Follow-Me Fahrroboter: Entwicklerdokumentation

Inhaltsverzeichnis

1. Entwurfsdokumentation

2. Architecture Notebook: Follow-Me Fahrroboter

2.1. Zweck

Dieses Dokument beschreibt die Philosophie, Entscheidungen, Nebenbedingungen, Begründungen, wesentliche Elemente und andere übergreifende Aspekte des Systems, die Einfluss auf Entwurf und Implementierung haben.

2.2. Architekturziele und Philosophie

Unsere Philosophie in diesem Projekt ist es, das bestmögliche Ergebnis in gegebener Zeit zu erzielen und das Projekt so modular wie möglich zu gestalten. Übersichtlichkeit und Erweiterbarkeit sind unsere großen Ziele. Unsere Teamphilosophie ist es, aufkommende Aufgaben so schnell und bestmöglich für die jeweilige Iteration zu erledigen und einen funktionsfähigen Prototyp zu entwickeln. Besondere Herausforderungen sind:

- Die Integration mit bestehenden Systemen (z.B. ROS 2).
- Die Berücksichtigung von Hardware-Einschränkungen (z.B. Raspberry Pi und Alphabot 2).

2.3. Annahmen und Abhängigkeiten

Es gibt zwei ausschlaggebende Punkte, die unsere Architektur stark beeinflussen:

1. **ROS 2**: Wir haben die Vorgabe erhalten, ROS 2 zu verwenden. ROS 2 bringt einige Eigenheiten mit sich, da es als eine Art Betriebssystem funktioniert. ROS 2 kommt mit einer Paketstruktur, innerhalb derer sich Nodes befinden, die den Code enthalten. Diese Nodes können systemweit mithilfe von Messages interagieren. Wir sind somit an diese Strukturvorgaben gebunden.
2. **Hardware-Limitierungen Prototyp 1**:
    - **Raspberry Pi 4 8GB**: Der Raspberry Pi ist ein kostengünstiger, kleiner Computer, der vielseitig einsetzbar und einfach zu bedienen ist. Er weist jedoch Leistungsgrenzen auf, die Entscheidungen unter Berücksichtigung dieser Performance-Limitierungen erforderlich machen.
    - **Alphabot 2**: Der Alphabot 2 ist für die Verwendung mit dem Raspberry Pi konzipiert und enthält Komponenten wie Servo- und DC-Motoren, PCA9685, LEDs, Joystick, IR-Fernbedienung, Line-Sensoren und eine Kamera. Einschränkungen bestehen aufgrund der begrenzten Batterielebensdauer, der engen Integration und der veralteten Software.
3. **Hardware-Limitierungen Prototyp 2**:
    - **Jetson Nano**: Der Jetson Nano ist ein etwas teurerer, aber leistungsstärkerer kleiner Computer, der vielseitig einsetzbar und einfach zu bedienen ist. Auch er weist Leistungsgrenzen auf, die Entscheidungen unter Berücksichtigung dieser Performance-Limitierungen erforderlich machen.
    - **Arduino**: Das Arduino bietet eine Vielzahl an Erweiterungs- und Einsatzmöglichkeiten, ist

jedoch dadurch limitiert, dass es kein Threading unterstützt. Ressourcenmanagement ist daher bei der Verwendung besonders wichtig.

2.4. Architektur-relevante Anforderungen

2.4.1. Benutzbarkeit (Usability)

- NFAU-2: Das System soll als portable Möglichkeit zur Mitführung gestaltet werden

2.4.2. Zuverlässigkeit (Reliability)

- NFAR-1: Das System sollte möglichst eine Stunde mit vollgeladenem Akku laufen
- NFAR-2: Der Roboter sollte in der Lage sein, durch eine handelsübliche Powerbank, sobald dieser an den Zusatzakku angeschlossen wurde, in Betrieb genommen zu werden.
- NFAR-3: Bei einem Fehler soll das System einfach durch Error-Handling in der Lage sein, sich nicht vollständig zu beenden, sondern einen Fehler zu werfen und so weiterhin zu laufen.

2.4.3. Leistung (Performance)

- NFAP-1: Die Antwortzeit der Bilderkennung soll schnell genug sein, um Personen folgen zu können. Eine Verarbeitungszeit muss hier bei < 1 Sekunde sein.
- NFAP-3: Der Roboter muss in der Lage sein, Bewegungsdaten mit einer Rate von ca. 100-1000 Datenpaketen pro Sekunde zu verarbeiten.
- NFAP-4: Der Roboter samt Software muss in der Lage sein, innerhalb von 10 Minuten zu starten und innerhalb von 1 Minute sich vollständig zu beenden.

2.4.4. Wartbarkeit (Supportability)

- NFAS-1: Durch die Integration mit dem OS ROS II, muss das System in der Lage sein, mit anderen Systemen kompatibel zu sein.

2.5. Entscheidungen, Nebenbedingungen und Begründungen

1. **Verwendung von ROS 2 Humble**: Wir haben uns für die Nutzung von ROS 2 Humble entschieden. Diese Version von ROS 2 bietet eine Vielzahl an Softwarelösungen sowie eine umfassende Dokumentation. Jedoch ist ROS 2 generell sehr komplex und erfordert eine lange Einarbeitungszeit.
2. **Verwendung von Ubuntu Server 22.04.03 LTS (64-Bit)**: Als Betriebssystem haben wir Ubuntu Server 22.04.03 LTS (64-Bit) ausgewählt, da es das empfohlene Betriebssystem für ROS 2 Humble ist. Es ist mit vielerlei Software kompatibel und gehört zu den aktuellsten Ubuntu-Versionen. Ein Nachteil besteht jedoch darin, dass es keine grafische Benutzeroberfläche bietet.
3. **Verwendung von Python als Programmiersprache**: Für die Programmierung verwenden wir Python. Python ist eine der beiden Sprachen, die mit ROS 2 kompatibel sind, und lässt sich

einfacher implementieren als C++. Allerdings können in ROS 2 mit Python keine Message-Formate erstellt werden, sodass man auf die Standardformate angewiesen ist.

4. **Verwendung von CV Bridge**: Die CV Bridge wird genutzt, um eine einfache Umwandlung des ROS 2 Image-Formats in das OpenCV Image-Format zu ermöglichen.
5. **Hinzufügen einer Web-Oberfläche**: Zur Anzeige der vom Human-Detector bearbeiteten Bilder und für das notwendige Debugging haben wir eine Web-Oberfläche integriert. Dies führt jedoch zu einem Performance-Verlust.
6. **Verwendung von Flask für das Web-Tool**: Für das Web-Tool haben wir uns für Flask entschieden. Die Implementierung in Python ist relativ einfach, jedoch treten teilweise Kompatibilitätsprobleme mit ROS 2 auf, und es entsteht ein erhöhter Performance-Bedarf durch das Threading.
7. **Verwendung einer USB-Kamera**: Wir verwenden eine USB-Kamera, da diese eine gute Qualität und einen großen Winkel bietet. Ein Nachteil ist jedoch der hohe Stromverbrauch und der Fakt, dass man sie schlecht mit Servos bewegen kann.
8. **Wechsel auf OpenCV Video Stream Capture**: Für das Videostream-Capturing nutzen wir OpenCV Video Stream Capture. Dadurch ist das direkte Ansprechen der Kamera in Python möglich. Allerdings läuft das Capturing permanent und verbraucht somit Performance.
9. **Wechsel auf YOLO (You Only Look Once)**: Schließlich sind wir auf YOLO (You Only Look Once) umgestiegen, da es eine bessere Erkennungsgenauigkeit bietet. Der hohe Performance-Verbrauch und die relative geringe Verarbeitungsgeschwindigkeit sind jedoch Nachteile.

2.6. Architekturmechanismen

2.6.1. Analysemechanismen

Bilderfassung und Bildvorverarbeitung: UC1 - Roboter starten

- **Name**: Bilderfassung und -vorverarbeitung
- **Beschreibung**: Mechanismus zur Erfassung von Bildern mit einer Kamera und deren Vorverarbeitung für Bildverarbeitung.
- **Attribute**:
  - **Quelle**: Kameraeingang (z.B. USB, CSI)
  - **Auflösung**: Konfigurierbare Bildauflösung (z.B. 640x480, 1280x720)
  - **Bildrate**: Konfigurierbare Bildrate (z.B. 30 FPS)
  - **Vorverarbeitungsschritte**: Bildskalierung, Normalisierung, Rauschreduzierung

Personenerkennung: UC1 - Roboter starten

- **Name**: Personenerkennung
- **Beschreibung**: Mechanismus zur Erkennung und Identifizierung von Personen in erfassten Bildern.
- **Attribute**:

- **Algorithmus**: Maschinelles Lernmodell (z.B. YOLO, SSD, Faster R-CNN)
- **Erkennungsgenauigkeit**: Konfigurierbarer Schwellenwert für Erkennungssicherheit
- **Erkennungsgeschwindigkeit**: Echtzeitverarbeitung

Berechnung der Steuersignale: UC1 - Roboter starten

- **Name**: Berechnung der Steuersignale
- **Beschreibung**: Mechanismus zur Berechnung von Steuersignalen für die Navigation des Roboters.
- **Attribute**:
  - **Winkelberechnung**: Berechnung des Winkels zur Verfolgung der erkannten Person
  - **Geschwindigkeitsberechnung**: Berechnung der Geschwindigkeit zur Verfolgung der erkannten Person

Verfolgung: UC1 - Roboter starten

- **Name**: Verfolgung
- **Beschreibung**: Mechanismus zur Verfolgung der erkannten Person.
- **Attribute**:
  - **Schnittstelle zur Hardware**: Schnittstelle zur Steuerung der Roboterhardware

