Mearka from that data.

These requirements need to be met for Mearka to have answered the problem statement.

This thesis has presented Mearka, a POC prototype based on cheap common-of-the-shelf (**COTS**) components that can automatically detect pixel positions of players in a video, as well as tag events live field-side through a simple app.

As part of the thesis, Mearka is designed, implemented, and evaluated. The goal was to investigate if it is possible to develop a camera-agnostic soccer tagging system that utilizes cheap COTS components while still delivering valuable positional and event-tagged data within a reasonable time frame.

Through a distributed design, we have designed a system that consists of a frontend, backend, and a position detection component. The system allows for **input** from the user, both in terms of **One-click live tagging**, during a game, but also through video, recorded with any camera, uploaded to the system. From the uploaded video, Mearka is able to **automatically** extract (pixel) positional data of players in the frame. The resulting **output** is soccer metadata containing tags and positions that can be extracted with a **one-click export** button.

Section 4.3.2 and 4.4 displays the UI of Mearka. Both interfaces are simple and contain few clickable components, which indicates that it presumedly is **easy-to-use**.

To meet the **data ownership**, **privacy compliance**, and **security** requirements, user-uploaded data is not stored for longer than it needs to. In addition, each session interacting with Mearka has its own UUID, so a user uploading a video to the system cannot access anyone elses data, which is implemented for **security** concerns. Mearka web-interface allows the user to do **data deletion** of their own data whenever they want to, through a button. This was implemented because Mearka is designed with **privacy compliance** in mind.

Based on the results, the system is estimated to do position detection on a 90 minute recording within 12 hours. It is estimated to finish in ≈11,7 hours, given 1920x1080 resolution and 25FPS. When measuring the time and accuracy with higher-resolution video, the accuracy did not improve significantly by raising the resolution, even though it took longer to process. Choosing a camera angle with as few distractions in the frame as possible (illustrated by figure 6.9) improved the mean accuracy by 15% more than the highest mean accuracy measured at the main camera angle, depicted in figure 6.7.

The results demonstrate that Mearka is able to provide a soccer tagging system based on cheap, common-of-the-shelf components, and deliver on the requirement of allowing inputs from the user, automatically tag positional data based on provided, camera-agnostic video, and delivering valuable soccer metadata, that contains player positional data[1], within the soft real-time requirement of 12 hours, defined in section 3.2.

## 8.2 Summary

Mearka is designed as a distributed network of loosely coupled components communicating through HTTP and different REST API endpoints. A loose coupling between components enables the flexibility to scale the system if needed. It also allows additional functionality to be implemented as long as the existing API stays the same. A user can tag events during an ongoing session through the Mearka-app and get position data by uploading a video once the session is over through a web interface. The user does not need to give additional information or inputs to get the positional data from the system. Once Mearka has detected positions in the video, the soccer metadata can be downloaded as a JSON file through one click of a button on the web interface.

Position detection is done by reading one frame at a time from the uploaded video with OpenCV, and using CVlib combined with the pre-trained model, YOLOv4, to do object detection on each frame. If the model finds a person, the positions are saved in a list and returned to the backend to be appended to the user's soccer metadata.

The evaluation investigates the throughput and accuracy of the system as well as the battery performance of the chosen camera used during development.

Based on the results, Mearka can deliver valuable positional data within the *soft real-time* requirement of 12 hours[2]. It also alludes to the CPU being the current bottleneck, as Mearka maxes out at around three frames per second while detecting. Three FPS max does not change much, even if the resolution differs by four times between the lowest and highest resolution investigated. The results also illustrate that camera position is more important than increasing the resolution to improve detection accuracy, as described in figure 6.8.

