

# ARTIFICIAL INTELLIGENCE IN SERVICE SYSTEMS: APPLICATIONS IN COMPUTER VISION

Project Description  
by

Julian Sauer  
Marcel Frühholz  
Louis Skowronek  
Alexander Rothmaier  
Constantin Hannes Ernstberger

At the Department of Economics and Management

Digital Service Innovation (DSI)

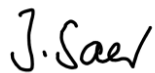
Karlsruhe Service Research Institute (KSRI) &  
Institute of Information Systems and Marketing (IISM)

Advisor:	Prof. Dr. Gerhard Satzger
Date of Submission:	14.08.2023

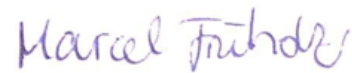
# Declaration of Academic Integrity

I hereby confirm that the present thesis is solely my own work and that if any text passages or diagrams from books, papers, the web, or other sources have been copied or in any other way used, all references—including those found in electronic media—have been acknowledged and fully cited.

Karlsruhe, 14.08.2023



Julian Sauer



Marcel Frühholz



Alexander Rothmaier



Constantin Hannes Ernstberger



Louis Skowronek

# Contents

<b>List of Figures .....</b>	<b>v</b>
<b>1 Problem statement .....</b>	<b>1</b>
<b>2 Methodological approach .....</b>	<b>4</b>
<b>3 Functional and non-functional requirements .....</b>	<b>7</b>
3.1 Functional Requirements.....	7
3.2 Non-functional Requirements.....	8
<b>4 Project Plan and Effort Estimation .....</b>	<b>10</b>
<b>5 Hardware Specifications and Deployment .....</b>	<b>15</b>
5.1 NVIDIA Jetson Nano Developer Kit and CSI Camera.....	15
5.2 Hardware Setup and Configuration .....	15
5.3 Physical Installation and Environment Setup .....	16
5.4 Deployment .....	17
<b>6 Software Architecture and Workflow .....</b>	<b>18</b>
6.1 Model.....	18
6.2 View.....	20
6.3 Controller.....	21
6.4 Unit Tests .....	22
<b>7 Use Cases &amp; Design Principles .....</b>	<b>23</b>
7.1 Minimum viable use case .....	23
7.2 Additional use cases .....	23
<b>8 Model .....</b>	<b>25</b>
8.1 Dataset Generation .....	26
8.2 Training.....	26
8.3 Evaluation.....	27
8.3.1 Confusion Matrix .....	27
8.3.2 Precision, Recall and F1-Score.....	29
<b>9 Challenges .....</b>	<b>32</b>
9.1 Annotation Process Challenges.....	32
9.2 Deployment Challenges .....	34
<b>10 Operating Instructions .....</b>	<b>36</b>

10.1	End-2-End Operating Concept.....	36
10.2	User Manual.....	37
10.2.1	Getting Started.....	37
10.2.2	Assembly Mode .....	38
10.2.3	Disassembly Mode .....	39
10.2.4	Advanced Settings .....	40
<b>11</b>	<b>Conclusion .....</b>	<b>41</b>
<b>12</b>	<b>Appendix.....</b>	<b>43</b>
12.1	Labeling Guide: Dos and Don'ts.....	43
12.2	Lego Assembly Instruction.....	46
12.3	Naming Conventions .....	49
<b>13</b>	<b>References.....</b>	<b>ix</b>

# List of Figures

Figure 1: Methodological approach.....	4
Figure 2: Project Gantt Chart.....	10
Figure 3: Tasks mapped by complexity and size .....	14
Figure 4: Setup of physical installation .....	16
Figure 5: Model View Controller (MVC) of the solution.....	18
Figure 6: Yolo Benchmarks.....	25
Figure 7: CV Model Pipeline.....	26
Figure 8: Size of the Training Dataset.....	27
Figure 9: Confusion Matrix.....	28
Figure 10: Precision-Recall Curve .....	29
Figure 11: F1-Confidence Curve .....	30
Figure 12: Flawed EXIF Orientation.....	32
Figure 13: Webapp Homepage .....	38
Figure 14: Webapp Assembly Mode.....	39
Figure 15: Advanced Settings Dialog.....	40

# 1 Problem statement

In today's dynamic and ever-evolving industrial landscape, remanufacturing has become a frequent topic of discussion in reducing energy consumption and emissions while maintaining quality requirements that arise from high customer expectations and demanding component requirements. Particularly for companies in the precision mechanics and optics industry like Zeiss, a remanufacturing approach can result in numerous challenges that can hinder efficiency, inhibit quality control, and decrease overall productivity. As a result, companies operating in this sector must constantly seek innovative solutions to optimize their processes and stay competitive. One such solution lies in harnessing the power of computer vision technology, which has the potential to revolutionize remanufacturing operations.

This project description aims to explore the application of computer vision in addressing the challenges faced by companies like Zeiss entering the remanufacturing landscape. Therefore, we will delve into specific use cases that leverage computer vision to tackle some of the challenges of industry practice like the variable quality of parts, dependence on technical expertise, large variety of parts and subcomponents, dynamic processes requiring flexibility, as well as increased component complexity leading to new defect types. In the scope of this project, the problem complexity has been broken down to its core components of object recognition and, compared to industry applications, relatively simple geometries, by (re-) manufacturing Lego Mindstorm components. However, this serves as a blueprint for the industry as the encountered issues and learnings are applicable across different tiers of complexity and scopes of implementation. Before diving deeper into the approach taken to tackle some of the mentioned issues, we first want to elaborate more on each of the challenges and potential use cases to make the application of computer vision for remanufacturing more comprehensible in a practical context.

Companies in the remanufacturing industry often encounter the variable quality of parts, which can significantly impact the efficiency and reliability of the remanufacturing process. Factors such as wear and tear, previous usage conditions, and manufacturing discrepancies contribute to this variability. To mitigate this challenge, one can implement computer vision systems to automate identifying and classifying core parts based on predefined quality criteria. This allows for efficient quality control and minimizes the risk of incorporating faulty parts into remanufacturing.

Dependence on technician expertise is another challenge faced by remanufacturing companies. Traditional processes heavily rely on the knowledge and skills of technicians. By integrating computer vision, one can provide real-time feedback and guidance to technicians during the remanufacturing process. The use of computer vision systems can help to identify assembly errors

and offer instant notifications or suggestions to improve assembly accuracy for less-trained personnel. This reduces the dependency on individual expertise, ensures consistent quality standards, and enhances operational efficiency.

The remanufacturing industry often deals with a wide variety of parts and subcomponents, each with its unique characteristics, shapes, and sizes. Effectively managing this complexity is crucial for streamlined and accurate operations. One can leverage computer vision algorithms to recognize and categorize different parts and subcomponents which are disordered like in a bin picking problem, thereby facilitating efficient assembly line management. With the ability to automatically identify the required components for each product or assembly step, computer vision accelerates the (re-) assembly process, minimizes humane-made errors, and improves overall productivity.

Dynamic processes requiring flexibility are inherent to the remanufacturing industry. Rapid changes in product types, specifications, and customer requirements demand adaptability. Computer vision provides a solution by enabling the training and learning of assemblers, especially when dealing with new or infrequently encountered components. Through the provision of visual instructions, augmented reality overlays, or step-by-step guidance, computer vision empowers assemblers to quickly acquire the necessary skills and knowledge. This reduces training time, enhances adaptability, and ensures smooth and efficient operations despite changing requirements.

With the increasing trend of electrification in industries, remanufacturing companies face the challenge of handling complex electronic components and systems. The intricate nature of these components makes defect identification and proper assembly more demanding. One can inspect intricate sub-components, such as circuit boards or microchips, by leveraging computer vision. Computer vision systems can analyze visual data to identify defects, assess soldering quality, and ensure proper connections. This guarantees the overall reliability and functionality of the remanufactured products.

Considering the wide range of potential use cases and the importance of tailoring the computer vision model to specific end-user applications, our project team has decided to postpone the final decision on one of the defined use cases to a point where the model development is just before being tailored to end-user applications. This approach allows for the development of a base computer vision model that can be adapted to different remanufacturing processes, reducing the marginal effort required for use case selection. We want to ensure that we understand the very specific issues of a hands-on remanufacturing situation to attain a proper problem-solution fit while ensuring a seamless customer experience. The customer in this case can be interpreted as a worker at an assembly line who is reliant on error-free and problem-specific supportive tools.

In the following section, we will discuss our methodological approach to develop and iteratively improve a functional computer vision model. Additionally, we will outline our considerations for

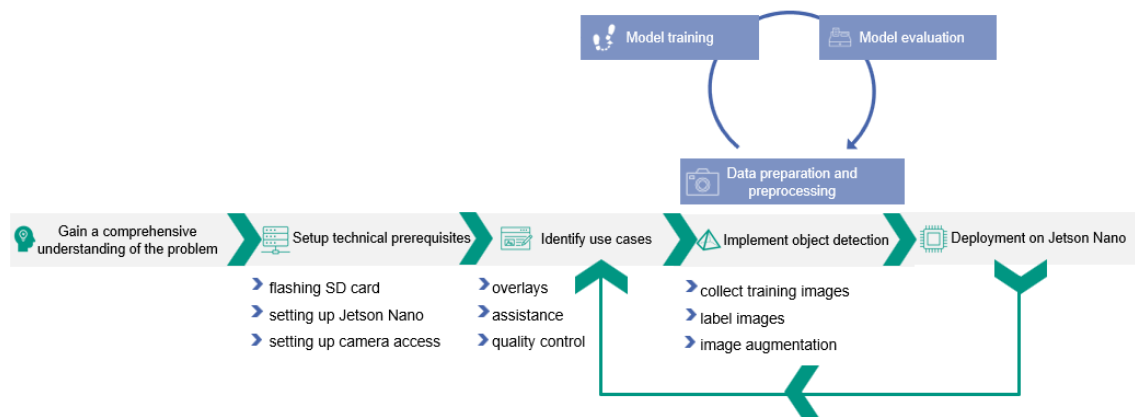
identifying and developing use cases, considering the specific requirements and challenges faced by companies like Zeiss. By harnessing the potential of computer vision technology, such companies can transform their remanufacturing operations, enhance quality control, increase productivity, and ultimately gain a competitive edge in the market. Thereafter, subsequent sections of this project description will delve into the technical aspects of our computer vision prototype, including its functions, design principles, and the hurdles we encountered during the project as well as an end-to-end operating concept and a user manual.



## 2 Methodological approach

Our project team followed a systematic and iterative methodology to successfully integrate computer vision technology into the remanufacturing processes. This approach depicted in Figure 1 consists of five sequential steps, with two iterations embedded within them. The five steps consist of the following:

- Step 1: Gain a Comprehensive Understanding of the Problem
- Step 2: Setup Technical Prerequisites
- Step 3: Identify Use Cases
- Step 4: Implement Object Detection (First Iteration)
- Step 5: Deployment of Jetson Nano (Second Iteration)



**Figure 1: Methodological approach**

The first iteration occurs in Step 4, which involves implementing object detection. The iteration loop includes three sub-steps: data preparation and preprocessing, model training, and model evaluation. The second iteration takes place between Step 5, which focuses on deploying Jetson Nano, and Step 3, which entails identifying use cases. Before elaborating on the iteration loops, in the following, the sequential steps are investigated more closely.

### Step 1: Gain a Comprehensive Understanding of the Problem

In this initial phase, our project team extensively studied the challenges faced in the remanufacturing processes as well as breaking down the problem of building a computer vision prototype that can detect a selected object in images and video into more digestible work packages that can then be distributed among the team members. This understanding guided our subsequent steps and enabled us to tailor the computer vision solution to specific operational needs.

### Step 2: Setup Technical Prerequisites

Before diving into the development process, we needed to ensure that all the necessary technical prerequisites are in place. This includes setting up the Jetson Nano, as an edge computing device with GPU, intended for applications of computer vision. The Jetson Nano setup involves tasks such as flashing the SD card with the appropriate operating system, configuring system settings, and connecting peripherals. Similarly, camera access setup involved configuring camera interfaces, installing drivers, and ensuring compatibility between the camera and the Jetson Nano platform.

### Step 3: Identify Use Cases

With a comprehensive understanding of the challenges and technical prerequisites, we identified specific use cases where computer vision could be applied effectively within our capabilities and time constraints. These use cases could include overlays to guide technicians, quality control to detect faulty assembly, or real-time feedback systems for error prevention. In this section, we planned a bottom-up approach starting with the most feasible use case to solve. As soon as this use case was up and running, we extended the scope of our application and added additional functionalities. The procedure and concrete use cases chosen will be elaborated on in the section *Use Cases and Design Principles*.

### Step 4: Implement Object Detection

In this step, our team focused on the implementation of an object detection model for our use case. Therefore, we leveraged a pre-trained object detection model and finetuned it on our custom data. The images of the Lego pieces were captured by phone cameras and then manually labelled using LabelImg as annotation tool. Regarding the annotation effort, we opted for rectangular bounding boxes and decided against polygons. Throughout the course of the project, our whole labelling process underwent a learning curve and we increasingly adjusted different parts to further streamline our process, as discussed in later parts. To further increase our dataset, we used different image augmentation techniques such as rotations, flips, scaling, mosaic and adding Gaussian noise. By applying these techniques, the model also becomes more robust and capable of handling diverse object orientations, distances to the camera, lighting conditions, and backgrounds.

Simultaneously, we started conceptualizing and implementing the User Interface as well as the architecture underneath it. The architecture follows the Model-View-Controller design pattern. Followingly, we set up a Flask web server representing the controller, built an HTML-based web page as the view, and embedded the AI model. Additionally, we defined HTTP requests to establish communication between the three components handled by the web server.

### Step 5: Deployment of Jetson Nano

The deployment marks the final step in our methodology and deals with the deployment of the finetuned object detection model on the Jetson Nano. This involves transferring the model to the Jetson Nano device by configuring the necessary runtime environments and dependencies as well as optimizing the performance to ensure a low-latency inference of the model that enables its application in the use-case. The deployment process also includes recompiling the model on the Jetson Nano to account for any hardware-specific considerations such as the GPU architecture.