2.6.2. Entwurfsmechanismen

Bilderfassung und -Vorverarbeitung: UC1 - Roboter starten

- **Name**: Bilderfassung und -vorverarbeitung
- **Beschreibung**: Mechanismus zur Erfassung und Vorverarbeitung von Bildern.
- **Entwurf**:
  - **Kameraschnittstelle**: Entwicklung einer benutzerdefinierten Kameraschnittstelle
  - **Bildformat**: Definition eines Standardformats für Bilder (z.B. RGB)
  - **Vorverarbeitungsschritte**: Implementierung von Funktionen zur Bildskalierung, Normalisierung und Rauschreduzierung

Personenerkennung: UC1 - Roboter starten

- **Name**: Personenerkennung
- **Beschreibung**: Mechanismus zur Erkennung und Identifizierung von Personen.
- **Entwurf**:
  - **Algorithmus**: Heraussuchen eines geeigneten ML-Modells
  - **Erkennungsgenauigkeit**: Implementierung konfigurierbarer Schwellenwerte
  - **Erkennungsgeschwindigkeit**: Echtzeitfähigkeit sicherstellen

Berechnung der Steuersignale: UC1 - Roboter starten

- **Name**: Berechnung der Steuersignale
- **Beschreibung**: Mechanismus zur Berechnung der Steuersignale zur Verfolgung der erkannten Person.
- **Entwurf**:
  - **Verfolgungsalgorithmus**: Entwicklung eines benutzerdefinierten Algorithmus
  - **Leistung**: Effizient und zuverlässig mit niedriger Latenz

Verfolgung: UC1 - Roboter starten

- **Name**: Verfolgung
- **Beschreibung**: Mechanismus zur Verfolgung der erkannten Person.
- **Entwurf**:
  - **Steuerungsschnittstelle**: Entwicklung einer Schnittstelle zur Steuerung der Roboterhardware

2.6.3. Implementierungsmechanismen

Bilderfassung und -vorverarbeitung: UC1 - Roboter starten

- **Struktur**:
  - **ROS2-Paket**: Erstellen eines ROS2-Pakets
  - **Knoten**: Entwicklung eines ROS2-Knotens zur Bilderfassung

Personenerkennung: UC1 - Roboter starten

- **Struktur**:
  - **ROS2-Paket**: Erstellen eines ROS2-Pakets
  - **Knoten**: Entwicklung eines ROS2-Knotens zur Personenerkennung
  - **Topics**: Nutzung eines Topics zur Veröffentlichung von Erkennungsergebnissen

Berechnung der Steuersignale: UC1 - Roboter starten

- **Struktur**:
  - **ROS2-Paket**: Entwicklung eines ROS2-Pakets
  - **Knoten**: Implementierung eines ROS2-Knotens zur Berechnung der Steuersignale
  - **Topics**: Nutzung eines Topics zur Übermittlung von Verfolgungsinformationen
  - **Schnittstellen**: Zugriff auf die Schnittstellen zur Steuerung der Roboterhardware

Verfolgung: UC1 - Roboter starten

- **Struktur**:

- **Schnittstellen**: Entwicklung einer Schnittstelle zur Steuerung der Roboterhardware
- **Serielle Kommunikation**: Kommunikation mit der Roboterhardware über serielle Schnittstelle
- **Netzwerkkommunikation**: Kommunikation mit der Roboterhardware über Netzwerkverbindung

2.7. Wesentliche Abstraktionen

2.7.1. Personen

- **Benutzer** - Person, die das System verwendet
- **Administrator** - Person, die das System konfiguriert und wartet

2.7.2. Roboter

- **Hardwareebene**: Raspberry Pi, Alphabot 2, Arduino Uno, Jetson Nano
- **Betriebssysteme**: ROS 2 Humble, Ubuntu Server 22.04.
- **Softwareebene**: Eigene entwickelte Software, Open-Source-Software.

2.8. Schichten oder Architektur-Framework

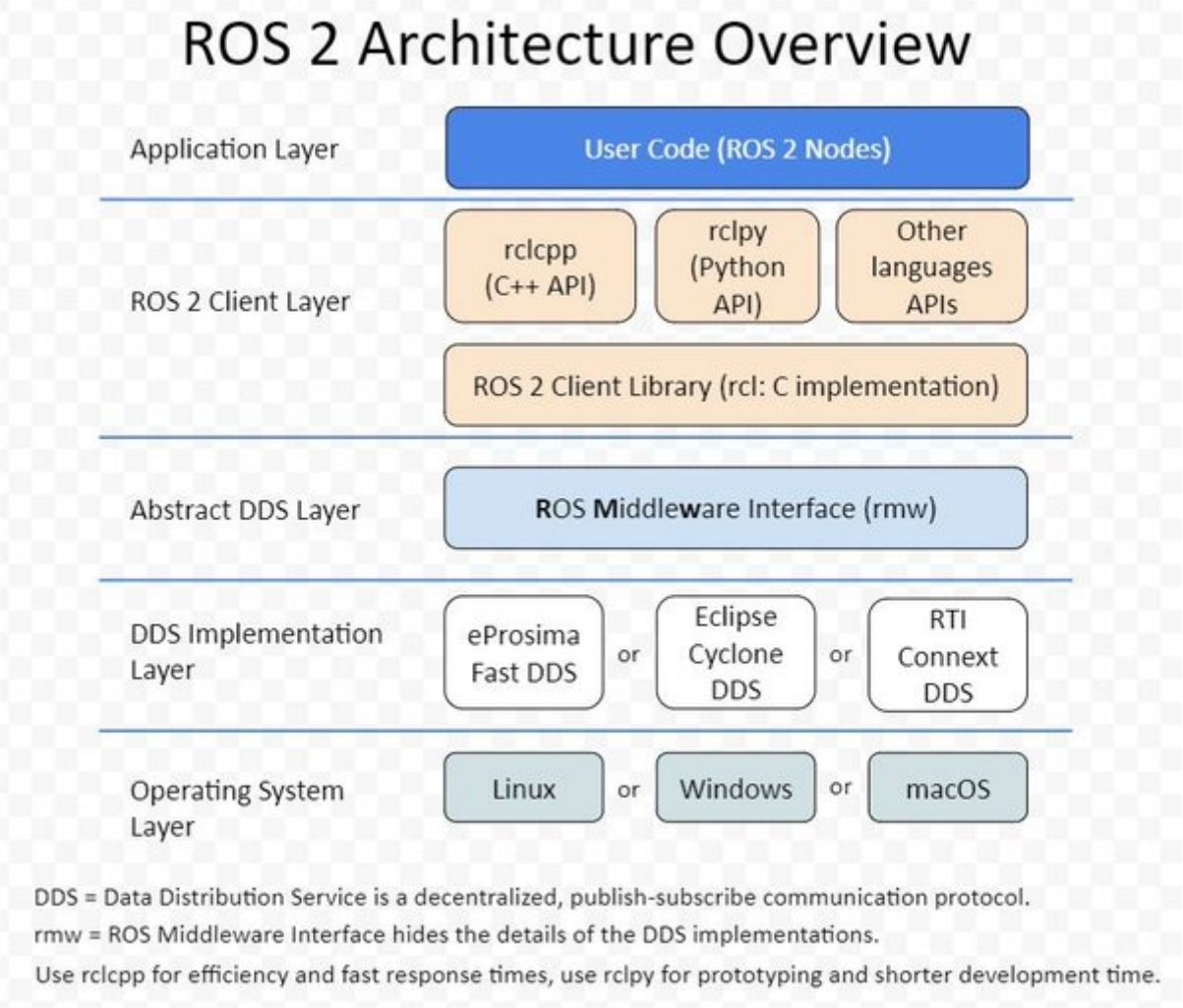

Figure 1. Diagramm der ROS2 Architektur

Unser Architektur-Framework wird durch die Verwendung von ROS II vorgegeben.

Als Middleware verwendet ROS II dabei die Data Distribution Service (DDS) Architektur.

Darauf baut dann der ROS II Client-Layer auf, welcher die Bibliotheken für die Verwendung von ROS zur Verfügung stellt.

Zuletzt der Application-Layer, welcher die eigentliche Anwendung enthält.