To develop Mearka, the **DJI Action 3** camera was chosen. It can record long enough[3], in almost any settings combination[4]. *DJI Action 3* has a wide variety

1. Player positions are pixel positions in the frame for this thesis.
2. Estimation based on a video recording with resolution 1920x1080 and 25 FPS. The video length is set to 90 minutes to mimic a minimum length soccer game.
3. The requirement is set to minimum 105 minutes, as a football match is 90 minutes plus 15 min break.
4. 3840x2160 resolution at 60FPS, with the camera screen turned on during recording, is the

of resolutions and FPS settings, enabling the user to record how they want. It can stream to an RTMP server [82], which can be additional functionality to implement in future work.

## 8.3 Future Work

Mearka is currently a simple prototype, and there are many potential directions for future work.

### 8.3.1 Streaming

As Mearka is implemented, it requires an already recorded file through the web interface for position detection. However, the end goal of Mearka is to be able to do position detection and tagging in real-time. One step towards real-time tagging and position detection is to expand the system to enable cameras to stream directly instead of waiting for a recorded file to be uploaded. As stated in section 4.2.2, the *DJI Action 3* camera chosen can stream video through RTMP [82], which could be used to help develop the system to allow for streaming.

### 8.3.2 Tracking

In Mearka, position detection is done by using CVlib with the pre-trained model YOLOv4 to detect the pixel positions in the image. However, the current implementation does not track the bounding boxes produced by CVlib across multiple frames. This means that the positions on the current frame do not relate to positions on the previous frame. Tracking each position and giving IDs or names to each bounding box would help differentiate players and teams.

In the ByteTrack-paper, the authors present a compelling and generic method to track detection boxes in a video, by associating almost every detection box between frames [83] instead of tracking only the detection boxes with a high enough confidence level. This method outperformed several other trackers like FairMOT [84] and CorrTracker [85], as illustrated by the comparison in [83, figure 1].

only combination that does not meet the requirement.

### 8.3.3 Extend Tagging Option

Currently, it is possible to tag through the Mearka-app while recording and get positional data through the Mearka web-interface afterwards. The tags created by the Mearka-app do not have any information since they only note that something happened at a given offset. Extending the system, web-interface, or Mearka-app to allow for more details to be filled in and more detailed tagging would yield more data that can be analyzed.

Uploading the metadata and video to the Mearka web-interface should allow the user to scrub through the video, jump between tags (as well as jump to the correct timestamp in the video), update tags, as well as add additional tags of different types.

#### Custom Tags

What type of tags and who (players) are tagged should be customizable, as different games might need different tags. This can potentially be done by adding a field in the metadata that lists the tags used and the people that can be tagged. Upon uploading the metadata, these fields can be parsed, and the options made available to the user through the UI.

#### Voice Recording

When an event is tagged, a voice recording can be started to record a short message from the user of the event they have tagged. This recording could simplify the tagging process in two ways.

Firstly, the need to look closely at the video to understand what was tagged is removed, as the essential things to note are said in the audio recording.

Improving on this can be done by using Natural Language Processing (NLP [86]), which is a form of ML, to recognize what is being said and add tags automatically. This would more immediately give more details in the tags, even those created by the Mearka-app, match-side, as the game is ongoing.

### 8.3.4 Translate pixel-positions to real world positions

Positional data in Mearka is currently the pixel coordinates in the frame of the upper left and lower right corner of a bounding box containing the detected person. Although this is valuable information, the next step is to convert these pixel positions from the frame to real-world coordinates (on a field). This would improve the data by better representing where the players are on the field, not only where in the frame they are. An example of how this could be illustrated to the user is depicted in figure 8.1.

Liang Peng proposes a system, "Pixel2Field", that does this translation between frame pixel coordinate to 2d field positions [65]. This system can be used as inspiration for tackling this problem.