The first iteration loop is about bringing the model to an acceptable object recognition performance such that it delivers results that can be worked with in industry practice. To attain this, the cycle of data preparation, model training, and model evaluation must be iterated several times as in any other machine learning task.

After having completed Step 5, we also revisited the identified use cases from Step 3, which represents the second feedback loop. With the computer vision model ready for deployment, we could now evaluate, extend, and refine the use cases based on the capabilities and limitations of the implemented system. This iterative feedback loop ensures that the use cases align closely with the capabilities of the deployed computer vision solution and maximizes the value and effectiveness of the application in remanufacturing processes. It also became apparent that the GPU of the Jetson Nano by its nature and computation power limits the choice of the CV algorithm as well as the complexity of the user interface and thus the use case.

By following this methodology, our project team aimed to develop a functional computer vision model. The iterative nature of the approach allows for continuous improvement and fine-tuning of the system, ensuring that the final solution meets the specific needs and requirements of operations. In going through this process and as we were confronted with a large variety of trade-off decisions to make, we saw the necessity to formulate functional and non-functional requirements for our computer vision system which will be examined in the next chapter.

## 3 Functional and non-functional requirements

In the context of the computer vision project for remanufacturing, defining both so-called functional and non-functional requirements is crucial. Functional requirements define what the software system should do and outline the specific functionalities it should provide. These requirements describe the desired behavior and capabilities of the software in terms of its features, operations, and interactions with users and other systems. Functional requirements are typically expressed as use cases and are in line with what we have previously defined as use cases in this project description. On the other hand, Non-functional requirements, also known as quality attributes or system qualities, specify the criteria that define the overall behavior, performance, and characteristics of the software system. These requirements focus on attributes such as reliability, performance, usability, security, maintainability, and scalability. Non-functional requirements are often related to the overall system behavior and user experience (Chung, 2012). By defining both types of requirements, the project team can align the development process with specific operational needs, guide system design, and implementation, and ultimately deliver a robust and effective computer vision solution.

### Functional Requirements

#### Real-Time Object Detection:

The system should be capable of detecting and recognizing 18 different LEGO parts of various colors, shapes, and sizes in real time from the camera feed.

#### High Precision Object Detection:

The system should prioritize achieving a low number of false positives to minimize mistakes. The detection algorithm should have a high precision rate.

#### Lighting Conditions:

The system should be designed to work effectively under regular or high lighting conditions commonly encountered during the remanufacturing process.

#### Intuitive User Interface Design:

The UI should be intuitive and user-friendly, providing immediate feedback on changes made to advanced settings. The live stream should be smooth and detected LEGO parts should be clearly visible via bounding boxes to avoid misunderstandings.

Detection Visualization:

The system should provide visual overlays, such as bounding boxes, on the camera feed to highlight the detected LEGO parts. This feature enhances user understanding and facilitates accurate identification of the parts.

Instruction Display:

The system should display step-by-step instructions for assembling or disassembling the LEGO set based on the detected parts. The instructions should be synchronized with the detected parts displayed in the video stream to ensure accurate guidance.

User Interaction and Customization:

The web app should allow users to initiate detection, view instructions, and perform necessary checks. Users should be able to customize settings, such as coloring and display of bounding boxes, as well as define detection confidence thresholds for a personalized experience.

Single User Support:

The system should support a single concurrent user accessing the UI and utilizing the detection and instruction display functionalities.

Communication:

Enable communication between the Flask web app server, the user interface, and the AI model. Detection results should be sent to the server via HTTP requests to exchange relevant information about the detected LEGO parts. Furthermore, the web app server should be able to send the user's visualization and detection settings as well as the necessary parts of each instruction step.

## Non-functional Requirements

Performance:

The system should exhibit low latency and real-time response for object detection and instruction display. This ensures timely and efficient detection results, minimizing any delays or lags during the remanufacturing process.

Reliability:

The system should achieve high accuracy in detecting LEGO parts, minimizing false positives and false negatives. This ensures that the correct components are identified, leading to reliable and high-quality remanufactured products. Additionally, there should be no discrepancies between the instructions and the detected parts by the Jetson nano visualized using bounding boxes.

Usability:

The web app should feature an intuitive user interface that allows technicians to navigate and

interact with the system easily. The instructions and visualizations should be presented clearly, with high contrast between labeled and non-labeled parts. The system should respond instantly to user settings, providing a seamless user experience and reducing the learning curve.

#### Smooth Video Streaming:

The live stream of the camera feed should be smooth and have minimal latency to provide a seamless user experience.

#### Portability:

The system should be deployable on the Jetson Nano device and not have significant hardware or software dependencies. This allows for easy setup and reduces potential compatibility issues.

#### Technology:

The system should utilize OpenCV for video streaming and processing, leveraging its capabilities for real-time object detection on the Jetson Nano.

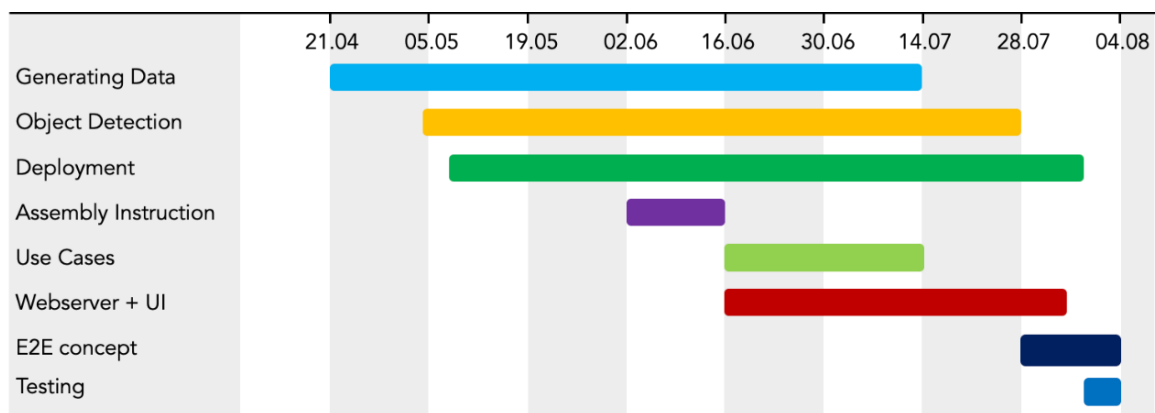
#### Maintainability:

The codebase should be well-structured, modular, and well-documented, making it easier to maintain, update, and extend in the future. This ensures the long-term sustainability of the system and facilitates future enhancements or bug fixes.

## 4 Project Plan and Effort Estimation

This chapter provides a detailed insight into effort estimation and project planning for the project. In the beginning, this chapter presents the project plan and a timeline, outlining the sequence of tasks. Afterwards an estimation of effort is given by listing and describing the required work packages.

The following Gantt chart provides a comprehensive overview of the project's organizational framework and outlines the roadmap for final execution. It offers a visual representation of the duration and timeline for each work package:



**Figure 2: Project Gantt Chart**

As observable in the diagram, the three most extensive work packages in the project were *Generating Data*, *Object Detection*, and *Deployment*. Thus, these tasks were almost consistently worked on throughout almost the entire project duration.

To assess the effort required for the entire project and allocate tasks effectively, the project tasks have been collected and structured into work packages. A work package represents a standalone, well-defined task that can be assigned clear responsibility. By breaking down the overall project into these building blocks, it has been divided into smaller, manageable parts, facilitating planning and effort estimation. The processed work packages include:

### Generating and labeling training images

This work package covers the generation of training and test data for the model. The process involves capturing real images of individual components and labeling the parts within the images. The aim is to create a diverse and representative dataset that covers various scenarios and objects the algorithm will encounter during its training phase.

To capture the images, the Lego piece was disassembled, and the individual parts were spread out on a work surface. Numerous pictures were taken, ensuring that all classes had sufficient representation in the dataset. The subsequent labeling of the training images involved manually annotating each image with bounding boxes and object classes with CVAT. The generated and labeled training images form the foundation for model training. Thus, this work package is crucial for the *Implementation of Object Detection* package and needed to be worked on before it, to possess sufficient training data for successful implementation.

Since this work package served as the foundation for the implementation of the application and the generation of training data was generally time-consuming, it was the first task worked on simultaneously by all project participants. It was in progress almost continuously from the beginning to the end of the project.

#### Implementation of object detection

This package aims to implement a functional object detection algorithm capable of recognizing and classifying Lego components. Object detection is a critical component of the software product as it ensures the accurate identification of the correct components in the application at a later stage. The initial steps involved selecting suitable frameworks to implement an appropriate algorithm. After evaluating several state-of-the-art models, the Ultralytics YOLOv8 model was chosen following several trials. This model was then customized and configured to meet the specific requirements of the project. The training data collected in the *Generating and Labeling Training Images* package was utilized to train the model. Additionally, data augmentation techniques were applied to the collected images to expand the dataset and enhance the model's robustness. The implementation process was iterative, requiring continuous adjustments to the training data and model.

This work package was closely linked to the *Generating and Labeling Training Images* package, as it relied on the generated data. It was also an integral part of the *Identifying Use Cases* package, as the implementation of object detection facilitated the definition of possible use cases. The implementation effort was manageable as the provided frameworks could be quickly integrated. However, completion of this work package depended on the availability of the training data from the previous package. The implementation process was ultimately an iterative one, continuously improved by incorporating new training data.

#### Assembly instruction

The assembly instruction work package entails the creation of a comprehensive user guide for the Lego set. The objective of this manual is to provide detailed instructions and guidance on the proper assembly of the components. The instruction holds significance for the final application as it will be presented to users during the set's assembly process. Utilizing the Studio 2.0 (BrickLink, 2023)



software, a Lego-style guide was generated, illustrating the placement of each component at every step. This work package could be handled independently as it was clearly delineated from other packages. The User Manual was only relevant in the final application. This work package was good manageable in terms of effort and was completed early in the project timeline.

### Deployment

In this work package, the installation and setup of the Jetson Nano, along with the necessary hardware components, were carried out. The goal was to make the Jetson Nano operational, enabling it to run the application locally in the subsequent stages of the project. The setup of the Jetson Nano involved installing packages such as Jetpack, Gstreamer, CUDA, and TensorRT. Additionally, a camera was connected, which would be used later in the application. Furthermore, the necessary Docker container configurations were implemented to ensure the program could run on the Jetson Nano. This work package was essential for being able to deploy the final software product on the device. The initialization of the Jetson Nano was particularly relevant to the *Implementation of Object Detection* and *Webserver* work packages, as the products from these work packages ultimately ran on the Jetson Nano. It was crucial that the necessary packages for object detection and the webserver also functioned on the Jetson Nano.

Overall, this work package was demanding due to various challenges encountered during the setup process. Conflicts with Python versions in the packages and camera integration significantly delayed the installation of the Jetson Nano.

### Identification of use cases

The *Identification of Use Cases* work package focuses on identifying and defining the various use cases for the software application. This process involves analyzing the requirements, needs, and goals of potential users. The objective is to understand how the application can effectively address specific problems or provide value in different scenarios. This work package serves as a crucial foundation for subsequent development and ensures that the software application aligns with the intended objectives and user requirements.

Initially, the technical capabilities were assessed, and potential issues that the software application aimed to resolve were identified. Based on the identified problems, use cases were defined and aligned with the technical possibilities. Subsequently, the use cases were prioritized, and an implementation sequence was established. Additionally, exceptions that might occur during runtime were also addressed. This work package depended on the output from the *Implementation of object detection* package since the use cases relied on the technical capabilities. The use cases were progressively expanded with improved model performance. Overall, this work package was very manageable in terms of effort.

### Webserver and UI

The User Interface and the communication base via HTTP play a pivotal role in our project, as they are the components that users directly interact with. This work package focuses on creating a robust and efficient communication channel between the trained model and the Flask web app server, while also crafting an intuitive and engaging user interface using HTML, CSS, and JavaScript.

Firstly, we invested effort in analyzing both the functional and non-functional requirements of the communication base and user interface. This step involves understanding the desired features, performance expectations, and necessary security measures to ensure a reliable and user-friendly experience. Next, we designed the architecture of the communication base, by defining endpoints, data formats, and API specifications for the HTTP requests. Concurrently, we developed the Flask web app server, setting up routes to handle various HTTP requests. This work package also included implementing endpoints to receive detection results from the model and provide visualization and instruction data to the web page. Moreover, it was necessary to program the HTTP requests, enabling the model to send detection results and receive instructions from the Flask web app server. This step involves integrating HTTP request libraries and skillfully handling data exchange. Overall, this work package required a high effort, as the implementation was time-consuming.

### Testing

While we tested our progress during the implementation manually and evaluated our computer vision model continuously, we decided to conduct a unit testing phase to ensure the reliability and accuracy of our application. Unit testing is a critical step in software development that allows us to verify the functionality of individual components against predefined test cases.

The testing focused on the communication between the different parts of our software architecture, as reliable communication is a crucial aspect to provide the user with a seamless experience. Our test suite covered various scenarios, including error handling for invalid requests and non-existent pages, algorithm validation with test images containing known objects, and data processing correctness. We also tested boundary cases, assessing how the application behaved with maximum and minimum values.

The unit testing phase provided valuable insights, helping us identify and fix bugs, optimize critical areas, and meet the specified requirements.

### E2E Operating concept and user guide

The E2E operating concept and user guide describe the complete workflow and usage instructions for the system, covering all aspects from its inception to its end use. It provides a comprehensive

understanding of how to set up the system, how to operate it, and how users can interact with it to achieve their desired outcomes. This work package was compiled at the end of the project, after the model and software were completed, and was in terms of effort manageable.

The following graph depicts the perceived complexity along with the perceived size of individual work packages. For this purpose, the involved team members scaled the complexity and size of tasks on a scale from 0 to 5, where 5 represents significant effort/complexity and 0 indicates no effort/complexity.