2.9. Architektursichten (Views)

2.9.1. Logische Sicht (C4-Modell):

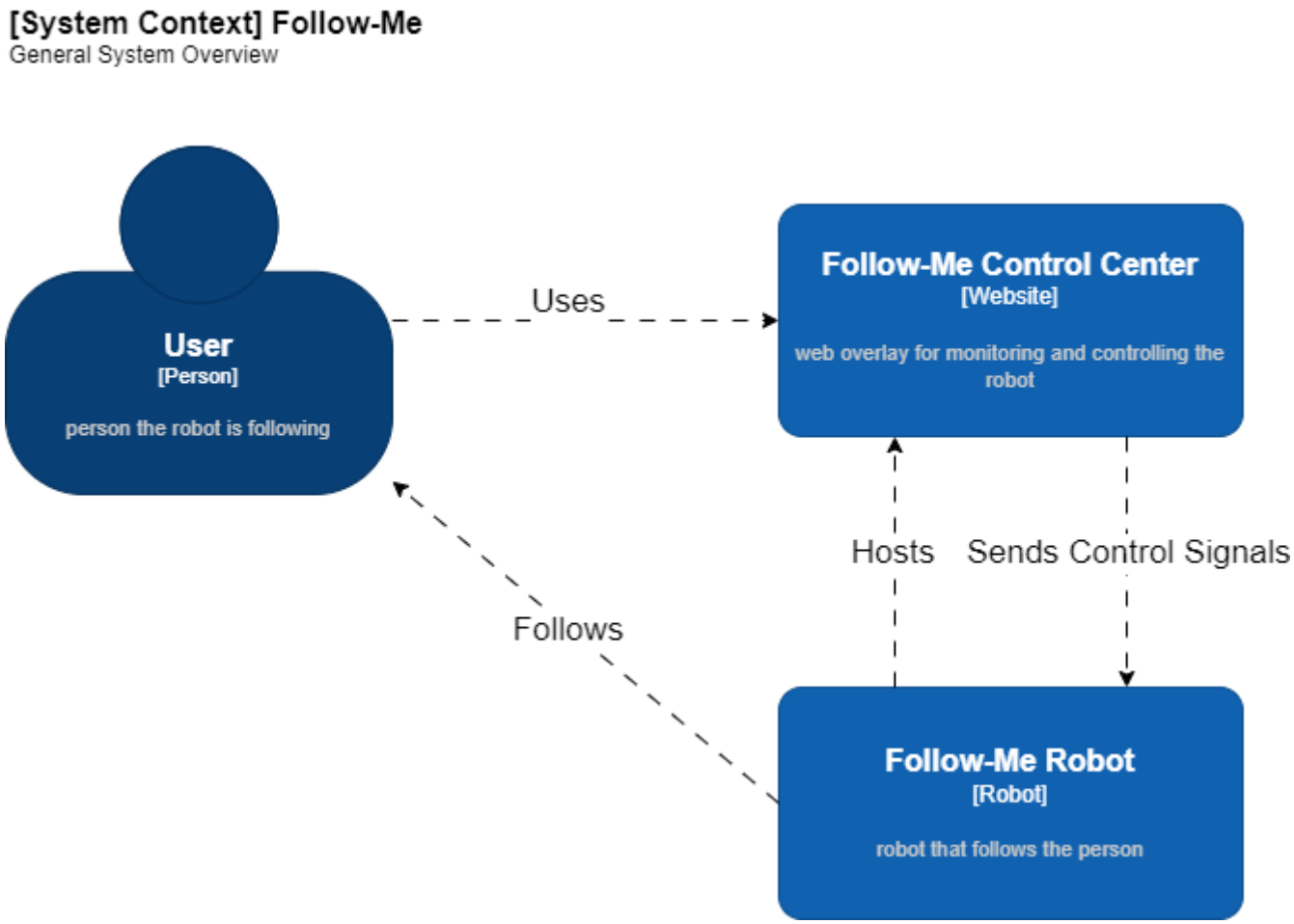

Figure 2. System Context Diagramm

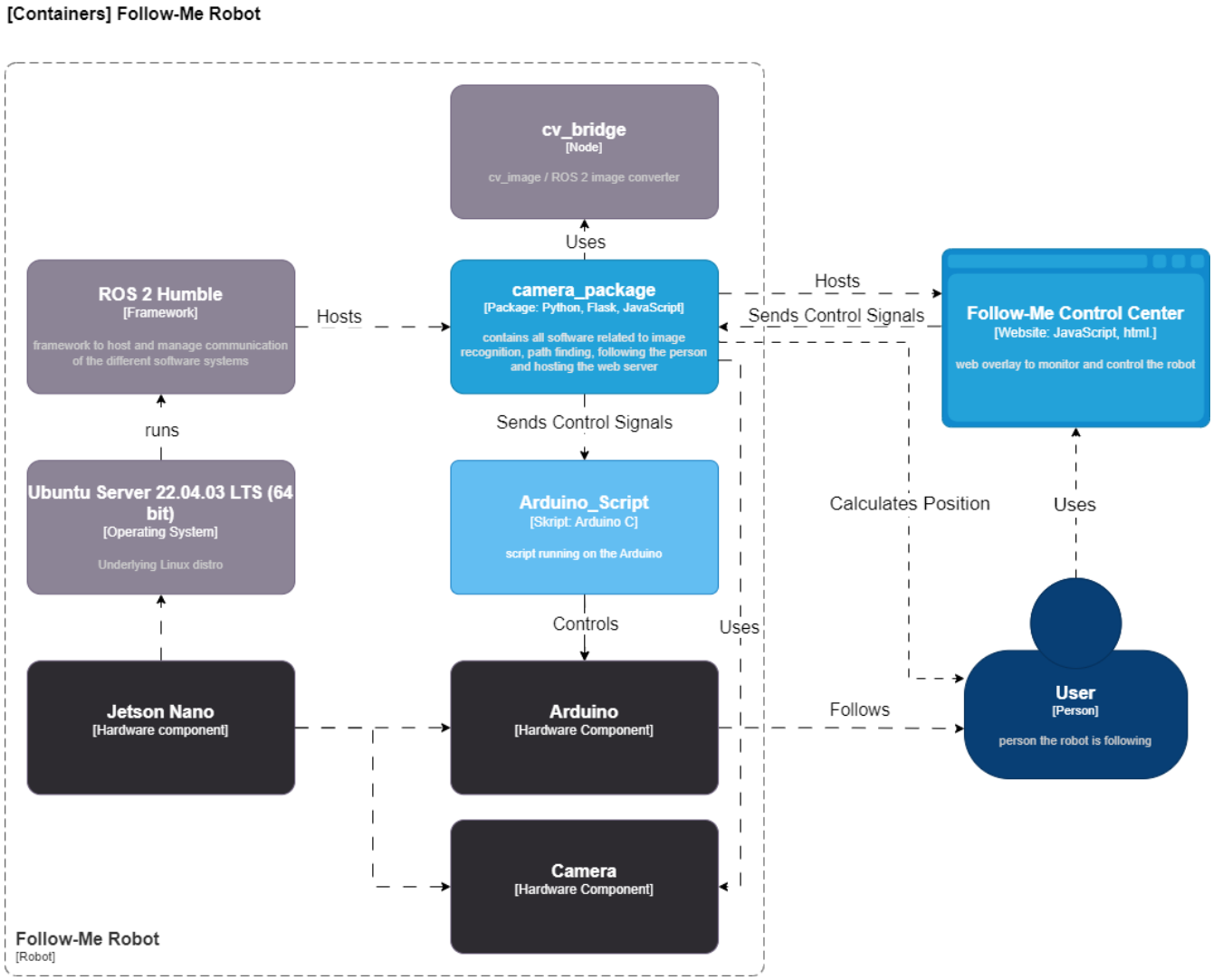

Figure 3. System Container Diagramm

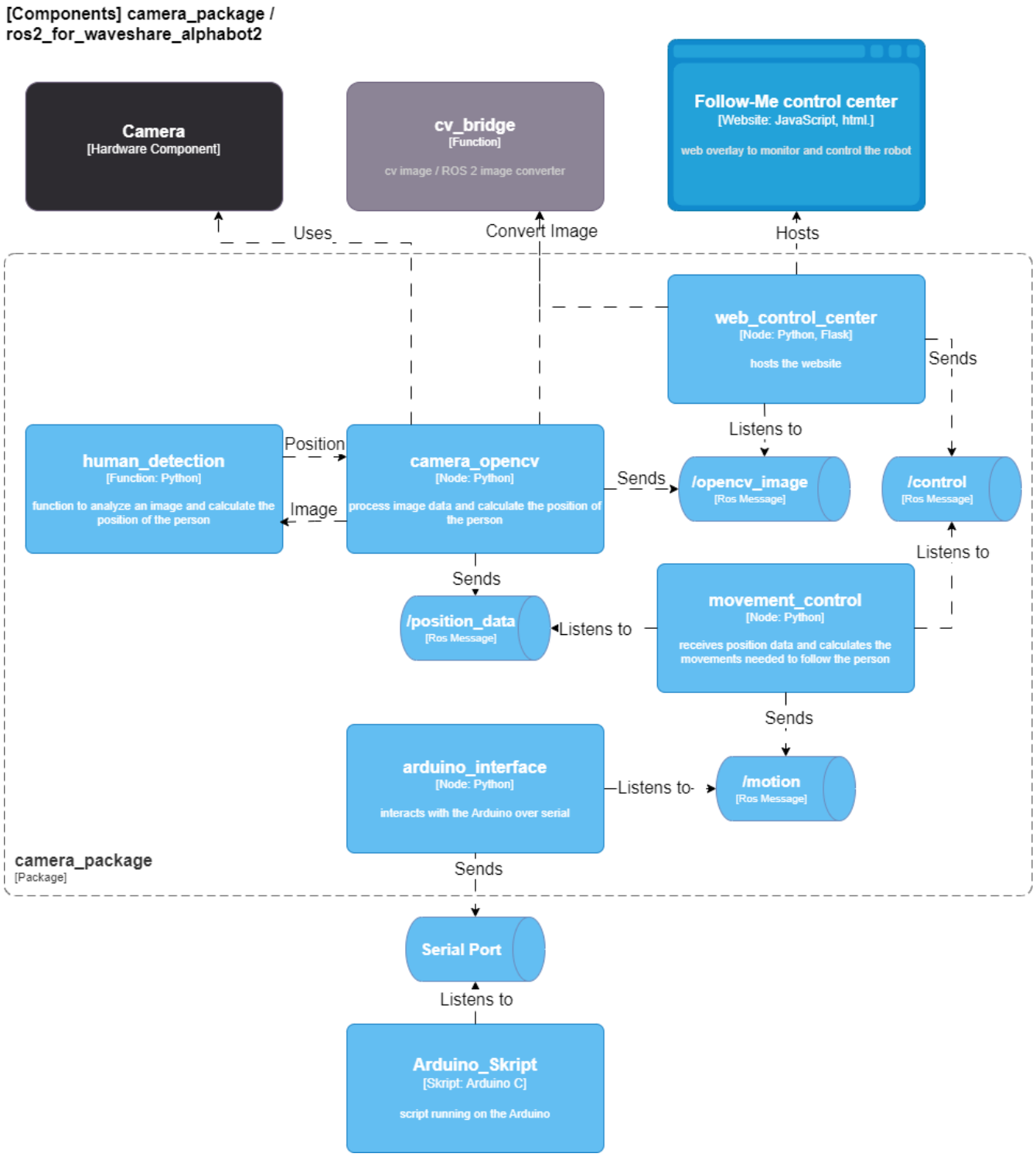

Figure 4. System Component Diagramm

2.9.2. Physische Sicht (Betriebssicht):

Wir verwenden folgende Hardware zur Betreibung der Software:

- Prototyp 1
  - Raspberry Pi 4 8GB
  - Alphabot II
- Prototyp 2
  - NVIDIA Jetson Nano

- Arduino Uno
- Adafruit Motor Shield v2.3
- 2 x DC Motoren
- USB-Webcam
- WLAN-Modul
- Batteriepack 4 AA-Baterien
- Batteriepack 5V 4A
- Bluetooth HC-05 Modul (nur für Tests benötigt)

Des Weiteren wird ein PC für das Ausführen der Software benötigt.