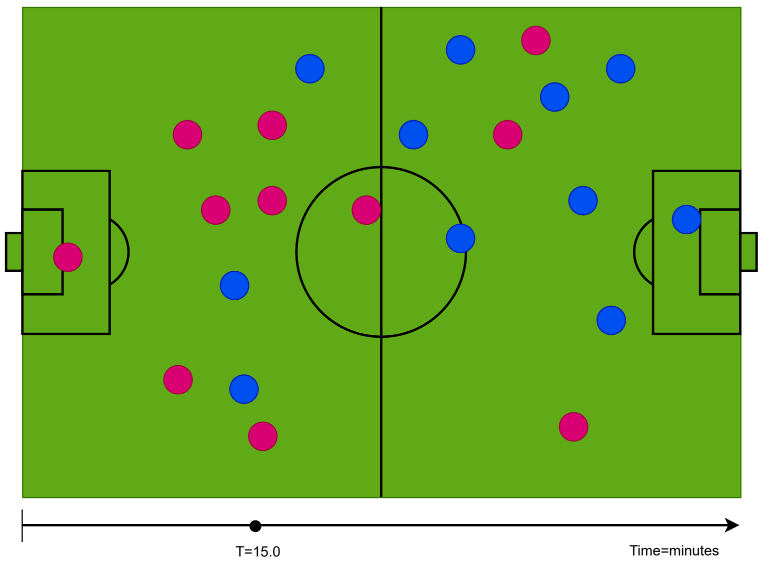


**Figure 8.1:** Example: Pixel-coordinate translated to field-positions

### 8.3.5 Video queue

The current implementation interprets an upload of multiple videos to be multiple video segments of the same video. These segments are concatenated into one video before position detection is run. Giving the user the option to choose if the videos are multiple parts of the same video or different videos could improve the system's value proposition.

This would allow the user to upload multiple matches or video clips and get tailored data for each clip. The system could utilize a queue where each video gets a unique id. Upon upload, the videos are placed in a processing queue on the backend. Allowing for batch processing since the position detection component can move on to the next video once one video position detection is complete.

### 8.3.6 Possible Real-Time

As section 6 illustrates, Mearka can process ≈3 FPS with the current setup. Processing video with 25FPS means it is not able to detect positions in real-time. However, processing only three video frames per second would possibly enable the system to give data real-time, as the number of frames needed to be processed is within the throughput of the system.

H.264 video is compressed by having frames containing values on every pixel only with a specific interval, e.g., every 10th frame. Every frame between these only stores the pixel values that have changed since the last “keyframe”.

If Mearka is set up to only process and evaluate those “keyframes”, it could potentially be efficient enough to be used to detect positions in real-time.

# Bibliography

[1] Sebastian Lyng Johansen. “Dárkon.” Submitted for review, May 2023. MA thesis. UiT - The Artic University of Norway, May 22, 2023.

[2] Rory Smith. “Expected Goals:” in: *The story of how data conquered football and changed the game forever*. HarperCollins Publishers, Sept. 2022, p. 299.

[3] Michael Lewis. “Moneyball:” in: *The Art of Winning an Unfair Game*. W. W. Norton & Company, Mar. 2004, p. 336. ISBN: 0393324818.

[4] www.statsperform.com. *STATS Acquires Prozone*. 2015. URL: `https://www.statsperform.com/press/stats-acquires-prozone/` (visited on May 7, 2023).

[5] Inc. Agile Sports Technologies. *Hudl*. 2007-2022. URL: `https://www.hudl.com/` (visited on Dec. 12, 2022).

[6] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. “Soccer Video and Player Position Dataset.” In: *Proceedings of the 5th ACM Multimedia Systems Conference*. MMSys ’14. Singapore, Singapore: Association for Computing Machinery, 2014, pp. 18–23. ISBN: 9781450327053. DOI: `10.1145/2557642.2563677`. URL: `https://doi.org/10.1145/2557642.2563677`.

[7] STATSports Group Limited. *APEX Athlete Series - GPS Performance Tracker*. 2023. URL: `https://eu.shop.statsports.com/products/apex-athlete-series` (visited on Apr. 20, 2023).

[8] www.footovision.com. *Reveal the true power of sports analytics*. 2021. URL: `https://www.footovision.com/` (visited on May 8, 2023).