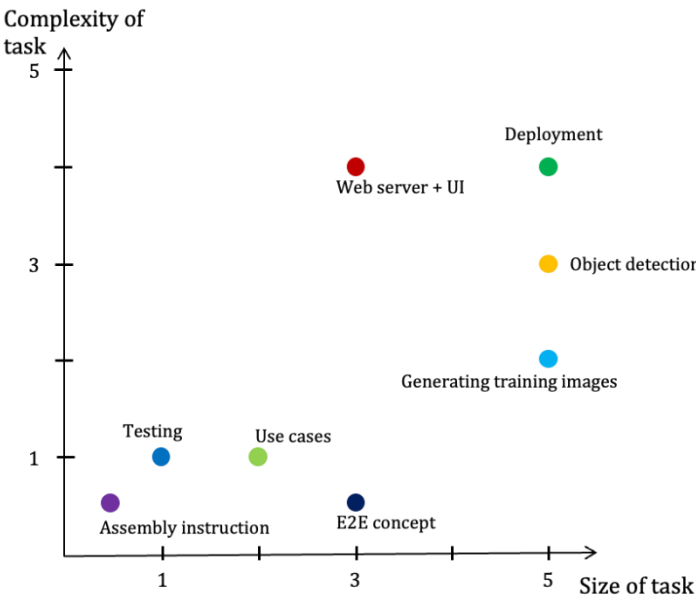


Figure 3: Tasks mapped by complexity and size

## 5 Hardware Specifications and Deployment

### NVIDIA Jetson Nano Developer Kit and CSI Camera

The NVIDIA Jetson Nano Developer Kit serves as the core hardware platform for our project. The Jetson Nano as edge-device is equipped with a Quad-core ARM A57 CPU and a 128-core Maxwell GPU with 4GB of memory, making it ideal for the development of lightweight computer vision applications. It enables robust visual computing, machine learning capabilities, and low-latency performance on a compact and energy-efficient platform.

Our application requires access to high-quality visual data, provided by a Camera Serial Interface (CSI) model camera. The specific model of the camera is based on the project's requirement and its compatibility with NVIDIA platforms, like the Jetson Nano. The CSI interface allows for high-speed data communication between the camera and the Jetson Nano, a critical requirement for real-time computer vision applications.

### Hardware Setup and Configuration

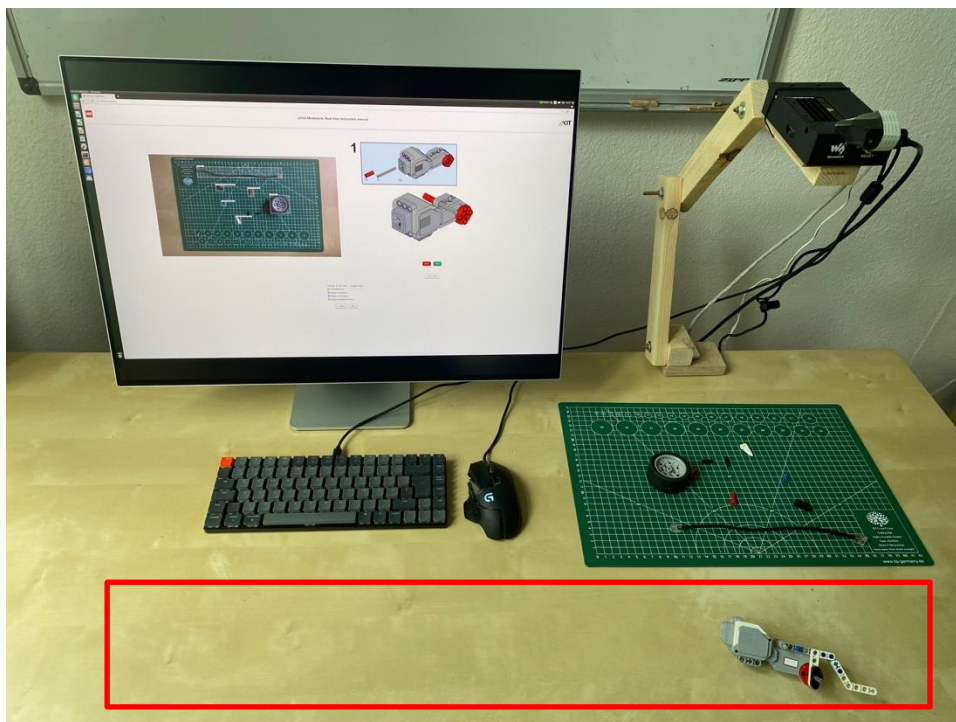
The NVIDIA Jetson Nano is setup by flashing an external SD card with the appropriate operating system provided by NVIDIA. System settings are configured, and all necessary peripherals are connected to establish a robust platform for development.

The Jetson Nano utilizes the NVIDIA Jetpack Software Development Kit (SDK) to facilitate the development process. The Jetpack SDK version 4.6.2 is bundled with a Linux Driver Package (L4T) and various tools that aid in AI software development and deployment. This version also includes a set of libraries and APIs that enable the Jetson Nano to be programmed for machine learning and computer vision applications. The SDK is installed as part of our development environment setup.

Our computer vision application relies on a real-time video feed from the camera. GStreamer enables the capture, processing, and output of this video feed in a streamlined manner. It is an open-source multimedia framework and provides a flexible and efficient pipeline for handling multimedia data. It allows for direct streaming of video from the camera to our application, where further processing of the data occurs. To interface with the camera, GStreamer's `nvarguscamerasrc` plugin is employed. This plugin is specifically designed for NVIDIA's Jetson series, ensuring seamless integration and high-quality video data acquisition.

## Physical Installation and Environment Setup

The setup of the physical installation and environment is a critical component of our project, as it directly affects the efficiency and effectiveness of our assembly assistance system. As shown in Figure 4, the finalized setup of the workplace exhibits a compact and efficient design. Central to this configuration is the Nvidia Jetson. For the sake of flexibility and convenience during prototyping, we have mounted the Jetson on a stand that allows for continuous height adjustment. This design choice enables us to experiment with different camera distances, thus finding a good balance between field-of-view and object recognition performance.



**Figure 4: Setup of physical installation**

One of the defining aspects of our setup is the orientation of the camera. In our design, the camera is affixed directly to the Jetson and points downwards towards the working area. We chose this configuration mainly due to the short cable connection between the Jetson module and the CSI camera.

The work surface features a green mat where the individual Lego Mindstorm parts are positioned. The distinctive color of the mat was chosen to enhance contrast, thereby aiding the computer vision system in more accurately identifying Lego pieces. Grid lines provide a visual reference that helps in differentiating between parts of similar appearance but varying lengths.

Next to the working area in which the parts are detected, there is a dedicated assembly area (see Figure 11 Figure 4 marked in red) in order to avoid detecting the same parts twice while conducting the assembly once the parts are removed from the working area.

The setup is complemented by an interface that displays the camera's video stream, including annotations. The display provides a real-time visual aid that allows the user to monitor the detection process. Additionally, the interface also contains information about the current assembly step, so that the user is clearly guided throughout the assembly process.

## Deployment

Our deployment process employs Docker, which enables a consistent environment for software to run by encapsulating all the necessary components within a container. This ensures our application's smooth operation and simplifies the deployment procedure, as the required packages and dependencies are bundled together.

The container holds a range of critical packages, including OpenCV, Torchvision, and Torch, among others, required for our project. We use the prebuilt jetson-inference container, an open-source deep learning package tailored for NVIDIA Jetson, developed by *dusty-nv*. This container brings several benefits including pre-compiled libraries that are optimized for the Jetson platform, saving us considerable time and potential challenges of manual installation and configuration.

We retrieve the Docker container image, specifically built for L4T Version R32.7.1, compatible with JetPack 4.6.2, from DockerHub. This image is based on the *l4t-pytorch* container.

Upon starting the jetson-inference container, we mount our custom repository to ensure it remains accessible within the container. While the prebuilt image incorporates most necessary packages, it lacks the *onnxruntime-gpu*. As this package is essential for running our YOLOv8 model, we proceed with its manual installation.

## 6 Software Architecture and Workflow

Our project aims to enhance the Lego building experience by implementing a LEGO part detection system. With the main goal of implementing a lightweight solution that can run on the Jetson Nano, we have developed a solution that follows the Model-View-Controller (MVC) design pattern. This approach ensures efficient communication, modular development, and a seamless user experience. By adhering to the MVC design pattern, we achieve a modular and scalable architecture. Each component has a distinct role and responsibility, enabling independent development and maintenance. It promotes code reusability, extensibility, and flexibility for future enhancements or modifications (Pop & Altar, 2014).

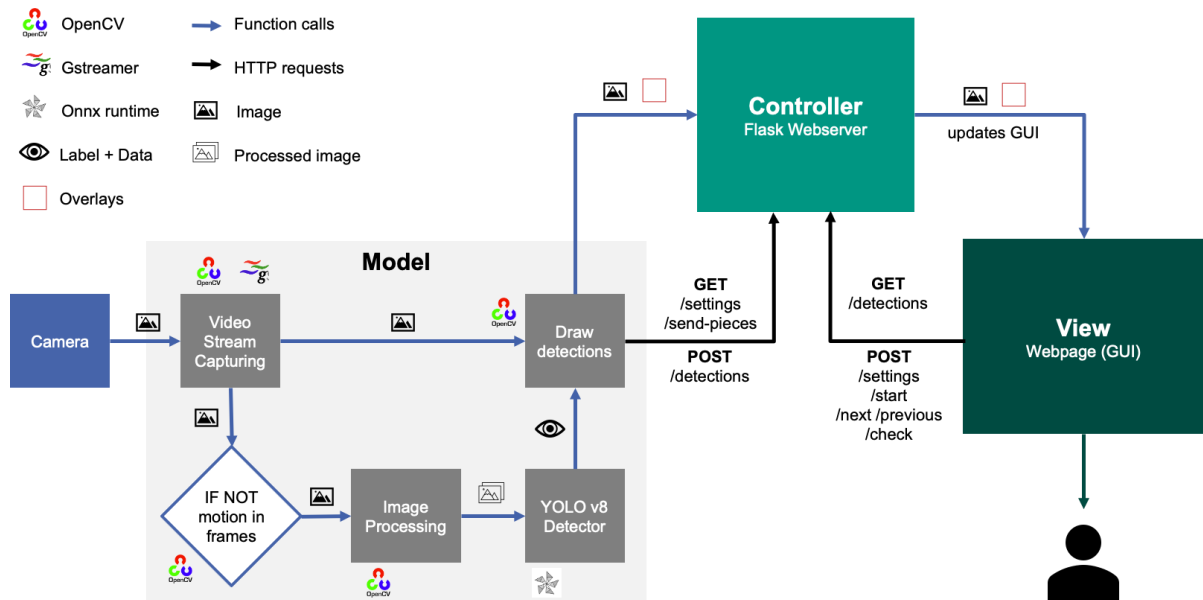


Figure 5: Model View Controller (MVC) of the solution

### Model

Following the MVC design pattern, we achieve a clear separation of concerns and enhance the overall system architecture. The AI model is part of the *Model* component, responsible for the LEGO part detection process. By separating the detection logic from the rest of the architecture, we achieve optimized performance and scalability. This allows for real-time analysis and faster processing of the camera stream.

Key Responsibilities of the *Model*:

**Video Stream Capturing:** For capturing the video stream, we deploy the gstreamer pipeline in combination with OpenCV. The system captures 4K resolution images, providing high-definition input for the subsequent processing stages.

**Motion Detector:** The captured image is passed to the object detector only if no motion is detected. This strategy conserves the limited computational resources of the Jetson Nano. When Lego parts are moved around, no detections are made, leading to a smoother video stream presentation in the View component.

**Image Preprocessing:** To ensure compatibility with the YOLO model, the images undergo preprocessing. They are resized to a uniform resolution of 640x640, providing a standardized input for the subsequent inference stage.

**Inference using ONNX Runtime-GPU:** The actual object detection process is performed by a YOLOv8 model. After being trained, this model is exported in ONNX format and runs on ONNX Runtime-GPU.

**Drawing Detections:** This function draws the required overlays on the original 4K images and passes it to the *Controller*. It receives the detected bounding boxes, classes, and scores from the YOLOv8 model. To generate the appropriate overlays, a GET request is issued, that retrieves relevant data, namely the components to be emphasized in the current phase and the user-defined preferences (bounding box color, labels, and confidence thresholds for visualizing a bounding box, among others).



## View

In the Model-View-Controller (MVC) architectural pattern of our project, the *View* is responsible for presenting the user interface and rendering the visual components that users interact with. Built using HTML, CSS, and JavaScript, its primary role is to create a user-friendly and visually appealing interface.

### Key Responsibilities of the *View*:

**User Interface (UI) Elements:** The *View* defines and styles various UI elements such as buttons, images, video feeds, and containers to present the application's functionalities and data in a visually cohesive manner.

**Dynamic Content Rendering:** Based on data received from the Controller, such as the instruction images and the live camera feed, the *View* is responsible for rendering dynamic content. It interacts with the Controller to request data and updates the UI elements accordingly.

**Button Functionality:** User interactions are handled by buttons defining first the instruction mode and later switching between instruction steps and checking detections. When users click these buttons, event listeners attached to them invoke corresponding functions in the Controller through asynchronous fetch requests. The Controller processes these requests and provides responses that the *View* uses to update the UI.

**Settings Management:** The *View* facilitates the presentation and adjustment of advanced settings, such as coloring options, confidence thresholds, and display preferences. It includes functionalities to show or hide the advanced settings based on user interactions and allows users to confirm and update their preferences through fetch requests to the Controller.

**Feedback Display:** Visual feedback is shown to users, such as displaying messages indicating whether necessary pieces are found during assembly or correctly disassembled during disassembly. It receives feedback messages from the Controller through asynchronous fetch requests.

**Layout and Styling:** Layout and styling of the webpage is defined using CSS styles. It ensures consistency in the visual presentation, color schemes, fonts, and spacing, providing an aesthetically pleasing user experience.

**Global Variables Management:** The *View* may define and manage global variables specific to the frontend, ensuring data integrity during interactions and updates.

## Controller

The *Controller* serves as the central component responsible for handling and coordinating user interactions, data flow, and application logic. In our project, the controller is implemented using Flask, a lightweight and versatile web framework for Python, playing a pivotal role in maintaining the state of the application and managing the interactions between the user interface, the YOLO model, and the data. It efficiently handles HTTP predefined requests from the frontend, processes them, and communicates with other components to ensure a seamless user experience.