2.9.3. Use Cases:

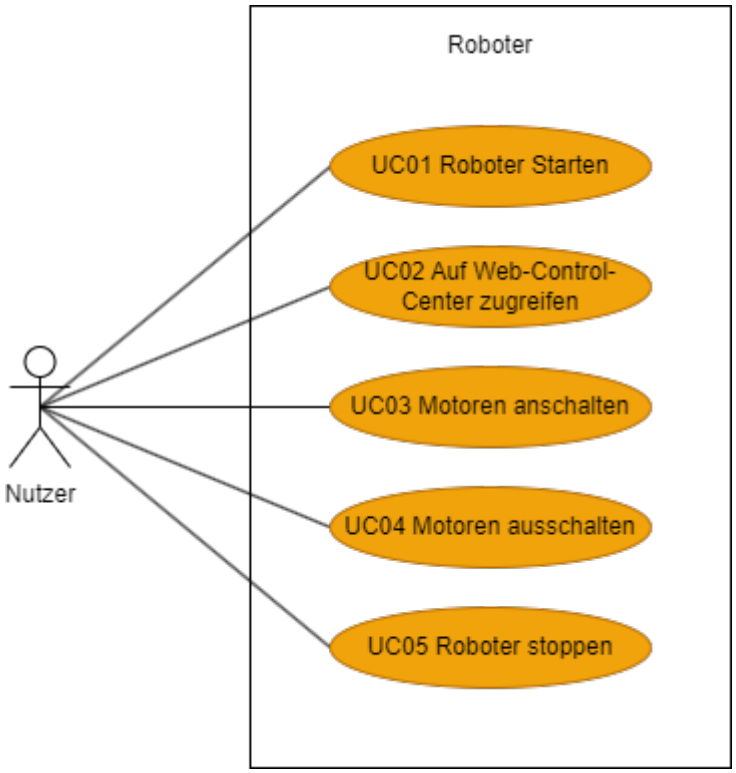

Figure 5. Use Case Diagramm

3. Softwaredokumentation

3.1. camera_opencv

3.1.1. Klasse: CameraOpencv

Die Klasse CameraOpencv repräsentiert einen ROS-Node, der Bilddaten von einer Kamera abonniert, diese verarbeitet und dann Positionsdaten veröffentlicht.

Methoden:

1. *init*(self): Der Konstruktor der Klasse initialisiert die ROS-Node mit dem Namen 'camera_subscriber', erstellt Publisher für Positionsdaten (/position_data) und Bilder (/opencv_image). Außerdem initialisiert er die OpenCV Videoaufnahme (vid0) und den Detektor für die Personenerkennung (detector). Ein Timer wird erstellt, der alle 0,5 Sekunden die Methode loop() aufruft.
2. loop(self): Diese Methode wird jedes Mal aufgerufen, wenn der Timer abläuft. Sie liest ein Bild von der Kamera, führt eine Personenerkennung durch und veröffentlicht die Positionsdaten sowie das verarbeitete Bild.
3. publish_data(self, Position, time1, return_image): Diese Methode veröffentlicht die Positionsdaten und das verarbeitete Bild. Sie konvertiert das Bild in ein ROS-Image-Format und fügt den Positionsdaten Zeitstempel hinzu, bevor sie diese veröffentlicht.

3.2. movement_control

3.2.1. Klasse: MovementControl

Die Klasse MovementControl repräsentiert einen ROS-Node, die Positionsdaten empfängt, diese verarbeitet und entsprechende Steuersignale an Servos und Motoren sendet.

Methoden:

1. *init*(self): Der Konstruktor der Klasse, initialisiert die ROS-Node mit dem Namen 'movement_control'. Verschiedene Parameter wie maximale Winkel, Distanz zur Person und Bewegungseinstellungen werden gesetzt. Publisher für Servo- und Motorsteuerbefehle werden erstellt. Abonnements für Positionsinformationen von der Kamera ('/position_data'), Joystick-Eingaben ('/joystick') und Steuerbefehle ('/control') werden eingerichtet.
2. position_callback(self, Position): Diese Methode wird aufgerufen, wenn Positionsinformationen von der Kamera empfangen werden. Sie verarbeitet die erhaltenen Daten, berechnet Bewegungsparameter und sendet Steuerbefehle an Servos und Motoren.
3. control_joystick(self, msg): Diese Methode wird aufgerufen, wenn Joystick-Eingaben empfangen werden. Sie verarbeitet die erhaltenen Eingaben und steuert die Bewegung entsprechend.
4. control(self, msg): Diese Methode wird aufgerufen, wenn allgemeine Steuerbefehle empfangen werden. Sie verarbeitet die erhaltenen Befehle und steuert die Bewegung entsprechend.

3.3. Utility-Funktionen

1. calculate_speed_variable_time(base_rpm, radius, angle, wheel_distance, wheel_radius, correction_factor): Berechnet die Geschwindigkeiten der Räder basierend auf einer variablen Zeit, abhängig von der Basis-Drehzahl, dem Radius, dem Winkel, dem Radabstand, dem Radradius und einem Korrekturfaktor.
2. calculate_movement_variable_time(self, base_rpm, angle, move): Verarbeitet die von der calculate_speed_variable_time Funktion Berechneten Bewegungsparameter und gibt die gewünschten Drehzahlen sowie die Bewegungsdauer für die Motoren zurück.

3. calculate_angle(self, servo_pan): Berechnet die Winkel für die Servos basierend auf den aktuellen Koordinaten. Es kann zwischen horizontaler (Servo-Pan) und vertikaler (Servo-Tilt) Bewegung gewählt werden.
4. determine_percentage_of_height(self): Bestimmt den prozentualen Anteil der Höhe der Person im Bild im Vergleich zur maximalen Höhe, die vom Winkel erfasst wird.
5. aproximate_distance(self, lenght_y): Näherungsberechnung der Entfernung zur Person basierend auf der erkannten Größe im Bild.
6. set_timestamp(list, set_first_time=True): Setzt den Zeitstempel in einer Liste auf die aktuelle Zeit.
7. get_current_time(): Gibt die aktuelle Zeit zurück.
8. compare_times(time1, time_old, old_time): Vergleicht zwei Zeitstempel und bestimmt, ob die neueren Daten verwendet werden sollen.
9. add_seconds_to_time(time1, milliseconds): Addiert Millisekunden zu einem Zeitstempel.
10. datetime_to_combined_int(dt): Konvertiert einen Zeitstempel in ein kombiniertes Integer-Format.
11. combined_int_to_datetime(hours_minutes_combined, seconds_milliseconds_combined): Konvertiert ein kombiniertes Integer-Format in einen Zeitstempel.

3.4. arduino_interface

3.4.1. Klasse: Serial_Arduino

Diese Klasse ist für die Kommunikation mit einem Arduino über die serielle Schnittstelle zuständig.

Methoden:

1. *init*(self): Der Konstruktor initialisiert die serielle Verbindung mit dem Arduino und wartet, bis die Verbindung hergestellt ist.
2. find_arduino_port(self): Diese Methode sucht automatisch nach einem angeschlossenen Arduino und gibt den entsprechenden Port zurück.
3. write(self, data): Diese Methode sendet Daten an das Arduino über die serielle Verbindung.
4. read(self): Diese Methode liest eine Zeile von Daten vom Arduino.

3.4.2. Klasse: ArduinoInterface

Diese Klasse repräsentiert einen ROS-Node, die Motorbefehle empfängt und an ein Arduino sendet.

Methoden:

1. *init*(self): Der Konstruktor initialisiert den ROS-Node und das Serial_Arduino-Objekt.
2. listener_callback(self, msg): Diese Methode wird aufgerufen, wenn Motorbefehle empfangen werden. Sie sendet die empfangenen Befehle an das Arduino und erwartet eine Antwort.

3.5. web_control_center

3.5.1. Klasse: WebControlCenter

Diese Klasse ist ein ROS-Node, die Bilder von einem ROS-Topic empfängt und über eine Flask-App streamt.

Methoden:

1. *init*(self): Der Konstruktor initialisiert die ROS-Node und abonniert das Bildtopic. Sie erstellt auch einen Publisher für Steuerbefehle.
2. image_callback(self, msg): Diese Methode wird aufgerufen, wenn ein Bild empfangen wird. Sie konvertiert das Bild in das JPEG-Format und speichert es im Flask-App-Objekt.

3.5.2. Flask-App:

Die Flask-App wird verwendet, um ein Video-Stream und eine Benutzeroberfläche für die Steuerung anzuzeigen.

Methoden:

- /: Die Index route rendert die index.html-Seite.
- /video_feed: Diese Route sendet den Video-Stream.
- /button_click/<button_name>: Diese Route wird aufgerufen, wenn ein Steuerungsknopf gedrückt wird. Sie veröffentlicht den entsprechenden Steuerbefehl.
- /print_message/<message>: Diese Route sendet eine Nachricht über SocketIO.

3.5.3. Weitere Funktionen:

- generate(): Diese Funktion generiert den Video-Stream.
- launch_follow_me(): Diese Funktion startet ein ROS2-Launch-File für die Funktion "Follow Me".

3.6. Motion_Driver

3.6.1. Klasse MotionDriver

Diese Klasse ist eine ROS2-Node welche Daten vom /motor-Topic empfängt um die Motorgeschwindigkeiten des Alphabot2-Roboters entsprechend anzupassen.

Methoden:

1. *init*(self): Initialisiert den Node und erstellt eine Subscription für die Motorgeschwindigkeiten.
2. _clip(value, minimum, maximum): Diese Funktion begrenzt einen Wert innerhalb eines angegebenen Bereichs. value: Der zu begrenzende Wert. Minimum: Der minimale erlaubte Wert. Maximum: Der maximale erlaubte Wert.