[9] soccerlytics.com. *Succeed With Data-Driven Soccer Analytics*. 2023. URL: `soccerlytics.com` (visited on May 8, 2023).

[10] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, Paul R. Young, and Peter J. Denning. “Computing as a Discipline.” In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: `10.1145/63238.63239`. URL: `https://doi.org/10.1145/63238.63239`.

[11] www.blackmagicdesign.com. *DaVinci Resolve 18 - Professional Editing, Color, Effects and Audio Post!* 2023. URL: `https://www.blackmagicdesign.com/products/davinciresolve` (visited on May 10, 2023).

[12] Casey Faris. *RESOLVE 18 CRASH COURSE - Davinci Resolve 18 Walkthrough*

*BEGINNER*

. 2022. URL: `https://www.youtube.com/watch?v=h9MrEaEL12M` (visited on Nov. 11, 2022).

[13] Skills Factory. *DaVinci Resolve 18 - Tutorial for Beginners in 15 MINUTES!*

*COMPLETE*

. Apr. 2022. URL: `https://www.youtube.com/watch?v=aLIHKHkvKMM` (visited on May 10, 2023).

[14] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." In: *arXiv e-prints*, arXiv:2004.10934 (Apr. 2020), arXiv:2004.10934. DOI: `10.48550/arXiv.2004.10934`. arXiv: `2004.10934 [cs.CV]`.

[15] G. Hartvigsen and D. Johansen. "Co-operation in a distributed artificial intelligence environment—The StormCast application." In: *Engineering Applications of Artificial Intelligence* 3.3 (1990), pp. 229–237. ISSN: 0952-1976. DOI: `https://doi.org/10.1016/0952-1976(90)90046-O`. URL: `https://www.sciencedirect.com/science/article/pii/095219769090046O`.

[16] Haakon Riiser, Pål Halvorsen, Carsten Griwodz, and Dag Johansen. "Low Overhead Container Format for Adaptive Streaming." In: *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. MMSys '10. Phoenix, Arizona, USA: Association for Computing Machinery, 2010, pp. 193–198. ISBN: 9781605589145. DOI: `10.1145/1730836.1730859`. URL: `https://doi.org/10.1145/1730836.1730859`.

[17] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. "Overview of the H.264/AVC video coding standard." In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), pp. 560–576. DOI: `10.1109/TCSVT.2003.815165`.

[18] Svein A Pettersen, Håvard D Johansen, Ivan A M Baptista, Pål Halvorsen, and Dag Johansen. "Quantified Soccer Using Positional Data: A Case Study." In: *Frontiers in physiology.* 9 (2018), p. 866. ISSN: 1664-042X.

[19] IBM. *What is a REST API?* URL: `https://www.ibm.com/topics/rest-apis` (visited on May 18, 2023).

[20] www.json.org. *Introducing JSON*. 1999. URL: `https://www.json.org/json-en.html` (visited on May 14, 2023).

[21] mozilla.org. *Working with JSON*. 1998-2023. URL: `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON` (visited on May 14, 2023).

[22] Python Software Foundation. *JSON encoder and decoder*. 2023. URL: `https://docs.python.org/3/library/json.html` (visited on May 14, 2023).