### The key functionalities of the *Controller*:

**Initialization and State Management:** The *Controller* initializes critical global variables. These variables are instrumental in maintaining the current state of the application, such as the user-selected mode (assembly or disassembly) and the current instruction step.

**Routing and Page Handling:** The *Controller* defines various routes (URL endpoints) to map HTTP requests to specific functions. For instance, the /settings route handles both GET and POST requests to retrieve and update user settings. Similarly, the /start route facilitates mode selection and initiates the first instruction based on user preferences.

**Instruction Management:** The *Controller* manages the progression of instruction steps, enabling users to navigate between steps using the /next and /previous routes. It ensures that the user stays within the valid step range, preventing navigation beyond the defined instruction boundaries.

**Necessary Piece Management:** Through the /send-pieces route, the controller receives the necessary pieces for the current instruction step and stores them in the necessary\_pieces global variable. This functionality allows the application to track the required components for successful completion.

**Handling Detection Results:** The Controller handles the detection results received from the YOLO model through the /detections route. It validates the data, stores the results in the detection\_results variable, and provides these results upon request through the /detections GET route.

**Label Handling and Validation:** By processing data through the /labels route, the *Controller* evaluates the detected parts against the necessary pieces for the current step. It determines if all essential components are identified correctly and generates appropriate messages for the user based on the results.

**Error Handling:** The *Controller* incorporates an error handler to deal with 404 page not found errors, ensuring a graceful response in case users access invalid URLs.

## Unit Tests

We focused our unit tests on the HTTP requests in our object detection application for several reasons. First and foremost, the HTTP requests are the primary means of communication between the client-side and server-side. Ensuring that these requests work correctly is crucial for the overall functionality of the application and user experience. As the *Controller* is the brain of our architecture, we decided to focus our software testing endeavors on HTTP requests which are all defined and handled by the Flask Webserver. Therefore, we utilized the Flask testing framework, which facilitated the creation of mock HTTP clients to interact with our application. This framework enabled us to test HTTP request handling, object detection algorithms, data processing, and response generation. By writing unit tests for the HTTP requests, we aim to verify that each endpoint responds as expected and returns the appropriate status codes and data formats. These tests cover a wide range of scenarios, such as successful requests, error handling, and edge cases, which helps identify potential bugs or issues in the application's core functionalities.

While unit tests are invaluable for validating the correctness of specific functions and components within the application, they might not be the most suitable approach for evaluating the performance of the YOLO model used for object detection. The YOLO model's performance is best assessed using other evaluation techniques, such as precision, recall, F1 score, and mean average precision. These metrics provide a comprehensive understanding of the model's accuracy, ability to detect objects, and its performance on different classes. Furthermore, validating the YOLO model often requires a large and diverse dataset with ground truth annotations. Therefore, we will tackle the evaluation of our trained and used YOLO model in an entire later chapter.

## 7 Use Cases & Design Principles

### Minimum viable use case

With this setup, we formulated a minimum viable use case which we aimed for at the start. That is that the system should recognize and provide input to the assembler on which part should be assembled next given a certain set of parts that is visible in the parts area. The exact procedure for that use case will be described in the following. First, an initialization is conducted by checking if parts are recognized at all. In case no parts are recognized at all the system provides the user with feedback to adjust either the camera settings, lighting conditions, or the field of view of the camera or to make sure that all parts are fully disassembled. Now the collection of the required parts can be started. While the system is recognizing the parts for the first assembly step in the working area it shows a dialogue that provides the user with the information that it is still searching for the required parts as well as a depiction of which parts it is searching for and the required number. In case the number of parts in the parts area equals the number of parts required for this step from the assembly instruction the user receives feedback that all necessary parts were identified correctly and that he or she can put them into the assembly area and start the assembly. If that is not the case the user will receive feedback that not all parts were not recognized correctly, and he or she should check if all available parts are within the detection zone. From that point on the assembly quality is fully dependent on the user. After having placed the parts in the assembly area, the assembly instruction is shown to the user on the right-hand side of the screen and the parts are assembled. The user confirms a completed assembly manually by pressing a button on the screen which then sets the system up for the next assembly step.

### Additional use cases

In further use cases, before the initialization, the user might first be able to select whether he or she wants to assemble or disassemble the given parts on the interface. Afterward, an initialization test starts to check if all required parts are available to perform the next step as well as the previous sanity check for the camera settings. For that, the number of parts in the working area is counted and thereby the state of the assembly process is recognized as each assembly step is unique which allows its determination only by counting parts. However, a crucial prerequisite for that is that the system can recognize all parts in the working area at once. Therefore, the functionality of additive use cases depends on the performance of the network and can be further expanded as well as the customer experience enhanced.

But not only the choice of disassembly or assembly can be added. The simultaneous recognition of all available parts in the working area allows for optimized exception handling. While the minimum

viable use case only provides the user with feedback on whether certain parts are recognized, a more sophisticated use case could tell the user which parts are missing and in the next step even show overlays in the working area for those parts that are missing. The user can then quickly find the missing parts, move them to the working area and complete the assembly with the correct instructions on the screen. In case there is more than one of the same parts left in the working area and the user only required one of these in the assembly step, this, however, bears the risk that the user automatically would pick up all parts with overlays and end up with too many parts in the assembly area. This can be avoided by a further improvement, that provides the user with feedback that he or she only needs one of those which are currently shown with overlay. Thus, potential errors can be avoided and throughput time reduced. Another way to implement this would be to feed the logic in the web application such that the system already knows how many more parts the user would need from the working area. Therefore, the parts removed from the working area have to be compared with the required number of parts from the assembly area to come up with the number of additional parts required.

Thinking ahead, one could also make the application's use more appealing by adding gamification features like a leaderboard with regards to the components assembled with the least faulty assembly as well as reasonable metrics like the tact time and required time to finish the current assembly step (Sailer, 2016).

## 8 Model

This chapter deals with the underlying object detection model and discusses the process of data generation (image capturing and labeling) as well as training and testing the model.

For the implementation of the object detection, we used YOLOv8, also known as You Only Look Once, which was firstly presented by Redmon et al (Redmon, 2016). This is a state-of-the-art object detection model that has gained significant attention in the computer vision community. The model's single-stage detection approach allows it to simultaneously predict bounding boxes and class probabilities directly from the input image in one pass, making it highly efficient and suitable for real-time applications. With its ability to detect and localize objects across different scales and categories, YOLOv8 continues to be a popular choice for various computer vision tasks.

The following plot depicts the comparison with older Yolo versions, it was published by Jocher et al (Jocher, 2023). For this purpose, both the number of parameters as well as the latency time were measured. The results indicate that the latest YOLOv8 outperforms older models on the Coco dataset. YOLOv8 incorporates advancements in architecture and training techniques, enabling it to achieve higher accuracy while requiring fewer parameters.

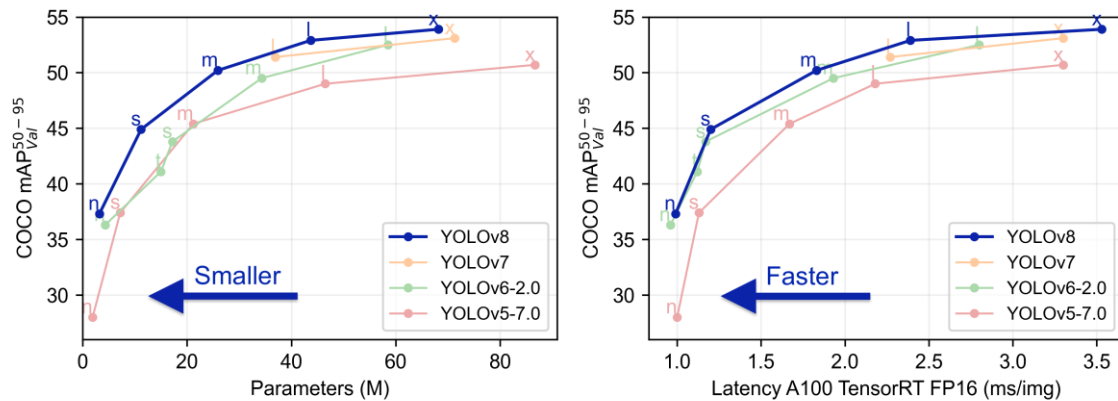
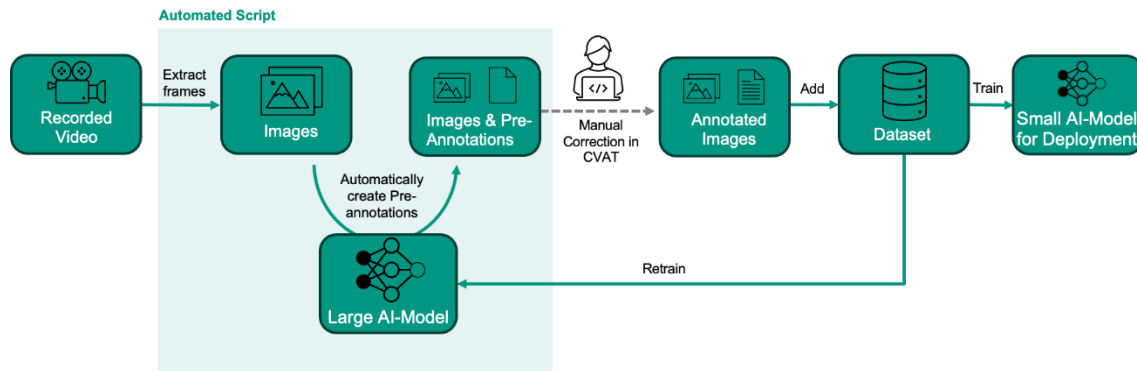


Figure 6: Yolo Benchmarks

Overall, YOLOv8 strikes a good balance between accuracy and speed, making it our preferred model of choice for this work.

## Dataset Generation

To utilize YOLOv8 for our specific use case, we had to generate a sufficient amount of training data. As our goal was to have several hundred instances for each class, we needed a process that could deliver this volume of data in manageable time. To achieve this, we established a pipeline that generates the training data from recorded videos. The pipeline is depicted in the following figure:



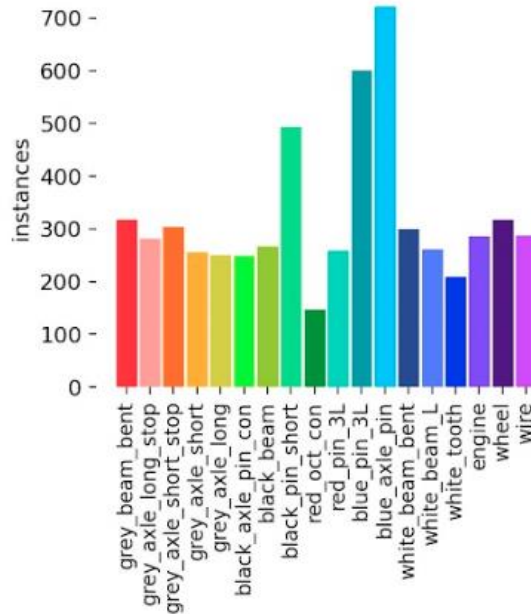
**Figure 7: CV Model Pipeline**

Firstly, videos were recorded capturing all sides of each object. From these videos, individual frames were extracted, which ultimately served as the training data. To expedite the image labeling process, we utilized a pre-trained YOLOv8M model to pre-label the individual frames. Subsequently, the images were partially labeled, but required manual cross-checking and correction. After manually verifying and correcting the labels in CVAT, the images were added to the training dataset. The dataset was then used to train a YOLOv8S model, which was intended to run on the Jetson Nano. Additionally, the data was also used to retrain YOLOv8M to enhance the quality of the pre-annotations.

## Training

We worked with a dataset comprising 18 different classes, totaling approximately 2000 images. Each class had around 300 images. To train the model, we fine-tuned a pretrained one, optimizing its performance for our specific task.

During training, we employed a variety of randomized augmentations, such as mosaic, flip, and rotation, to enhance the model's ability to generalize. These augmentations played a crucial role in preventing overfitting and boosting the model's robustness. The training process spanned 100 epochs and took approximately half an hour, thanks to the Nvidia V100 we utilized. The following graph shows the instances of each class.



**Figure 8: Size of the Training Dataset**

## Evaluation

In this section, the object detection model's performance will be assessed and examined. Initially, the confusion matrix for the model will be presented to evaluate its proficiency in recognizing specific object classes. Afterward, the precision, recall, and F1 Score of the model will be further assessed. Through the analysis of these metrics, a comprehensive understanding of the model's capabilities and limitations concerning the recognition of individual classes and its overall classification accuracy is acquired.