3. move_PWMA(self, speed_percent): Diese Funktion wird verwendet, um die Geschwindigkeit des linken Motors einzustellen.
4. move_PWMB(self, speed_percent) Diese Funktion ist ähnlich wie move_PWMA, wird jedoch verwendet, um die Geschwindigkeit des rechten Motors einzustellen.
5. set_motor_speeds(self, left_speed, right_speed, duration): Setzt die Geschwindigkeiten der linken und rechten Motoren für eine bestimmte Dauer.
6. rpm_to_percent(self, rpm): Konvertiert RPM in Prozent.
7. *del*(self): Räumt die GPIO-Pins auf.

3.6.2. Klasse DCMotorController

Methoden:

1. *init*(self): Dies ist der Konstruktor für die DCMotorController-Klasse. Er initialisiert den Knoten mit dem Namen 'dc_motor_controller' und richtet die Parameter für den Knoten ein
2. motor_speeds_callback(self, msg): Callback-Funktion, die aufgerufen wird, wenn eine Nachricht auf dem /motor-Topic empfangen wird. Setzt die Motorgeschwindigkeiten.

3.7. Bilderkennung

3.7.1. mit YOLOv3

3.7.2. Klasse: HumanDetector

Die Klasse `HumanDetector` ermöglicht die Erkennung und Verfolgung von Personen in Videoframes mithilfe von YOLOv3.

Methoden:

1. *init*(self, show_frame=False):
   - Initialisiert die `HumanDetector`-Klasse mit notwendigen Attributen.
   - Lädt das YOLOv3-Netzwerk mit den Konfigurations- und Gewichtsdateien.
   - Initialisiert Layernamen, Boxen, und andere erforderliche Attribute.
   - `show_frame` gibt an, ob das verarbeitete Frame angezeigt werden soll.
2. locate_person(self, frame):
   - Ermittelt Personen in einem gegebenen Frame.
   - Führt die YOLO-Erkennung durch und zeichnet Boxen um erkannte Personen.
   - Bestimmt die am besten zentrierte Person im Bild und gibt ihre Position und Größe zurück.
   - Zeigt das verarbeitete Frame an, wenn `show_frame` True ist.
3. get_percentage_of_height(self, location, frame_height):
   - Berechnet den prozentualen Anteil der Höhe einer erkannten Person im Vergleich zur Frame-Höhe.

4. draw_coordinate_system(self, frame):
    - Zeichnet ein Koordinatensystem auf das Frame.
5. cv_to_custom_coordinates(self, x_cv, y_cv, frame_width, frame_height):
    - Konvertiert Koordinaten von OpenCV-Format in ein benutzerdefiniertes Format.
6. get_most_centered_person(self, frame_height, frame_width):
    - Bestimmt die am besten zentrierte Person im Frame basierend auf der Distanz zum Frame-Zentrum.

3.7.3. mit Hog Detector und Haar Cascade

3.7.4. Klasse: HumanDetector

Die Klasse `HumanDetector` ermöglicht die Erkennung und Verfolgung von Personen in Videoframes mithilfe von Haar-Cascade und MIL-Tracker.

Methoden:

1. *init*(self, show_frame=False):
    - Initialisiert die `HumanDetector`-Klasse mit notwendigen Attributen.
    - Lädt den Haar-Cascade-Klassifikator für die Ganzkörpersuche.
    - Initialisiert den MIL-Tracker und andere erforderliche Attribute.
    - `show_frame` gibt an, ob das verarbeitete Frame angezeigt werden soll.
2. locate_person(self, frame):
    - Ermittelt Personen in einem gegebenen Frame.
    - Verarbeitet das Frame, um Personen zu erkennen und zu verfolgen.
    - Gibt Informationen über die erkannte Person zurück, wenn diese verfolgt wird.
3. process_frame(self, frame):
    - Verarbeitet jedes Frame, um Personen zu erkennen und zu verfolgen.
    - Aktualisiert den Tracker und visualisiert die erkannte Person im Frame.
4. select_human(self, frame, humans):
    - Wählt die erste erkannte Person zur Verfolgung aus.
    - Initialisiert den MIL-Tracker für die ausgewählte Person.
5. get_percentage_of_height(self, location, frame_height):
    - Berechnet den prozentualen Anteil der Höhe einer erkannten Person im Vergleich zur Frame-Höhe.
6. draw_coordinate_system(self, frame):
    - Zeichnet ein Koordinatensystem auf das Frame.
7. cv_to_custom_coordinates(self, x_cv, y_cv, frame_width, frame_height):

- Konvertiert Koordinaten von OpenCV-Format in ein benutzerdefiniertes Format.

4. Design

4.1. Entwurf 1

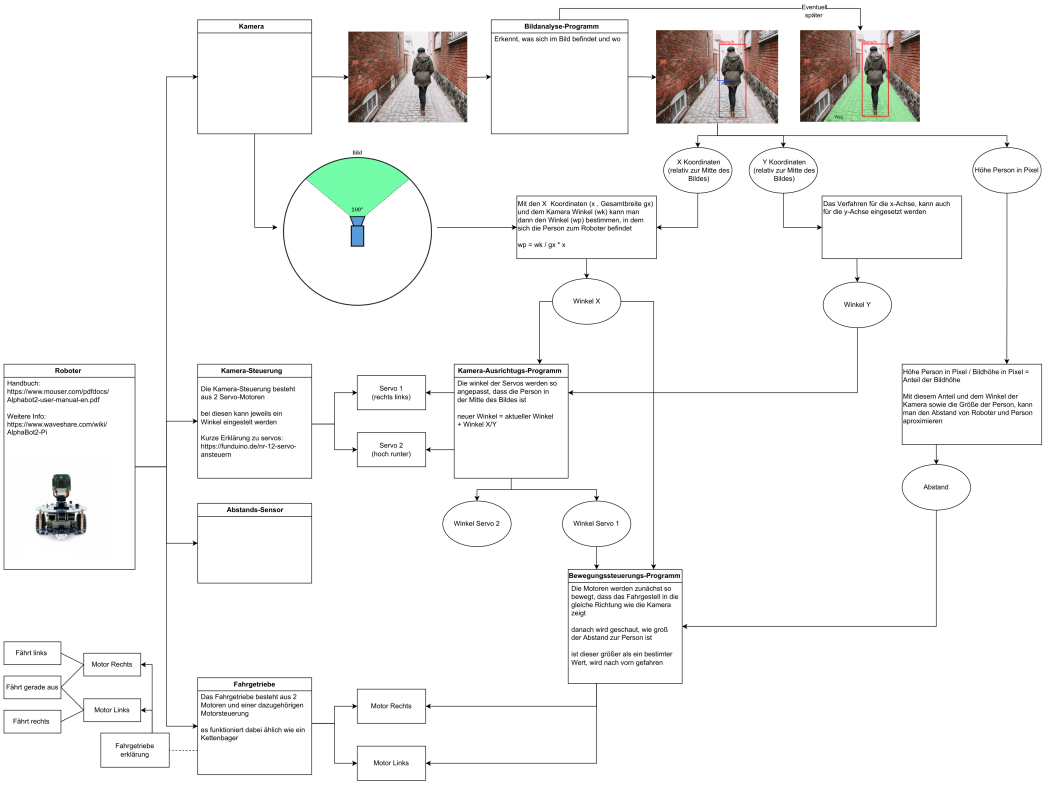

Figure 6. Basiskonzept

Der erste Projektentwurf entstand noch vor dem ersten Kundenmeeting und diente dabei als Vorbereitung. Hierbei ging es im Allgemeinen um ein Basiskonzept und die bessere Visualisierung für den Auftraggeber.

4.2. Entwurf 2

4.2.1. Vorgaben

- Verwendung von ROS
  - Mit ROS ist die grundlegende Struktur unserer Software festgelegt. ROS 2 ist in einem Paket-System strukturiert, welches aus mehreren Nodes besteht (den eigentlichen Softwarekomponenten).
- Verwendung des Alphabot 2
  - Die Verwendung des Alphabots bringt vor allem Hardware-Anforderungen mit sich

4.2.2. Technische Entscheidung

- Verwendung eines vorgefertigten Linux-Images (mit ROS vorinstalliert)
  - Image: Ubiquity Robotics Raspberry Pi Image https://learn.ubiquityrobotics.com/noetic_pi_image_downloads
  - Pro:
    - die Verwendung eines Images spart viel Zeit beim initialen Setup des Systems
    - ROS funktioniert out of the box
    - Tests können schneller durchgeführt werden
  - Contra:
    - weniger Anpassungsmöglichkeiten
- Verwendung des ROS for Waveshare Alphabot2 Repository https://github.com/ShaunPrice/ROS_for_Waveshare_Alphabot2
  - Pro:
    - das GitHub-Repository ist voll entwickelt und kompatibel mit unserem Linux-Image
    - es ist speziell für den Alphabot geschrieben
  - Contra:
    - es ist mittlerweile 5 Jahre alt

Dieser Entwurf entstand im Rahmen der Testphase und so sind noch keine konkreteren Strukturen entstanden.