[23] Inc. Meta Platforms. *Networking - Handling the response*. 2023. URL: `https://reactnative.dev/docs/network` (visited on May 14, 2023).
[24] Andrew Gerrand. *JSON and Go*. 2011. URL: `https://go.dev/blog/json` (visited on May 14, 2023).
[25] FFmpeg.org. *A complete, cross-platform solution to record, convert and stream audio and video*. 2023. URL: `https://ffmpeg.org/` (visited on Apr. 21, 2023).
[26] Ethem Alpaydin and Francis Bach. *Introduction to Machine Learning*. eng. 3rd ed. Adaptive computation and machine learning. Cambridge: MIT Press, 2014. ISBN: 0262028182.
[27] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollar, and Larry Zitnick. "Microsoft COCO: Common Objects in Context." In: *ECCV*. European Conference on Computer Vision, Sept. 2014. URL: `https://www.microsoft.com/en-us/research/publication/microsoft-coco-common-objects-in-context/`.
[28] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. "Deep Neural Networks for Object Detection." In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger. Vol. 26. Curran Associates, Inc., 2013. URL: `https://proceedings.neurips.cc/paper_files/paper/2013/file/f7cade80b7cc92b991cf4d2806d6bd78-Paper.pdf`.
[29] Léon Bottou. "Stochastic Learning." In: *Advanced Lectures on Machine Learning*. Ed. by Olivier Bousquet and Ulrike von Luxburg. Lecture Notes in Artificial Intelligence, LNAI 3176. Berlin: Springer Verlag, 2004, pp. 146–168. URL: `http://leon.bottou.org/papers/bottou-mlss-2004`.
[30] "Mean Squared Error." In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 337–339. ISBN: 978-0-387-32833-1. DOI: `10.1007/978-0-387-32833-1_251`. URL: `https://doi.org/10.1007/978-0-387-32833-1_251`.
[31] "Mean Absolute Error." In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 652–652. ISBN: 978-0-387-30164-8. DOI: `10.1007/978-0-387-30164-8_525`. URL: `https://doi.org/10.1007/978-0-387-30164-8_525`.
[32] Kaan Gokcesu and Hakan Gokcesu. *Generalized Huber Loss for Robust Learning and its Efficient Minimization for a Robust Statistics*. 2021. arXiv: `2108.12627 [stat.ML]`.
[33] J. MacQueen. *Some methods for classification and analysis of multivariate observations*. English. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967). 1967.
[34] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey." In: *J. Artif. Int. Res.* 4.1 (May 1996), pp. 237–285. ISSN: 1076-9757.

[35] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. “A brief introduction to OpenCV.” In: *2012 Proceedings of the 35th International Convention MIPRO*. 2012, pp. 1725–1730.

[36] The Apache Software Foundation. *APACHE LICENSE, VERSION 2.0*. Tech. rep. The Apache Software Foundation, Jan. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0 (visited on Apr. 23, 2023).

[37] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (visited on Apr. 23, 2023).

[38] Keras-team. *Keras*. 2023. URL: https://github.com/keras-team/keras.

[39] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You Only Look Once: Unified, Real-Time Object Detection.” In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[40] Dag Johansen, Magnus Stenhaug, Roger B. A. Hansen, Agnar Christensen, and Per-Mathias Høgmo. “Muithu: Smaller footprint, potentially larger imprint.” In: *Seventh International Conference on Digital Information Management (ICDIM 2012)*. 2012, pp. 205–214. DOI: 10.1109/ICDIM.2012.6360105.

[41] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Østein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. “Bagadus: An Integrated Real-Time System for Soccer Analytics.” In: *ACM Trans. Multimedia Comput. Commun. Appl.* 10.1s (Jan. 2014). ISSN: 1551-6857. DOI: 10.1145/2541011. URL: https://doi.org/10.1145/2541011.

[42] Mohammed Amine El Mrabet, Khalid El Makkaoui, and Ahmed Faize. “Supervised Machine Learning: A Survey.” In: *2021 4th International Conference on Advanced Communication Technologies and Networking (CommNet)*. 2021, pp. 1–10. DOI: 10.1109/CommNet52204.2021.9641998.

[43] Frank F. Tsui. *Essentials of software engineering*. eng. Burlington, Mass, 2014.

[44] Axis Communications AB. *AXIS P3807-PVE Network Camera*. 2023. URL: https://www.axis.com/products/axis-p3807-pve#technical-specifications (visited on May 12, 2023).

[45] ONVU Technologies Group AG. *C-08 Outdoor Camera*. 2023. URL: https://www.oncamgrandeye.com/product/c-08-outdoor-camera/ (visited on May 12, 2023).