### CONFUSION MATRIX

The confusion matrix is a visual representation of the model's performance. It is a square matrix with the x-axis labeled as "True" and the y-axis labeled as "Predicted." The rows in the matrix correspond to the true class labels of the data, while the columns represent the predicted class labels made by the model. Each cell in the matrix contains the normalized count of data points that fall into a specific combination of true and predicted classes. The confusion matrix for our model is presented by following figure:



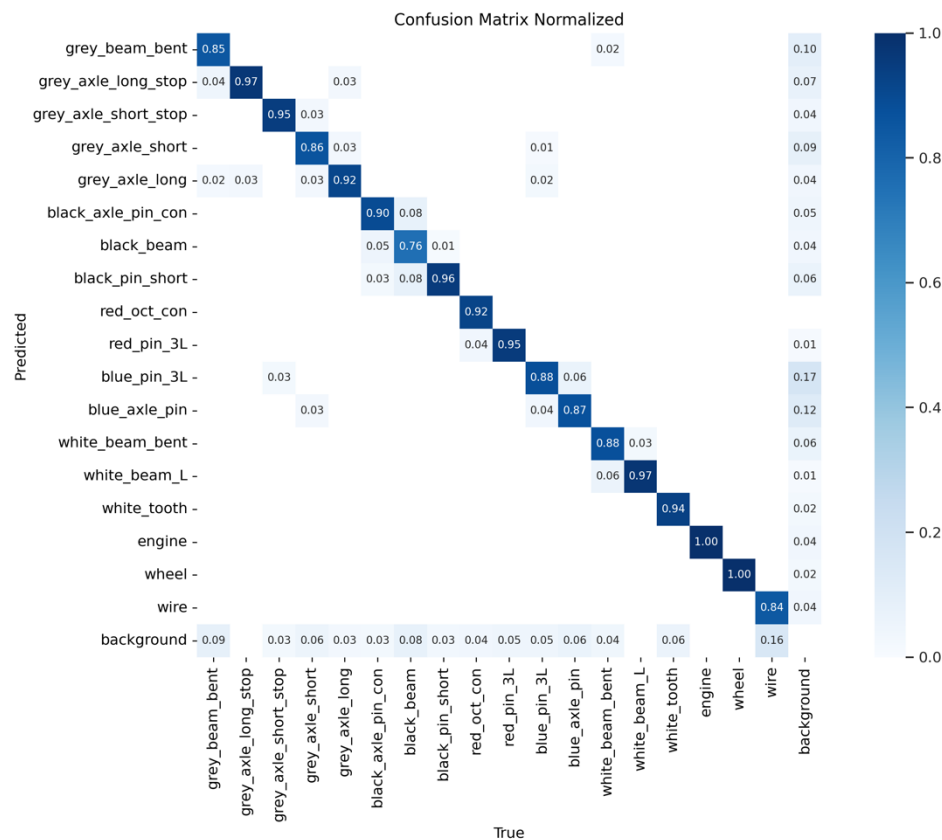


Figure 9: Confusion Matrix

The displayed figure presents the analysis of the confusion matrix, providing insights into the part detection performance. The model demonstrates accurate recognition for most parts, particularly larger components like the wheel and the engine, which consistently receive correct identifications. Out of the 18 parts, 11 are detected with a confidence level exceeding 90%, indicating high true positive rates. However, there are 7 parts for which the detection confidence falls below 90%, resulting in a relatively higher false positive rate.

One notable observation is that the Black Beam exhibits the lowest detection rate at 76%, significantly lower than all other parts, suggesting challenges in correctly identifying this specific component.

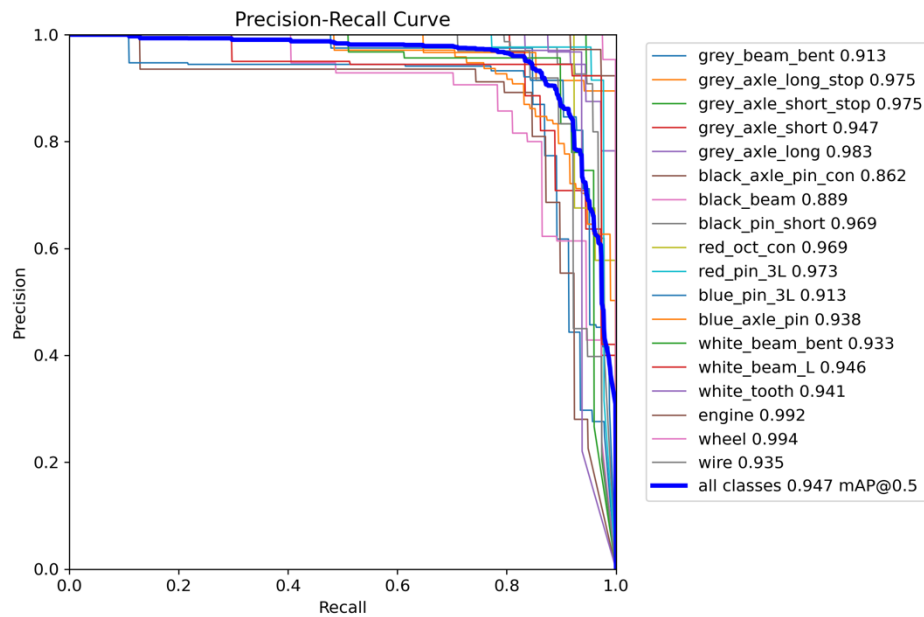
Furthermore, the analysis reveals a trend where parts with similar colors are more prone to confusion. For instance, the grey axle long part is frequently misclassified with other gray parts due to their similar color and shape, making differentiation difficult. This pattern extends to the three black parts: Black axle pin con, black beam, and black pin short, which are also frequently mistaken for each other due to their comparable shapes and colors. Similar confusion patterns are observed with parts of different colors as well; for instance, red, blue, and white parts occasionally get

mistaken for parts of the same color. Additionally, the two blue parts show a higher tendency to be misclassified as part of the background compared to other parts.

In conclusion, the confusion matrix demonstrates the model's strong classification capabilities, achieving impressive results for most parts, while also highlighting the challenges posed by parts with similar colors and shapes.

### PRECISION, RECALL AND F1-SCORE

In the precision-recall curve, the relationship between precision and recall is visualized across different probability thresholds used for classification. Precision represents the proportion of correctly identified positive instances among all predicted positive instances, while recall measures the model's ability to correctly identify positive instances among all actual positive instances in the dataset. In a perfect PR curve, precision remains 100% even as recall approaches its maximum value of 100%. The curve's trajectory showcases the model's trade-offs between precision and recall, assisting in identifying the optimal threshold that aligns with the specific requirements of the application. Following graph presents the PR-curve for our model:



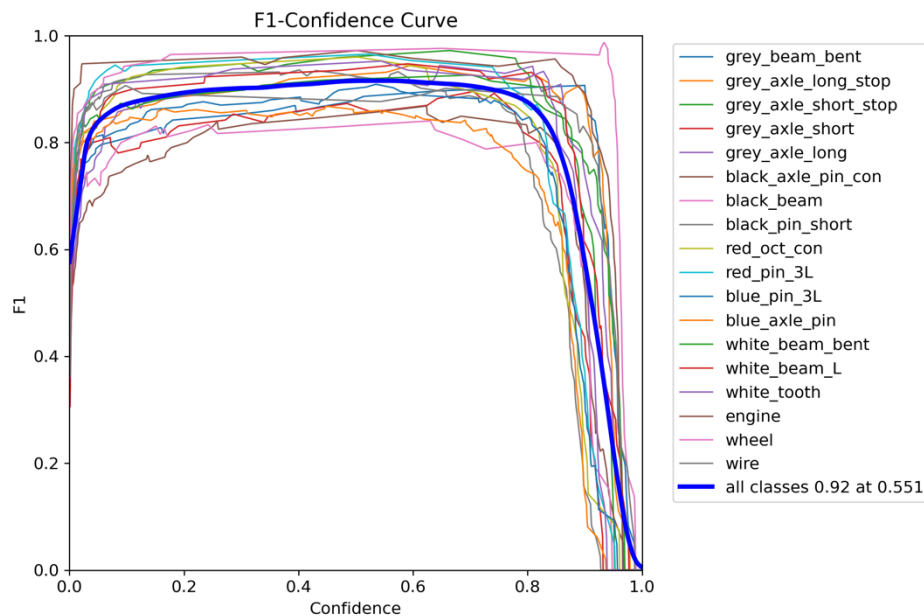
**Figure 10: Precision-Recall Curve**

The PR curve for all classes reveals promising results. The curve exhibits a slow decline followed by a steep drop, indicating good performance by the model. Like the findings in the confusion matrix, the PR curve also highlights poorer performance for the black parts. In particular, the black beam exhibits a trajectory significantly below the other curves. The gray parts also appear below the curve, reinforcing the impression that these parts are more challenging to detect. Additionally,

the curve demonstrates that larger components such as the engine or wheel are consistently and reliably recognized by the model.

In the F1 curve, the F1-score is plotted against different probability thresholds used for classification. The F1-Score is a harmonic mean of precision and recall, providing a balanced evaluation metric that considers both false positives and false negatives. The curve's shape is influenced by how the model balances precision and recall at various thresholds. Evaluating the F1 curve aids in selecting the most suitable threshold. A well-performing model will have an F1-Confidence Curve that exhibits a smooth, upward trend, indicating that as the confidence threshold increases, the F1 score also increases. This suggests that the model maintains a good balance between precision and recall across various confidence levels. On the other hand, a suboptimal model may show fluctuations or dips in the F1-Confidence Curve, indicating instability in its performance. Such fluctuations might occur when the model is overly sensitive to changes in the confidence threshold, leading to inconsistent precision and recall values.

The F1-Confidence curve is presented by the following graph:



**Figure 11: F1-Confidence Curve**

The F1-Confidence Curve for this model in Figure 8 demonstrates that many classes exhibit a high F1 score even at higher confidence thresholds. This is particularly true for the larger parts, which the model can classify easily. When averaged across all classes, the F1 score is approximately 0.92 at a threshold of around 0.5, indicating a good performance. However, it is also evident that the smaller black and gray parts fall significantly below the curve for all classes. The curves for these parts show higher fluctuations, indicating more inconsistent precision and recall values.

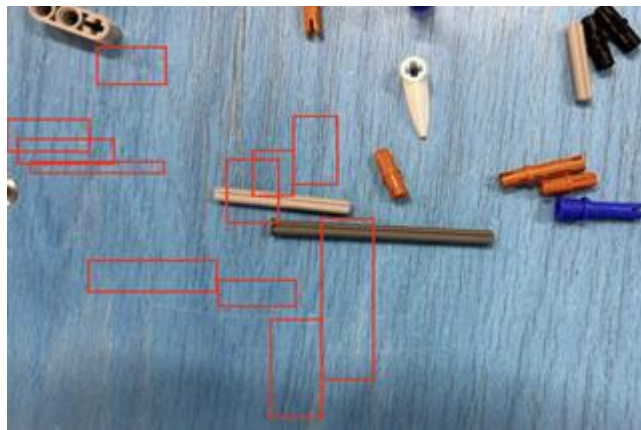
Overall, the F1-Confidence Curve indicates that the model performs well for most classes, even at a higher confidence threshold. The F1 score at the selected threshold suggests a strong balance between precision and recall. Nonetheless, it highlights the model's struggles with smaller black and gray parts, leading to lower F1 scores and less consistent precision and recall values in those cases.

## 9 Challenges

### Annotation Process Challenges

When it comes to the process of capturing and annotating images, we encountered challenges of different kinds on the way and went through a learning process.

Initially, we captured the training images using different phone cameras, which resulted in varying resolutions, image quality, and different formats. The used phones stored meta information about the EXIF orientation, which was not recognized by the used labeling tool LabelImg. Resulting in rotation mismatches of images and annotations (see Figure 12). The problem could be resolved by deleting the EXIF orientation tag from the images' metadata. This enabled proper interpretation of image orientation during subsequent annotation and processing steps in LabelImg and the used Python libraries.



**Figure 12: Flawed EXIF Orientation**

To avoid a distributional shift between train and test data, we opted to capture all images with the Jetson Nano's CSI camera instead of different phone cameras, since the inference was ultimately intended to take place on the Jetson Nano. After we overcame initially setup challenges of the Jetson Nano and its camera, we could do so and ensure consistency across the data and avoid issues arising from the use of different cameras during training and inference.

After resolving the issues related to image capturing and rotation, early model training revealed that the number of labeled images was insufficient, and a significantly larger amount had to be collected and labeled. Since LabelImg was not compatible with the Apple M1 chip and half of our group could not participate in the labelling process, we had to change to another labeling tool without any compatibility problems. Furthermore, our annotation process involved a lot of manual task assignment as well as uploading and downloading images and annotations on Google Drive, which turned out to be error-prone and cumbersome.

To streamline the labeling process and improve efficiency, we transitioned to use CVAT. CVAT is a web-based computer vision annotation tool that offers a more efficient and centralized approach to labeling. This switch allowed all team members, including those using the Apple M1 chip, to actively participate in the labeling process, enhancing collaboration and productivity. Furthermore, images can be directly uploaded in CVAT, eliminating multiple up- and downloads and the use of Google Drive. Furthermore, tasks can be directly assigned to team members in CVAT, allowing better coordination in the labeling process. In total, the adoption of CVAT as a unified labeling tool proved to be an effective solution in resolving the faced issued and maintaining smooth progress in the labeling process.

After early model training, during the evaluation of the results, particularly when analyzing the Confusion Matrix, we observed that some visually dissimilar classes with similar names were poorly recognized. This led us to suspect that certain images had been incorrectly labeled. Our suspicion turned out to be true, and as we corrected the annotations, we brainstormed ways to reduce human-made errors in the labeling process. As a result, we decided to change our naming conventions to more descriptive names and created a labeling guide to support us and ensure a more consistent labeling process. The labeling guide outlined specific guidelines for various aspects of the labeling process. One important aspect was the modification of bounding box sizes and determining when objects should no longer be labeled. For instance, objects that were heavily obscured or even unidentifiable by a human annotator would not be labeled. The used naming conventions and labeling guide are depicted in the appendix.

In the further course of our work, we noticed that our model's performance in recognizing the objects was still not satisfactory yet. Therefore, we considered how we could simplify the task for our model and introduce an inductive bias by using a fixed setting.

Initially, we captured images from a long distance, mostly including all the parts to be recognized. As a result, we reduced the distance between the Jetson Nano and the parts to be recognized and fixed it at a specific distance from the surface by building a wooden device, which can be found in chapter 0. As a logical consequence of the reduced distance, we also had to reduce the number of objects in the images to be labeled, resulting in 2-3 objects per image that are visually similar and could be more challenging to distinguish from each other compared to others. This also accelerated our annotation process, as fewer classes were present in the images.

To further facilitate our model's task, we consistently used the same surface and opted for a cutting mat with grid lines to assist our model in recognizing distances and sizes. The grid lines are particularly intended as a reference point to help distinguishing between pieces which only differ in length and size. To maintain consistent lighting conditions, we utilized a standing lamp with white light, positioned from above to illuminate the parts being recognized. Particularly with the

black parts that have similar visual characteristics, varying lighting conditions and resulting shadows can significantly hinder the recognition process.

These measures resulted in a significant improvement; however, it became apparent that the detection rate suffered greatly when any changes were made to the setting. For instance, when the lamp was turned off, the recognition performance noticeably decreased. Since the goal is to develop a robust model, we made the decision to slightly vary the light settings and capture additional images for further annotation. By additionally adding more images, including the initial annotated images with a different setting, without mat and lamp, we were able to achieve higher robustness.