4.3. Entwurf 3

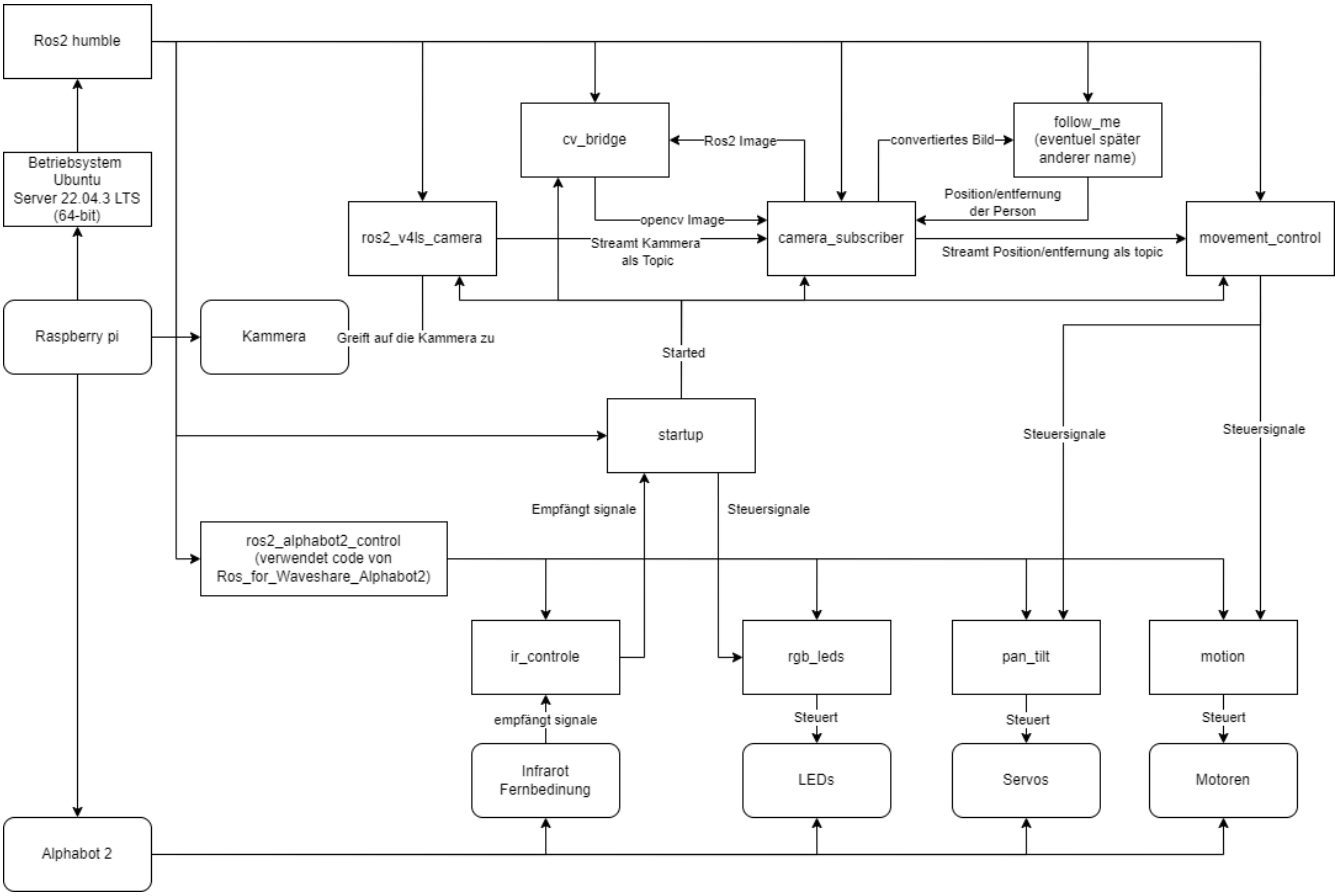

Figure 7. Systemarchitekturdiagramm für Entwurf 3

Abbildung entspricht keinen Modellierungsstandards und ist eher als Vorentwurf zu betrachten.

4.3.1. Änderung zum letzten Entwurf

- Das ROS für Waveshare Alphabot2 Repository kann in der aktuellen Form nicht verwendet werden, da es für ROS 1 geschrieben ist, welches auf Python 2 basiert. Das Problem mit Python 2 ist, dass es bald nicht mehr unterstützt wird. Dadurch sind viele modernere Features nicht verfügbar - somit ein Umstieg auf ROS 2 nötig.
- Das Linux-Image wird nicht mehr verwendet, da es auf einer veralteten Linux-Version basiert, welche nicht mit ROS 2 kompatibel sind

4.3.2. Technische Entscheidung

- Verwendung von ROS 2 Humble
  - Pro:
    - ROS 2 Humble ist eine der neuesten ROS 2 Versionen
    - es gibt eine Vielzahl an Software
    - viele Systeme unterstützen ROS 2
    - umfangreiche Dokumentation
  - Contra:
    - ROS 2 ist im Allgemeinen recht komplex und hat daher eine lange Einarbeitungszeit

- Verwendung von Ubuntu Server 22.04.03 LTS (64 Bit)
  - Pro:
    - das zu ROS 2 Humble empfohlene Operating-System
    - es ist kompatibel mit einer Vielzahl an Software
    - eine der aktuellsten Ubuntu-Versionen
  - Contra:
    - es hat keine grafische Benutzeroberfläche
- Verwendung von Python als Programmiersprache
  - Pro:
    - Python ist eine von zwei Sprachen, die mit ROS 2 kompatibel sind und deutlich einfacher zu implementieren als die Alternative C++
  - Contra:
    - mit Python kann man in ROS 2 keine eigenen Nachrichtenformate erstellen und ist daher an die Standardformate gebunden
- Übersetzung des ROS for Waveshare Alphabot2 Repository von ROS 1 in ROS 2
  - Pro:
    - das Repository liefert eine gute Grundlage, um zu verstehen, wie ROS Systeme allgemein aufgebaut sind
    - es enthält eine Vielzahl von Informationen speziell zum Alphabot2
  - Contra:
    - Teilweise zu komplex für unseren Anwendungsfall
- Verwendung der ROS2_v4ls_camera https://github.com/tier4/ros2_v4l2_camera/tree/galactic Camera Node
  - es wurde eine Vielzahl an Kamera-Nodes ausprobiert, die meisten sind jedoch für Raspbian geschrieben und funktionieren nicht mit unserem System
  - Pro:
    - funktioniert mit dem System
    - ist einfach zu installieren
  - Contra:
    - schwer zu konfigurieren
- Verwendung von CV Bridge
  - Pro:
    - ermöglicht die einfache Umwandlung vom ROS 2-Image-Format in das OpenCV-Image-Format

4.3.3. Strukturelle Entscheidungen

- das Modell zeigt den allgemeinen Aufbau des Systems.
- es gibt eine Node für jede Hardwarekomponente des Alphabot 2, welche mittels Messages angesteuert werden kann.
- des Weiteren gibt es die camera_subscriber_node, welche das empfangen und Auswerten der Bilder übernimmt.
- ebenfalls die movement_control-Node, welche die ausgewerteten Daten empfängt und in Signale für die Nodes umwandelt, die die Hardware-Komponenten steuern.

4.4. Entwurf 4

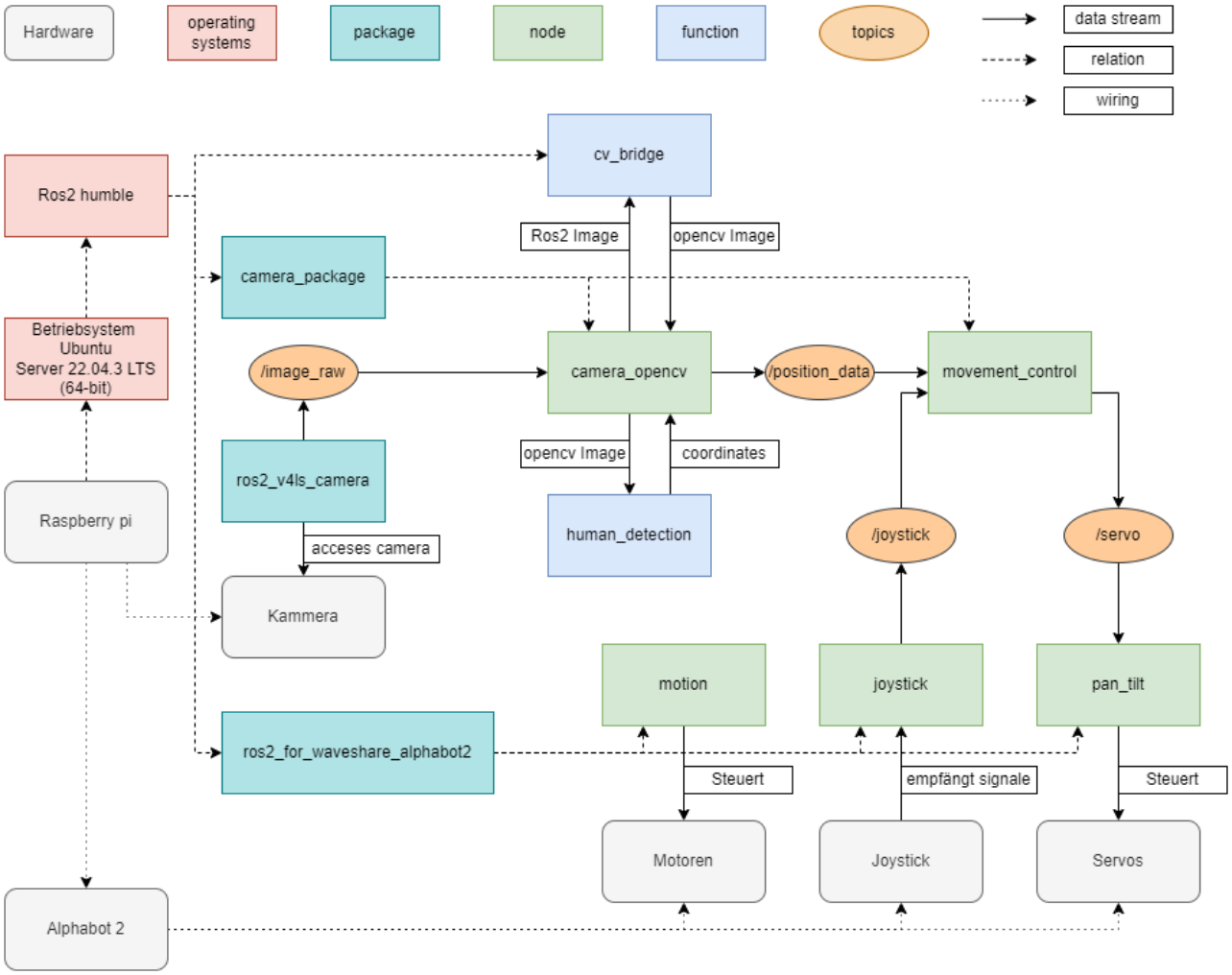

Figure 8. Systemarchitekturdiagramm für Entwurf 4

Entwurf 4 ist der erste funktionale Entwurf und auch der erste Entwurf mit einem Prototyp. Abbildung entspricht keinen Modellierungsstandards und ist eher als Vorentwurf zu betrachten.