[46] www.PacerGroup.Net. *Understanding IP Certification*. 2018. URL: https://www.pacergroup.net/pacer-news/understanding-ip-certification/ (visited on May 12, 2023).

[47] Insta360. *Insta360 X3*. Sept. 2022. URL: https://store.insta360.com/product/x3?utm_source=website&utm_medium=product_page_button&utm_campaign=x3 (visited on May 12, 2023).

[48] Insta360. *Insta360 ONE RS*. Mar. 2022. URL: https://store.insta360.com/product/one_rs?utm_source=website&utm_medium=product_page_button&utm_campaign=one_rs (visited on May 12, 2023).

[49] GoPro Inc. *MAX*. Oct. 2019. URL: https://store.insta360.com/product/one_rs?utm_source=website&utm_medium=product_page_button&utm_campaign=one_rs (visited on May 12, 2023).

[50] GoPro Inc. *GoPro Camera Battery Life*. Oct. 2022. URL: https://community.gopro.com/s/article/gopro-camera-battery-life?language=en_US#max (visited on May 12, 2023).

[51] GoPro Inc. *HERO11 Black*. Sept. 2022. URL: https://gopro.com/en/us/shop/cameras/hero11-black/CHDHX-111-master.html?option-id=CHDHX-111-master (visited on May 12, 2023).

[52] GoPro Inc. *HERO11 Black: Digital Lenses FOV Information*. Sept. 2022. URL: https://community.gopro.com/s/article/HERO11-Black-Digital-Lenses-FOV-Information?language=en_US (visited on May 12, 2023).

[53] "kafi65" GoPro Inc. *Overheating Problem with GoPro 11 Black*. Nov. 2022. URL: https://community.gopro.com/s/question/0D53b00008o6E8bCAE/overheating-problem-with-gopro-11-black?language=en_US (visited on May 12, 2023).

[54] "goldenc4312" GoPro Inc. *Over heating problem with HERO 11 Black*. Apr. 2023. URL: https://community.gopro.com/s/question/0D53b00009DNsxZCAT/over-heating-problem-with-hero-11-black?language=en_US (visited on May 12, 2023).

[55] GoPro Inc. *Over heating problem with HERO 11 Black*. Sept. 2022. URL: https://community.gopro.com/s/article/HERO11-Black-Camera-Is-Too-Hot?language=en_US (visited on May 12, 2023).

[56] DJI. *Osmo Action 3 Adventure Combo*. Sept. 2022. URL: https://store.dji.com/sk/product/osmo-action-3?site=brandsite&from=landing_page&vid=120471 (visited on May 13, 2023).

[57] MarcoR. *Why the Action 3 is out of focus - a great explanation video*. 2022. URL: https://forum.dji.com/thread-276860-1-1.html (visited on May 14, 2023).

[58] rex_alpha3 Et al. Effective_Youth_8036 xavster. *DJI Osmo Action 3 Batch 11/22 no more focus issues?* 2022. URL: https://www.reddit.com/r/dji/

comments/z6h3db/dji%5C_osmo%5C_action%5C_3%5C_batch%5C_1122%5C_no%5C_more%5C_focus%5C_issues/ (visited on May 14, 2023).

[59] JustName. *Tutorial how to fix the Osmo Action 3 focus issue in 5 Minutes*. 2022. URL: https://forum.dji.com/thread-280288-1-1.html (visited on May 14, 2023).

[60] Android. *What is Android*. URL: https://www.android.com/what-is-android/ (visited on May 16, 2023).

[61] The GraphQL Foundation. *A query language for your API*. 2023. URL: https://graphql.org/ (visited on May 19, 2023).

[62] Armin Lawi, Benny L. E. Panggabean, and Takaichi Yoshida. "Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System." In: *Computers* 10.11 (Oct. 2021), p. 138. ISSN: 2073-431X. DOI: 10.3390/computers10110138. URL: http://dx.doi.org/10.3390/computers10110138.