In summary, our image capture annotation process has evolved from a disorganized, error-prone pipeline involving a lot of human effort to a more streamlined process of automatically extracting frames from a video captured by the Nano's CSI camera, pre-annotating them with a larger YOLO model, and correcting annotations in CVAT using an efficient upload and task assignment system.

## Deployment Challenges

Also, during the deployment of our model on the Jetson Nano, we faced different challenges due to given hardware limitations and software compatibility issues which will be discussed in the following part.

As previously pointed out, we chose a YOLO V8 model for the object detection, since the YOLO (You Only Look Once) architecture has emerged as state-of-the-art in this task, and the V8 is the latest available version. For convenience and simplicity, we leveraged the Ultralytics Python framework to train it and set it up on premise. During the deployment of our model on the Jetson Nano, we ran into different problems. Since the used Ultralytics framework is only compatible with Python versions greater than Python 3.8, we could not get it running on the given Jetson Nano, which hardware only allows for Python 3.6.

To address the compatibility issues, we explored three different options. First, we considered adjusting the dependencies of the Ultralytics framework to make it compatible with Python 3.6 on the Jetson Nano. However, due to the complexities of patches and modifications, we deemed this option unsuitable and dismissed it as we were exploring more robust ways of deployment. Alternatively, an older Yolo model which is compatible could be used. However, after comparing given benchmarks comparing the latest compatible model, YoloV5, and the latest available Yolo V8 model, we observed that using the older model would negatively impact our system's performance. Consequently, we opted against this approach.

The most viable approach appeared to be using a container specifically tailored to the Jetson Nano and its hardware to bypass compatibility issues. Nvidia, the manufacturer, provides a container for

inference on the Jetson Nano, which includes several pre-trained models for both training and inference for different computer vision tasks. Unfortunately, the most suitable model available is an SSD MobileNetV2, which did not perform as well as the initially targeted YoloV8 model in benchmarks. Nonetheless, we initially chose to utilize this model along with the provided functions for training and inference. However, we observed very poor performance, which prompted us to explore further alternatives.

Through our research, we discovered the ability to export models in the ONNX format. The ONNX format is a standardized representation of machine learning models that enables interoperability between different frameworks. Initially, we faced some compatibility issues when attempting to install the ONNX Runtime within the container. However, with perseverance, we successfully installed it. This allowed us to train our YOLOV8 model on a Virtual Machine equipped with an A100 GPU and then export it in the ONNX format to use it in the container on the Jetson Nano for inference purposes.

After solving the above problems and deploying our model in the container, we observed a significant latency in the inference process. The delay between camera movements and detections was too high, preventing a smooth and seamless execution of the use case. After further analysis, we found out that this delay stemmed from the model's detection as well as from the HTTP transmission. To address this problem, we opted to only run detections on every fifth frame and turn down the camera's resolution to 1280x720 (lowest possible resolution) to overcome the bottlenecks and speed up the transmission. It should be noted that those values are hyperparameter, which can easily be adjusted.

To further improve inference speed, we integrated a motion detection mechanism into the camera's real-time feed. This detector functions by comparing two consecutive frames, both of which are converted to greyscale images. If the pixel differences between these frames exceed a predetermined threshold, motion is detected. By only sending the image for inference when there is no motion, the frequency of detections is reduced. This refines the overall user experience by providing a smoother video stream in the associated web application due to less processing overhead.

The effectiveness of the motion detector depends on several hyperparameters. These include the threshold for detecting differences between two pixels, the frequency at which images are compared (e.g., every frame, every alternate frame), and the percentage of different pixels required to confirm motion. Under stable lighting conditions, this system enhances the user experience by providing more seamless video playback. It's worth noting, however, that changes in lighting or increased noise due to high ISO settings can reduce its performance. For this reason, it's advised to use this feature with caution in a fluctuating environment.



# 10 Operating Instructions

## End-2-End Operating Concept

This guide provides instructions for the initial setup and regular use of the Object Detection application. A more detailed description can be found in the repository. It is meant to be utilized with a Jetson-Nano developer kit running on SDK 4.6.1. Nonetheless, it can be adapted for compatibility with other SDK Versions as needed.

### 1. Initial Setup

These steps need to be carried out only once to set up the project:

#### 1.1 Clone the object-detection-project repository.

Command to clone the repository:

```
git clone https://git.scc.kit.edu/aiss-applications-in-computer-vision/object-detection-project
```

#### 1.2 Download the pre-built Docker image.

For Jetpack 4.6.1, use this version. More details on other versions can be found on the Jetson-Inference GitHub page.

Command to download the image:

```
docker pull dustynv/jetson-inference:r32.7.1
```

#### 1.3 Navigate into the object-detection-project directory and download the correct version of onnxruntime-gpu.

More details on other versions can be found on the Jetson Zoo website.

Command to navigate and download:

```
cd path/to/object-detection-project/  
wget https://nvidia.box.com/shared/static/jy7nqva7l88mq9i8bw3g3sklf4kccn2.whl -O onnxruntime\_gpu-1.10.0-cp36-cp36m-linux\_aarch64.whl
```

## 2. Regular Use

Perform these steps every time you run the application:

### 2.1 Navigate to the jetson-inference directory, run the container, and mount the object-detection-project repository into it.

Commands to navigate and run:

```
cd path/to/object-detection-project/  
docker/run.sh
```

### 2.2 Install onnxruntime-gpu.

Command to install:

```
pip3 install downloads/onnxruntime_gpu-1.10.0-cp36-cp36m-linux_aarch64.whl
```

### 2.3 Run the application.

Command to run the application:

```
cd inference/  
  
python3 app.py --use_camera_stream
```

## User Manual

This guide provides instruction on how to use the *LEGO Mindstorm: Real-Time Instruction Manual* webapp. Next to explanation of the two different modes *Assembly* and *Disassembly*, it provides information on possible settings.

## GETTING STARTED

### 1. Initial Setup

- Make sure you have followed the End-2-End Operating Concept.
- Run the following command: `python3 app.py`

### 2. Mode Selection

- Upon launching the app, you will be presented with two options: *Assembly* or *Disassembly*.
- Select *Assembly* if you want to build your LEGO set or *Disassembly* if you want to disassemble it.

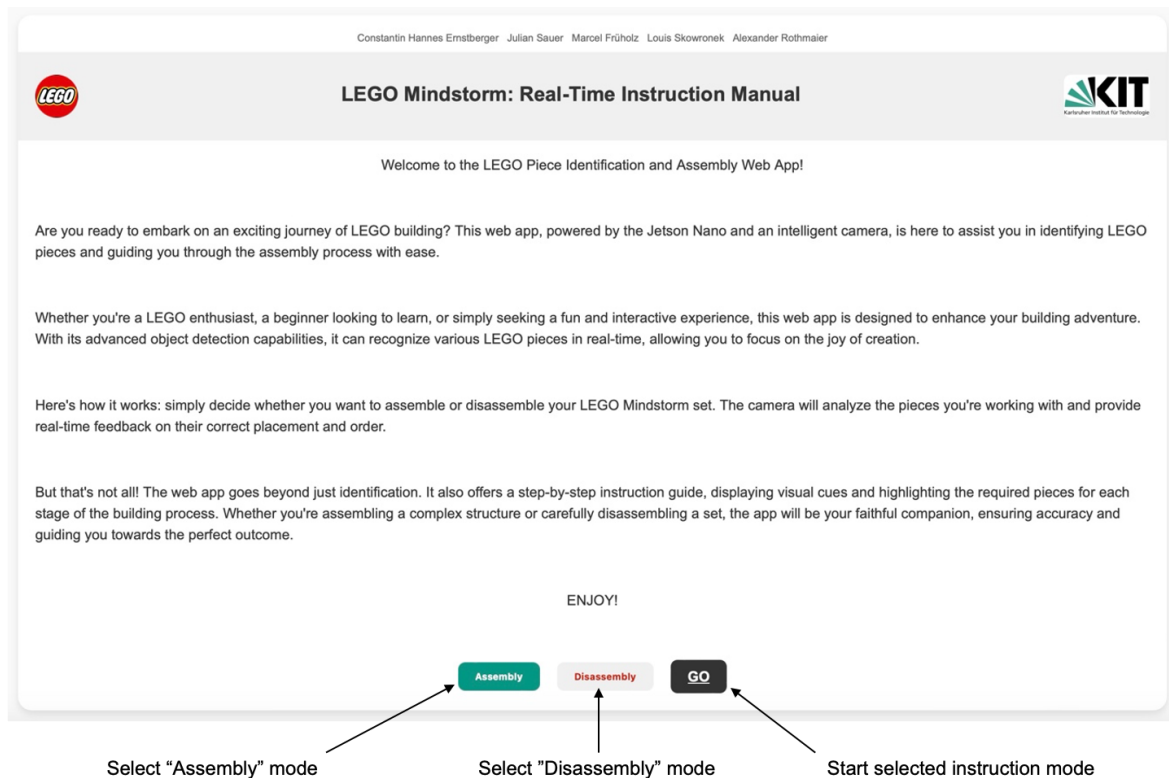


Figure 13: Webapp Homepage

## ASSEMBLY MODE

### 1. Step-by-Step Instructions:

- In Assembly mode, you will start at Step 1 of the building process.
- On the left side of the screen, you will see a live video feed from the camera, showing your LEGO set.
- On the right side, you will find the current instruction step along with the necessary LEGO piece and an image of how to put the parts together, marked in green.
- The detected LEGO pieces will be highlighted with bounding boxes.
- Press the "Check" button to verify if all the necessary parts are present.
  - If yes, the displayed text will tell you to proceed with picking up the necessary parts and build them together according to the instruction image.
  - If not, the displayed text will tell you how many parts are missing.

### 2. Next and Previous Steps:

- After successfully assembling the parts, click *Next* to move on to the next instruction step.
- If you feel you made a mistake, you can also go back to the previous step by clicking *Previous*.

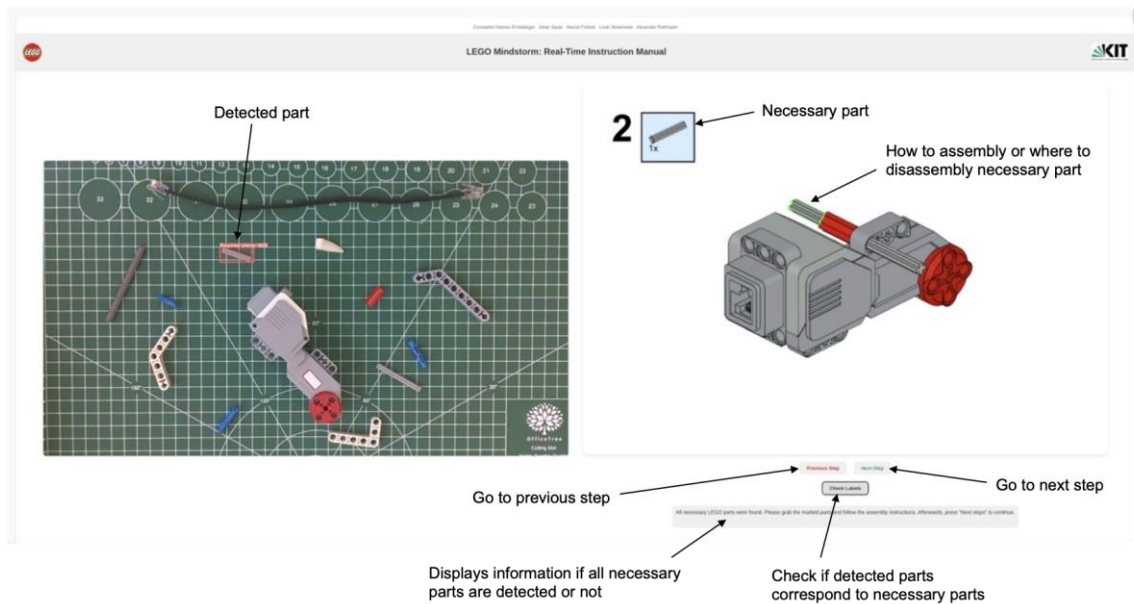


Figure 14: Webapp Assembly Mode

## DISASSEMBLY MODE

### 1. Step-by-Step Disassembly:

- In Disassembly mode, you will start at Step 15 of the building process, where your LEGO set is fully built.
- The screen will look the same as in Assembly mode, but you will be required to disassemble the parts displayed on the screen and place them in front of the camera view.
- Press the *Check* button to verify if all the necessary parts were disassembled correctly:
  - If yes, the displayed text will tell you to proceed.
  - If not, the displayed text will tell you how many parts are missing.

### 2. Next Step:

- After successfully disassembling the parts, click *Next* to move on to the next instruction step.

## ADVANCED SETTINGS

Throughout the whole assembly or disassembly process, you can choose between different settings influencing the detection results and visualization.

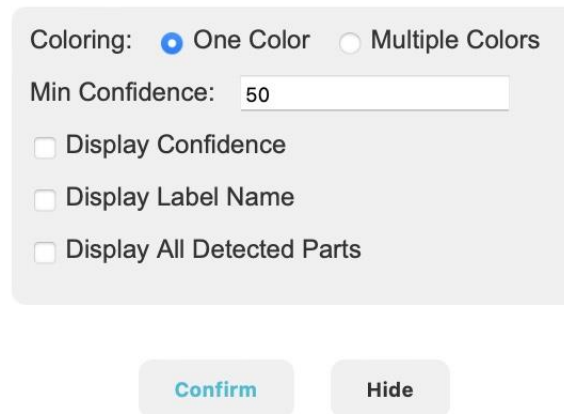
The image shows a software dialog box titled 'Advanced Settings'. It contains several configuration options: a 'Coloring' section with two radio buttons, 'One Color' (which is selected) and 'Multiple Colors'; a 'Min Confidence' section with a text input field containing the value '50'; and three unchecked checkboxes labeled 'Display Confidence', 'Display Label Name', and 'Display All Detected Parts'. At the bottom of the dialog are two buttons: 'Confirm' in blue and 'Hide' in grey.

Figure 15: Advanced Settings Dialog

### 1. Advanced Settings:

- Open *Advanced Settings* dialog by clicking on the button.