4.4.1. Änderung zum letzten Entwurf

- leichte Änderung des strukturellen Aufbaus

4.4.2. Strukturelle Entscheidungen

- Aufteilung des Systems in zwei Pakete.
  - diese Entscheidung wurde getroffen, um das System möglichst modular zu gestalten.
  - das ros2_for_waveshare-Paket ist speziell für den Alphabot 2 geschrieben und stellt somit eine Art Update des ROS for Waveshare Alphabot 2-Repositories dar. Die Idee ist, dass das Paket unabhängig von unserem System mit dem Alphabot 2 verwendet werden kann.
  - das camera_package enthält alle Tools zur Bildverarbeitung und Berechnung der Eingangssignale. Da es unabhängig vom ersten Paket funktioniert, könnte man in der Zukunft z. B. recht einfach auf eine andere Plattform umsteigen, ohne den Code stark zu modifizieren.

4.5. Entwurf 5

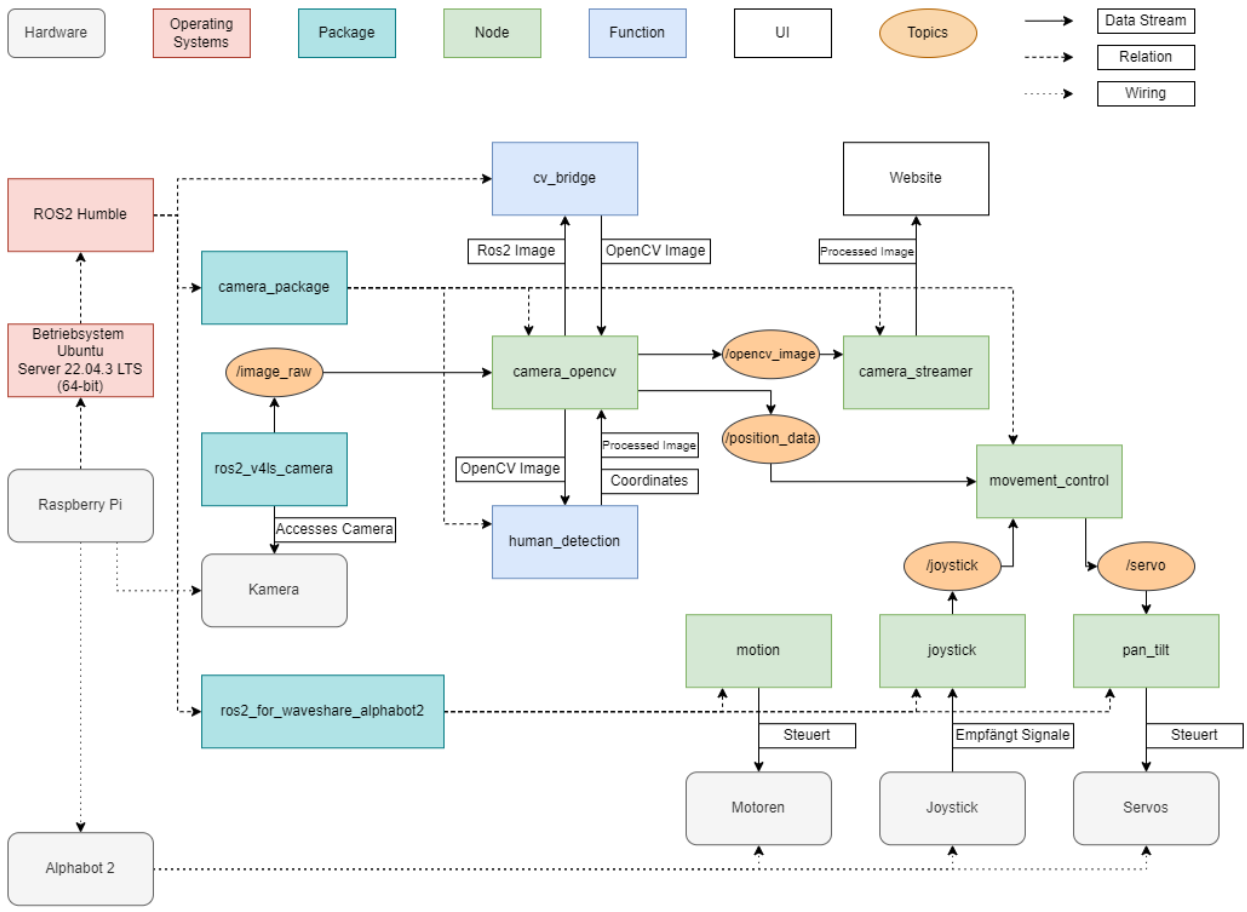

Figure 9. Systemarchitekturdiagramm für Entwurf 5

Abbildung entspricht keinen Modellierungsstandards und ist eher als Vorentwurf zu betrachten.

4.5.1. Technische Entscheidung

- Hinzufügen einer Web-Oberfläche, die die vom human_detector bearbeiteten Bilder anzeigt
  - Pro:

        - das Tool ermöglicht es zu sehen, wie gut das Tracking funktioniert und ist somit unbedingt notwendig für das Debugging.
    - Contra:
        - Performanceverlust
- Verwendung von Flask für das Web-Tool
    - Pro:
        - relativ einfache Implementierung in Python
    - Contra:
        - teilweise Kompatibilitätsprobleme mit ROS 2
        - muss in einem separaten Thread laufen, da es sonst Probleme mit ROS 2 gibt
        - erhöhter Performancegebrauch durch Threading

Durch die Implementierung des camera_streamers war es deutlich einfacher zu verstehen, wie gut die Erkennung funktioniert. Somit ist uns auch ein großes Problem aufgefallen: Die bisher verwendete Kamera hat einen viel zu geringen Winkel für unseren Anwendungsfall, da Personen ungefähr drei Meter vom Roboter entfernt stehen müssen, um überhaupt vollständig im Bild erkannt zu werden. Zudem ist der Bilderkennungsalgorithmus, den wir verwenden, recht ungenau und erkennt Personen entweder nicht oder erkennt Personen in Gegenständen.