[63] Wikipedia contributors. *ExFAT — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=ExFAT&oldid=1155042128 (visited on May 20, 2023).

[64] Microsoft. *File Systems*. 2008. URL: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938937(v=technet.10)%5C?redirectedfrom=MSDN (visited on May 20, 2023).

[65] Liang Peng. "Pixel2Field Single Image Transformation to Physical Field of Sports Videos." In: Oct. 2019. ISBN: 978-3-030-33719-3. DOI: 10.1007/978-3-030-33720-9_51.

[66] Google. *Build simple, secure, scalable systems with Go*. 2023. URL: https://go.dev/ (visited on May 22, 2023).

[67] Gin Team. *Gin Web Framework*. 2022. URL: https://gin-gonic.com/ (visited on May 22, 2023).

[68] Python Software Foundation. *Python*. 2023. URL: https://www.python.org/ (visited on May 22, 2023).

[69] Pallets. *Flask*. 2010. URL: https://flask.palletsprojects.com/en/2.3.x/ (visited on May 22, 2023).

[70] Meta Open Source. *React - The library for web and native user interfaces.* 2023. URL: https://react.dev/ (visited on May 22, 2023).

[71] Segun Adebayo. *Create accessible React apps with speed*. 2023. URL: https://chakra-ui.com/ (visited on May 22, 2023).

[72] expo.dev. *Create amazing apps that run everywhere*. 2023. URL: https://docs.expo.dev/ (visited on May 22, 2023).

[73] mozilla.org contributors. *HTTP*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP (visited on May 22, 2023).

[74] Microsoft. *TypeScript is JavaScript with syntax for types*. 2023. URL: https://www.typescriptlang.org/ (visited on May 22, 2023).

[75] Google. *google/uuid*. 2021. URL: https://github.com/google/uuid (visited on May 23, 2023).

[76] The Python Software Foundation. *5.5. Dictionaries*. 2023. URL: https://docs.python.org/3/tutorial/datastructures.html?highlight=dictionary (visited on May 23, 2023).

[77] Codecademy. *7 Top Machine Learning Programming Languages*. 2023. URL: https://www.codecademy.com/resources/blog/machine-learning-programming-languages/ (visited on May 23, 2023).

[78] arunponnusamy. *cvlib*. Jan. 11, 2021. URL: https://github.com/arunponnusamy/cvlib (visited on May 23, 2023).

[79] Ebubekir BUBER and Banu DIRI. "Performance Analysis and CPU vs GPU Comparison for Deep Learning." In: *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[80] Raksha Kale and Dr Thorat. "Image Segmentation Techniques with Machine Learning." In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (Dec. 2021), pp. 232–235. DOI: 10.32628/CSEIT1217653.

[81] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008. URL: https://www.w3.org/TR/REC-xml/REC-xml-20081126.xml (visited on May 26, 2023).

[82] M. Thornburgh H. Parmar. *Adobe's Real Time Messaging Protocol*. Dec. 21, 2012. URL: https://rtmp.veriskope.com/pdf/rtmp_specification_1.0.pdf (visited on May 29, 2023).

[83] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. *ByteTrack: Multi-Object Tracking by Associating Every Detection Box*. 2022. arXiv: 2110.06864 [cs.CV].

[84] Yifu Zhang, Chunyu Wang, Xinggang Wang, Wenjun Zeng, and Wenyu Liu. "FairMOT: On the Fairness of Detection and Re-identification in Multiple Object Tracking." In: *International Journal of Computer Vision* 129.11 (Sept. 2021), pp. 3069–3087. DOI: 10.1007/s11263-021-01513-4. URL: https://doi.org/10.1007%2Fs11263-021-01513-4.

[85] Qiang Wang, Yun Zheng, Pan Pan, and Yinghui Xu. *Multiple Object Tracking with Correlation Learning*. 2021. arXiv: 2104.03541 [cs.CV].