### 2. Coloring:

- One Color: Only bounding boxes of necessary parts will be displayed in red.
- Multiple Colors: Bounding box of each detected part will be displayed in a different color.

### 3. Min Confidence:

- Select the min confidence necessary of a detected part for its bounding box to be displayed on the screen.
- Any value outside the possible range (0 – 1) will be set to 1.

### 4. Display Confidence:

- Select if the confidence of each detected part should be displayed or not.

### 5. Display Label Name:

- Select if the label name if each detected part should be displayed or not.

### 6. Display All Detected Parts:

- Select if all detected parts should be displayed or only the necessary parts. If *One Color* coloring is selected, bounding boxes of non-necessary parts will be displayed in grey.

### 7. Confirm:

- Confirm and send selected settings to the server.

### 8. Hide:

- Hide *Advanced Settings* dialog.

# 11 Conclusion

In today's dynamic and ever-evolving industrial landscape, which is increasingly moving towards sustainability and circularity, remanufacturing has become a frequent topic of discussion for manufacturing companies, particularly in the precision mechanics and optics industry, like Zeiss. The challenges faced by these companies are manifold including variable part quality, dependence on technician expertise, part complexity, process flexibility, and increased electrification. The current status quo involves a lot of manual effort and the technical expertise of engineers in the remanufacturing of complex components. To stay competitive and meet high customer expectations, companies must constantly seek innovative solutions to optimize their processes.

Therefore, this project aimed to explore the application of computer vision technology to address these challenges in remanufacturing. By leveraging the power of computer vision, the project sought to demonstrate a real-life case of how remanufacturing operations can be facilitated by increasing productivity and decreasing manual labor. Therefore, the scope of the project was focused on the simulation of disassembly and assembly steps in remanufacturing by assembling Lego Mindstorm components using an object detection system. The insights gained are applicable across different tiers of complexity and scopes of implementation.

The methodology followed in this project demonstrated a systematic and iterative approach involving five sequential steps with two iterations embedded within them. The approach enabled continuous improvement and fine-tuning of the system, ensuring a close alignment with the specific operational needs and requirements of remanufacturing processes. By breaking down the problem into core components of object recognition and employing the Model-View-Controller (MVC) design pattern, the project achieved a clear separation of system components and enhanced the overall system architecture.

In the scope of this project, a minimum viable use case was formulated, allowing the system to recognize and provide input to the assembler on which part should be assembled next based on visible parts. In the future, by adding further use cases, the system's functionality could be expanded, optimizing exception handling, providing feedback on missing parts, and incorporating gamification features.

However, the integration of AI and computer vision into remanufacturing operations presents several hurdles that demand careful consideration and innovative solutions. Throughout the development and deployment process, various challenges emerged, ranging from image quality issues and compatibility conflicts to optimizing model performance and reducing latency. These obstacles required the team to exhibit adaptability, perseverance, and creativity to overcome them successfully. Before taking a decision to implement a similar system, companies should perform

sophisticated analysis of the costs and benefits of this undertaking as well as evaluating their internal capabilities in terms of planning, implementation and lastly maintenance of the system.

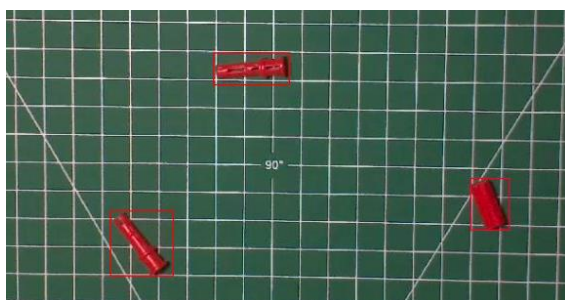
Overall, even though only presentive a very immature system, the project successfully demonstrated the effectiveness of computer vision technology in remanufacturing, presenting a powerful tool for the industry's future. As the prototype evolves and improves, it has the potential to drive significant positive change, unlocking new possibilities for remanufacturing processes, and ultimately, contributing to a more sustainable and efficient industrial sector.

# 12 Appendix

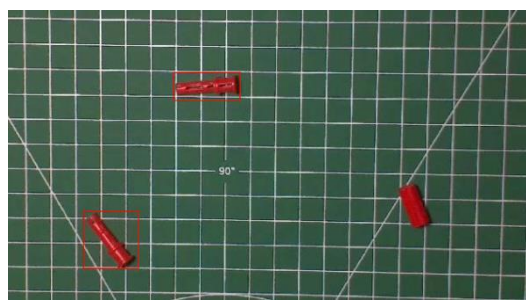
## Labeling Guide: Dos and Don'ts

- **Label every object of interest**

Do:

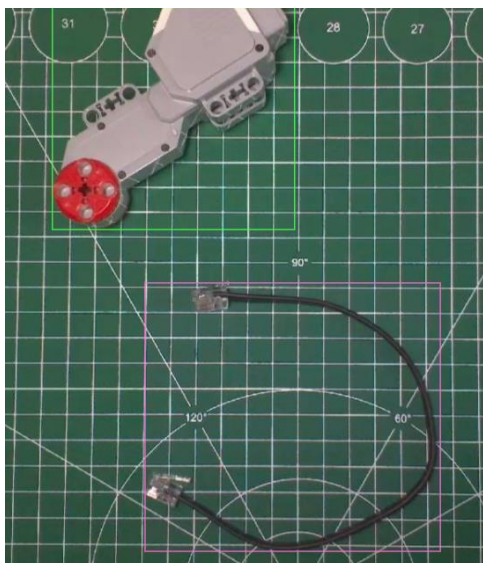


Don't:

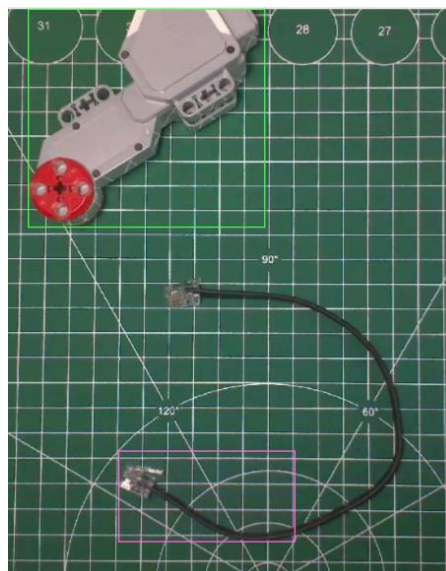


- **Label the entirety of an object**

Do:

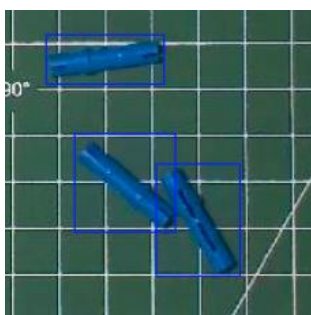


Don't:

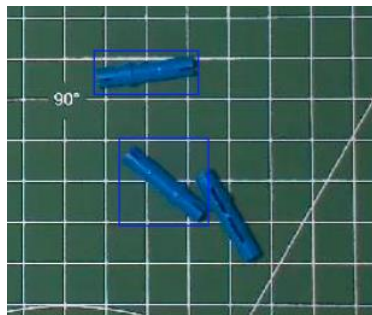


- **Label occluded object**

Do:



Don't:



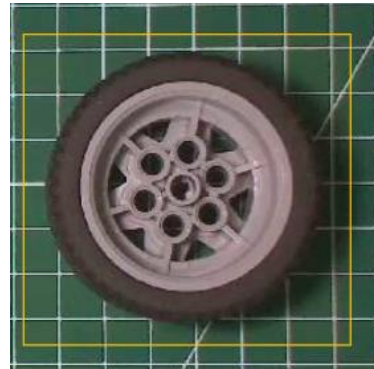


- **Create tight bounding boxes**

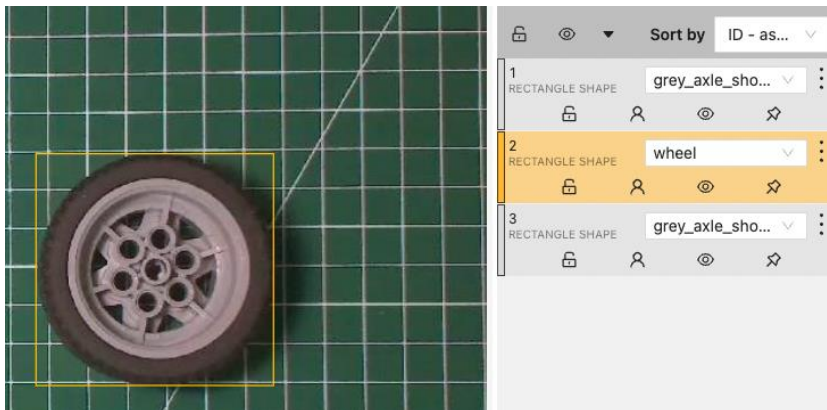
Do:



Don't:

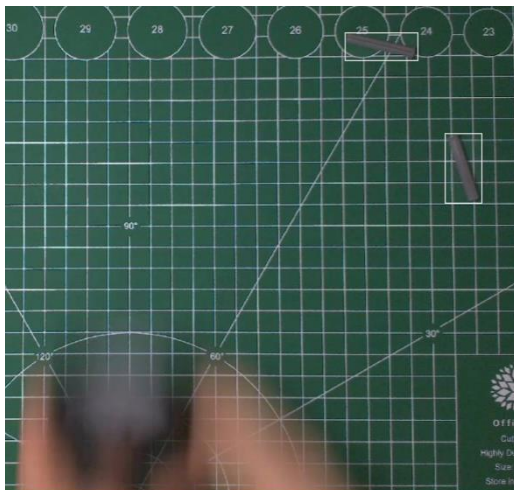


- **Use specific label names**

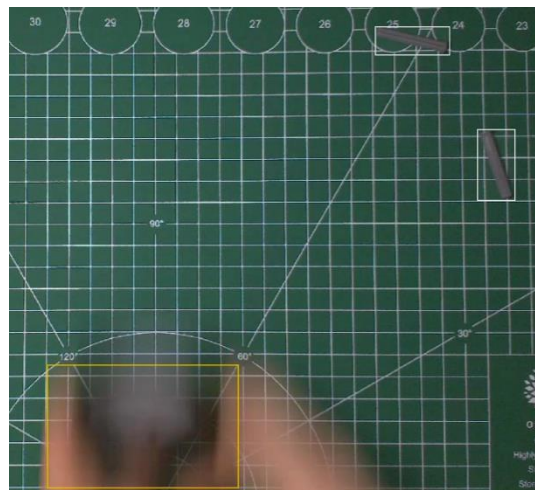


- **Do not label blurred objects**

Do:

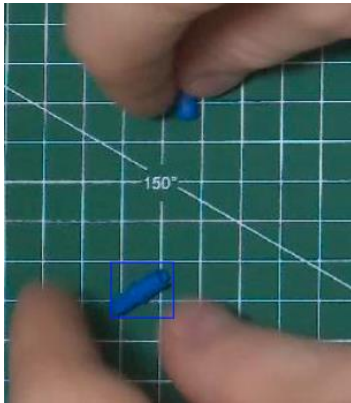


Don't:

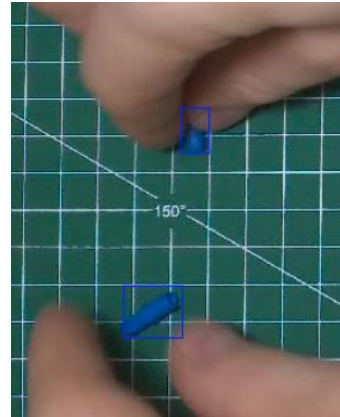


- **Do not label parts that are not clearly recognizable**

Do:

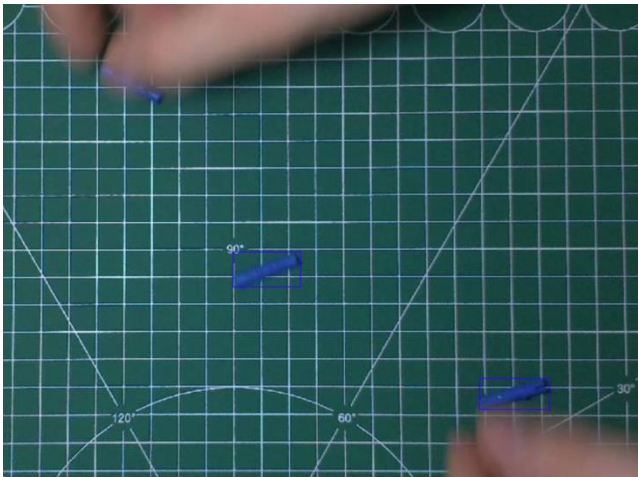


Don't:

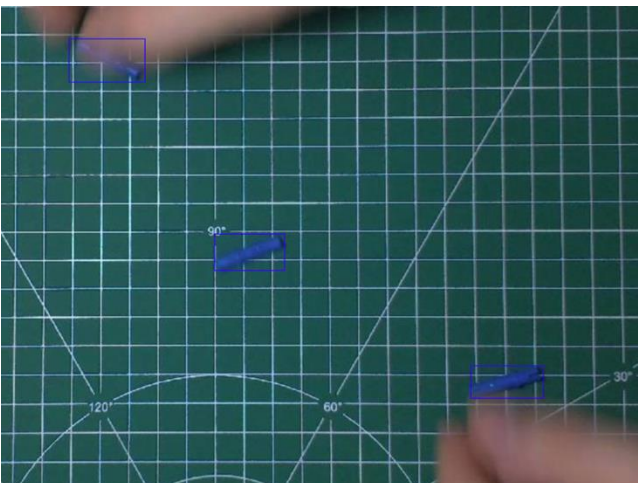


- **Do not label based on context**

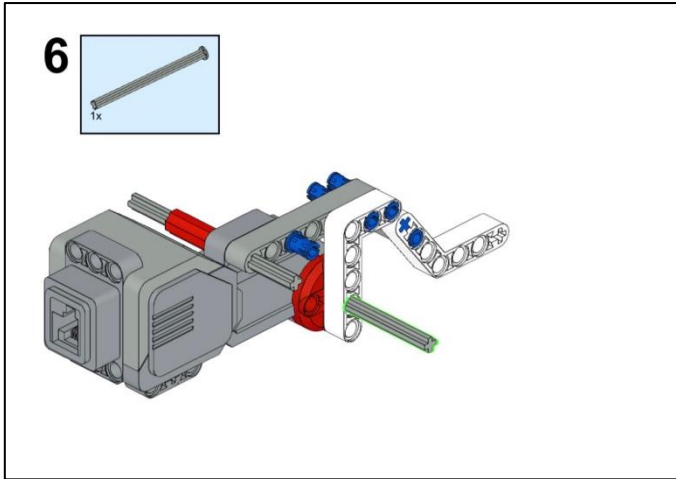
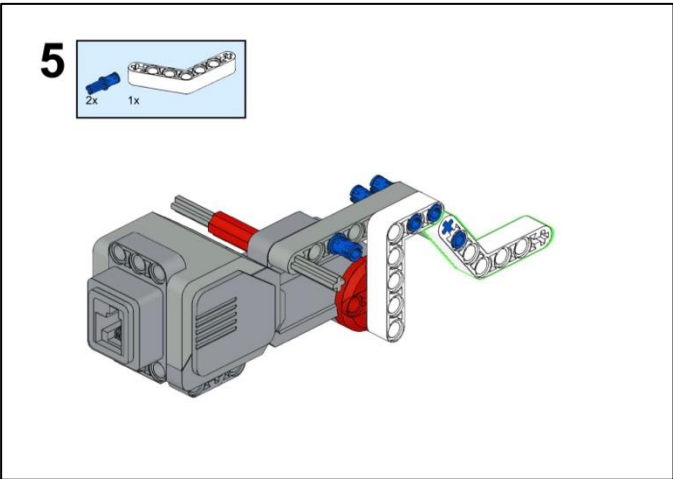
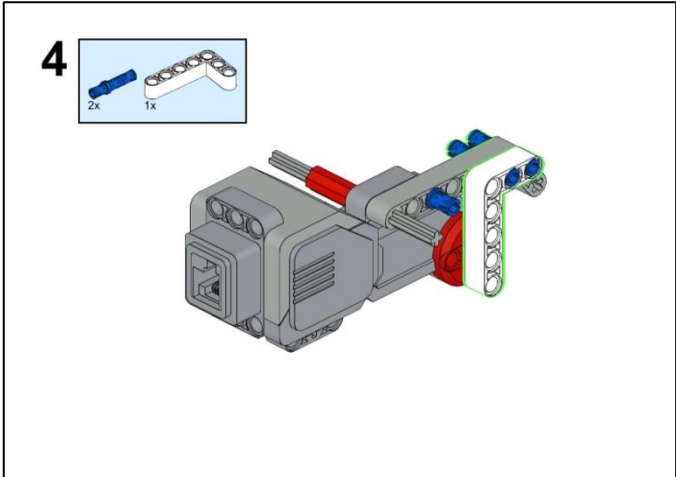
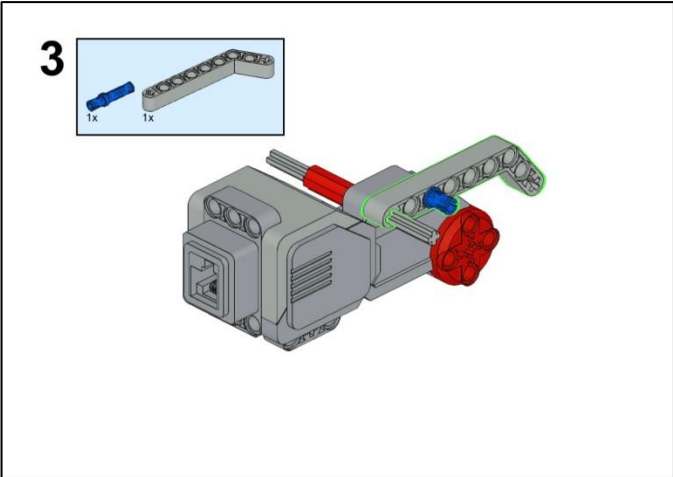
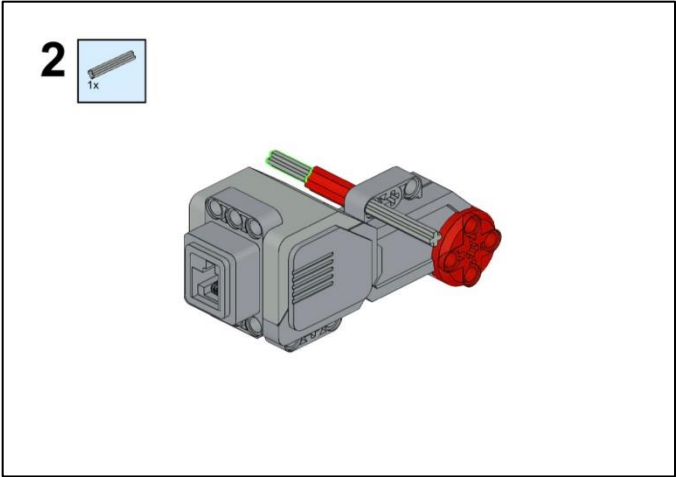
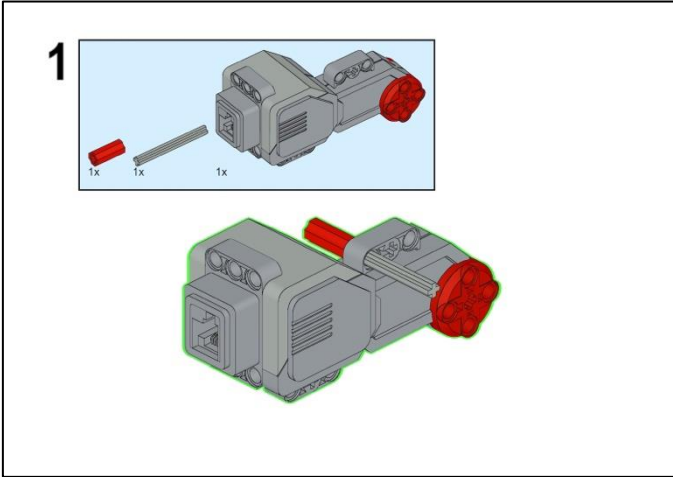
Do (Object at top is not clearly recognizable, no label required):

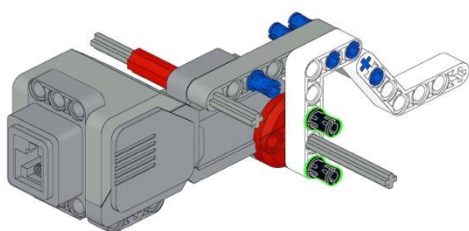
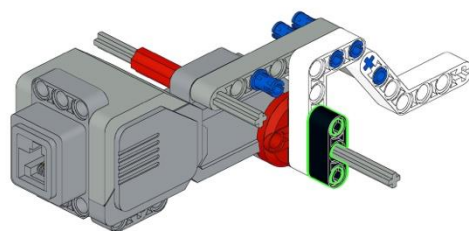


Don't (No labels based on knowledge from previous annotations):

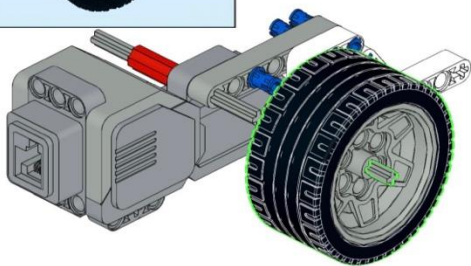


Lego Assembly Instruction

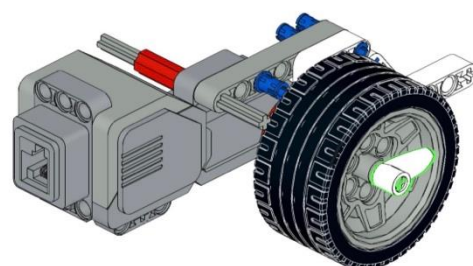


7  2x8  1x

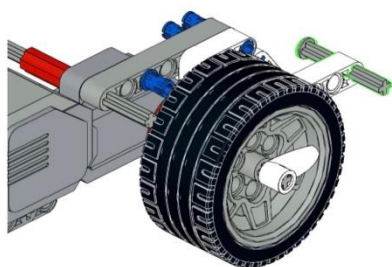
9



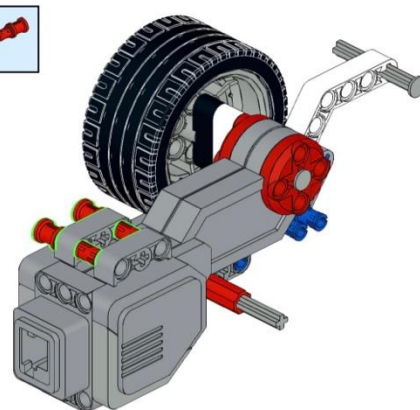
10



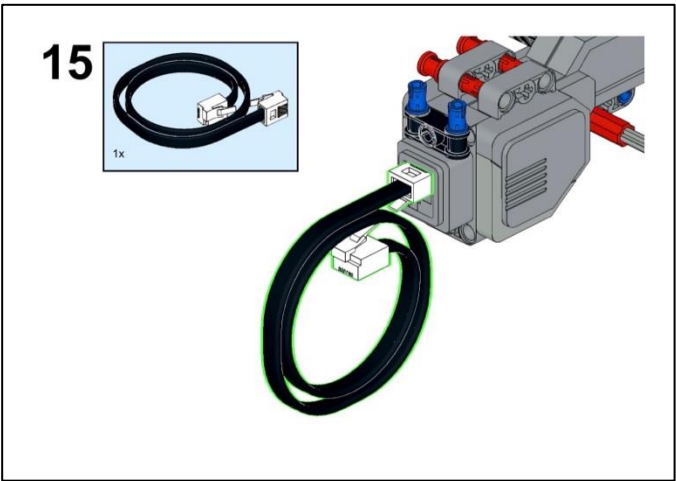
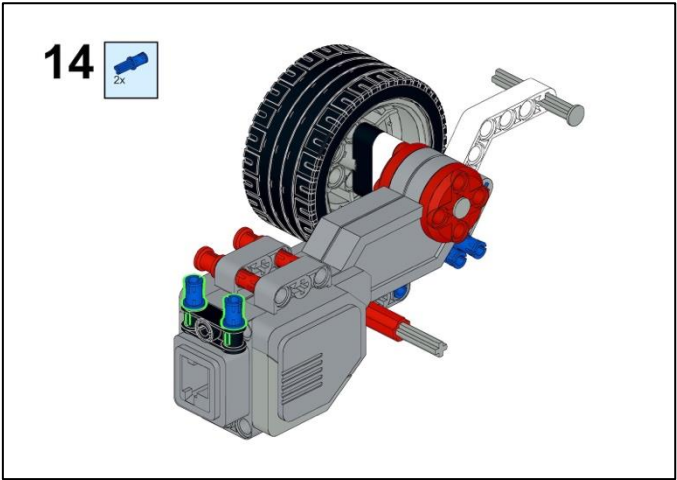
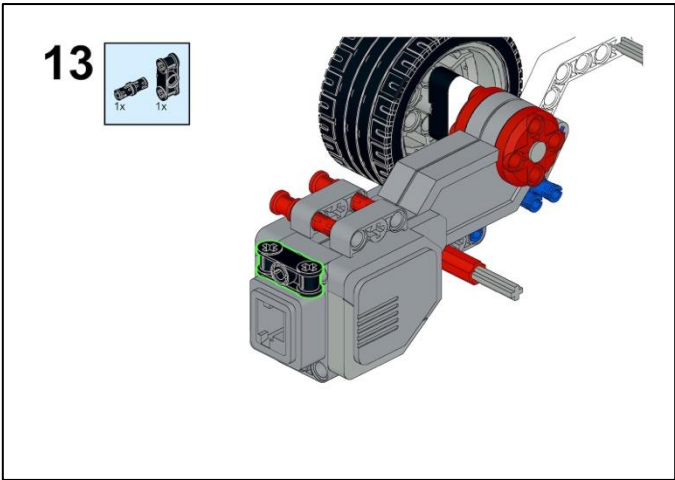
11



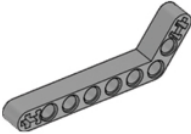
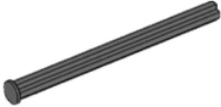

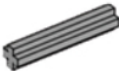
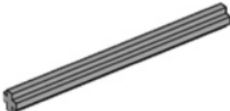




12





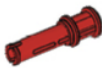


# Naming Conventions

<u>grey_beam_bent</u>	
<u>grey_axle_long_stop</u>	
<u>grey_axle_short_stop</u>	
<u>grey_axle_short</u>	
<u>grey_axle_long</u>	
<u>black_axle_pin_con</u>	
<u>black_beam</u>	
<u>black_pin_short</u>	
<u>red_oct_con</u>	

---

red\_pin\_3L



---

blue\_pin\_3L



---

blue\_axle\_pin



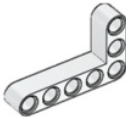
---

white\_beam\_bent



---

white\_beam\_L



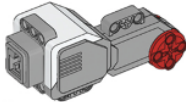
---

white\_tooth



---

engine



---

wheel



---

wire



## 13 References

- Chung, L. N. (2012). *Non-functional requirements in software engineering (Vol. 5)*. . Springer Science & Business Media.
- Sailer, M. (2016). *Wirkung von Gamification auf Motivation*. Springer Fachmedien.
- Pop, & Altar. (2014). Designing an MVC model for rapid web application development. *Elsevier*.
- Redmon. (2016). You Only Look Once: Unified, Real-Time Object Detection. *CVPR*.
- BrickLink. (2023). Studio 2.0. BrickLink. Available online:  
<https://www.bricklink.com/v3/studio/download.page> (Access: 10 August 2023)
- Jocher, G.; Chaurasia, A.; Qiu, J. YOLO by Ultralytics. GitHub. 1 January 2023. Available online:  
<https://github.com/ultralytics/ultralytics> (Access: 11 August 2023).