[86] Kai Jiang and Xi Lu. "Natural Language Processing and Its Applications in Machine Translation: A Diachronic Review." In: *2020 IEEE 3rd International Conference of Safe Production and Informatization (IICSPI)*. 2020, pp. 210–214. DOI: 10.1109/IICSPI51290.2020.9332458.

[87] Vincent T. *Progressive vs. Interlaced*. Dec. 22, 2019. URL: https://medium.com/hd-pro/progressive-vs-interlaced-e18e2924800e (visited on May 24, 2023).

# A

# Appendix

## A.1 Progressiv versus interlaced scan modes

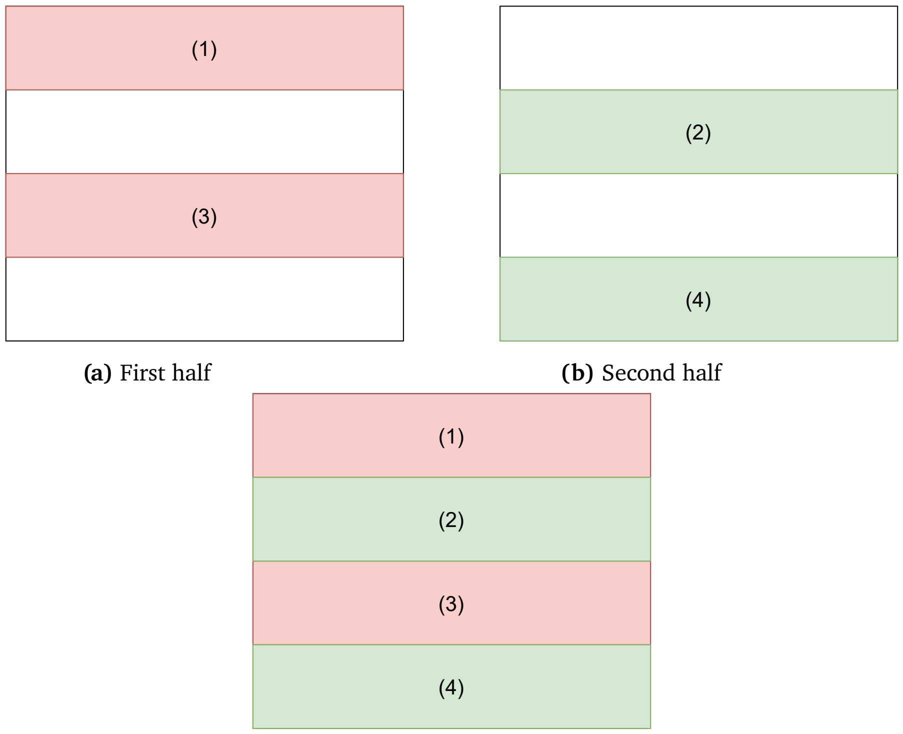
When talking about progressive or interlaced video, it refers to how the image is displayed on the screen.

An image is a matrix of pixels. For the resolution 1920x1080, which denotes the resolution of FullHD, there are 1080 rows of 1920 pixels. When all 1080 rows of 1920 pixels are displayed on top of each other, the image is displayed in full.

Progressive video is the most common way to display videos today, and it displays the entire frame at a time [87]. For video recorded in 25 FPS, this means that every $\frac{1}{25}$th of a second, a new frame is displayed.

**Figure A.1:** Progressive image

Contrary to progressive, interlaced video displays half of the frame at a time in rapid succession to make it look like one whole frame [87]. For video with 25 FPS, the interlaces method must send 50 half-frames each second to make up the 25 complete frames. To display half-frames, odd-numbered rows are sent first, and then the even numbered rows after, which then completes the image. Sending odd and even numbered rows is illustrated in figure A.2a, A.2b, while figure A.2c illustrates that the two halves make up one frame when combined.



**(a)** First half **(b)** Second half

**(c)** Combined frame

**Figure A.2:** Interlaced video overview