4.6. Entwurf 6

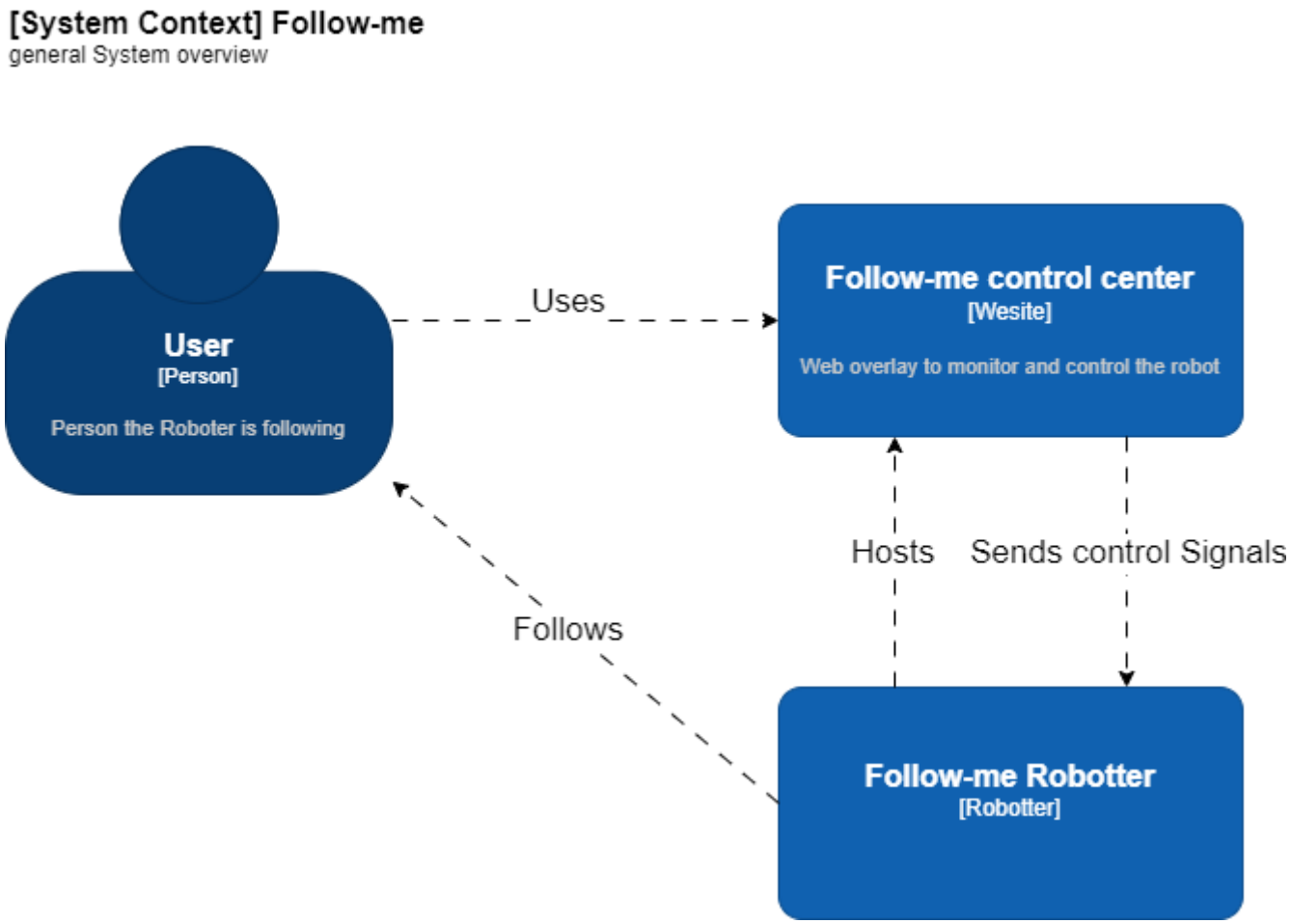

Figure 10. C4-Context Entwurf 6

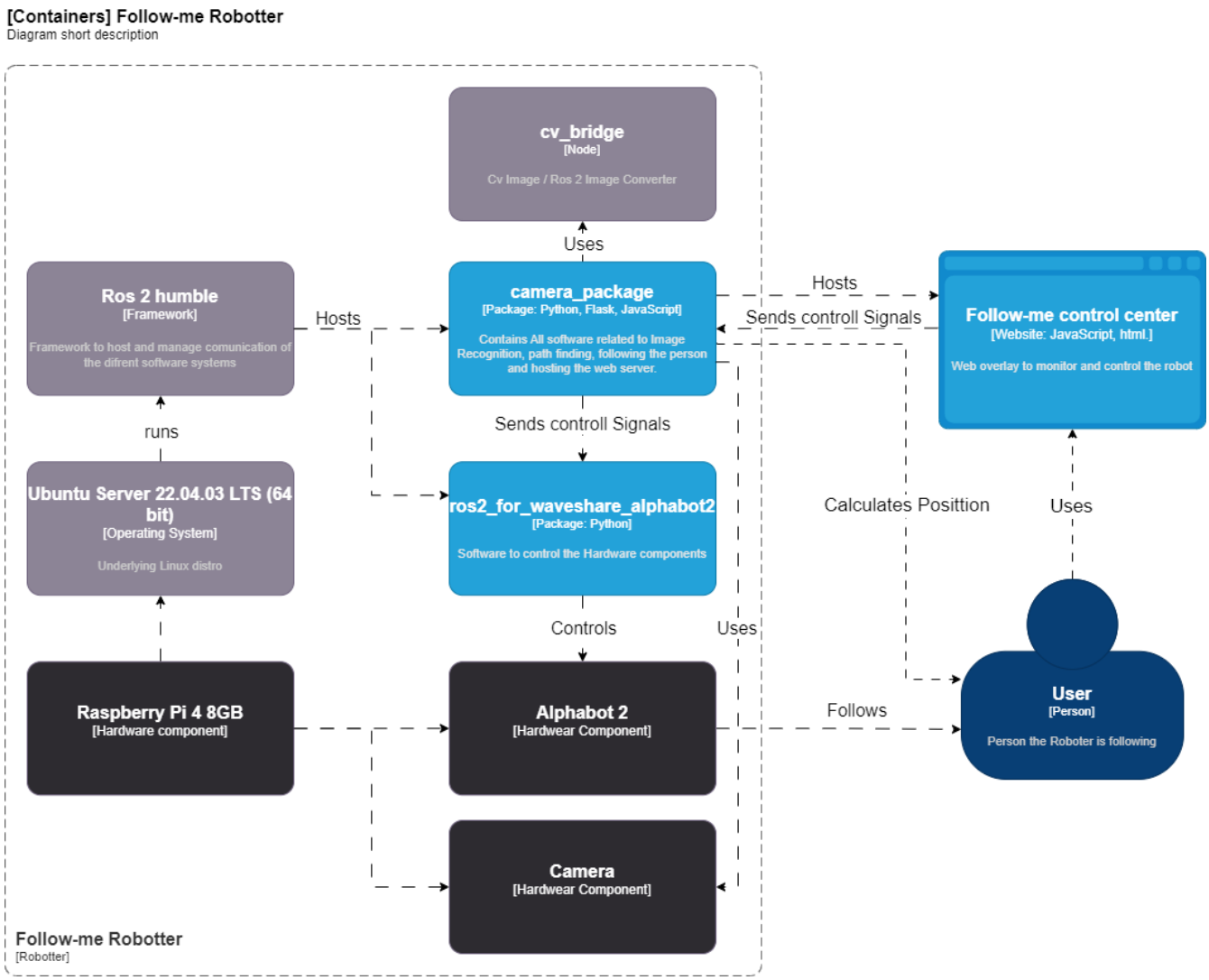

Figure 11. C4-Container Entwurf 6

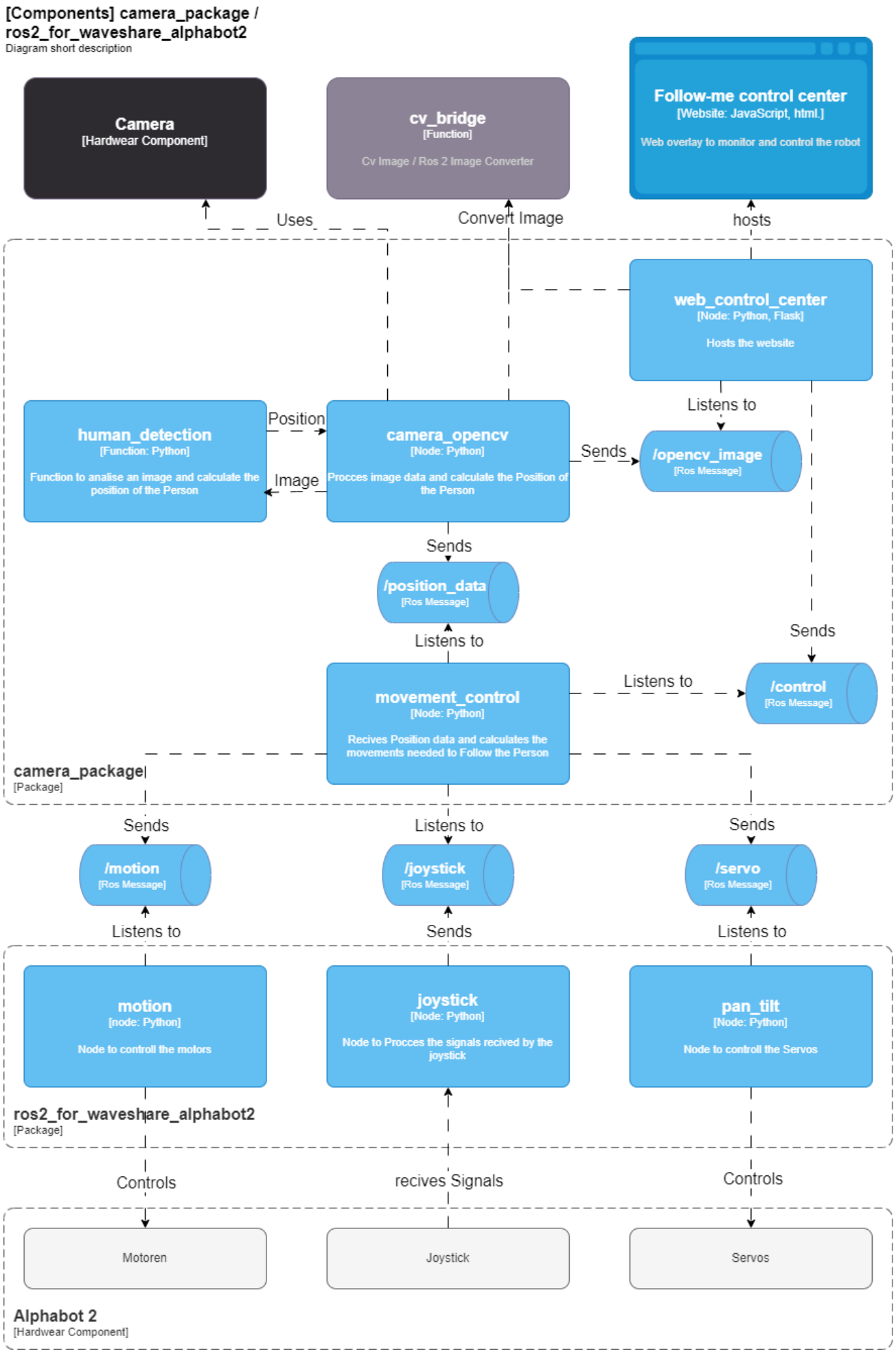

Figure 12. C4-Component Entwurf 6

4.6.1. Änderung zum letzten Entwurf

- Entfernung der ros2_v4ls_camera-Node
- Austausch der vorinstallierten Kamera auf dem Alphabot2 durch eine USB-Kamera
  - Da die vorinstallierte Kamera nicht für unsere Zwecke ausreicht
- Entfernung der Servos
  - die neue Kamera ist zu schwer für die Servos; die dafür gebaute Software bleibt trotzdem im Projekt für eine eventuell spätere Benutzung.

4.6.2. Technische Entscheidung

- Wechsel zu einer USB-Kamera
  - Pro:
    - bessere Qualität und ein deutlich größerer Winkel
  - Contra:
    - deutlich schwerer, deshalb Entfernung der Servos
    - höherer Stromverbrauch
- Wechsel zur OpenCV-Video-Stream-Capture-Funktion
  - Pro:
    - direktes Ansprechen der Kamera in Python möglich
  - Contra:
    - capturing findet permanent statt und kommt somit mit einem gewissen Maß an Performanceverbrauch
    - die Kamera kann nur im Rahmen einer Node verwendet werden
- Wechsel zu YOLO
  - Pro:
    - bessere Erkennungsgenauigkeit
  - Contra:
    - hoher Performanceverbrauch
    - recht langsam

Zum aktuellen Zeitpunkt ist noch nicht klar, ob wir YOLO einsetzen können, da es derzeit viel zu langsam ist - die aktuelle Tendenz liegt bei nein.

4.7. Entwurf 7 - Prototyp 2

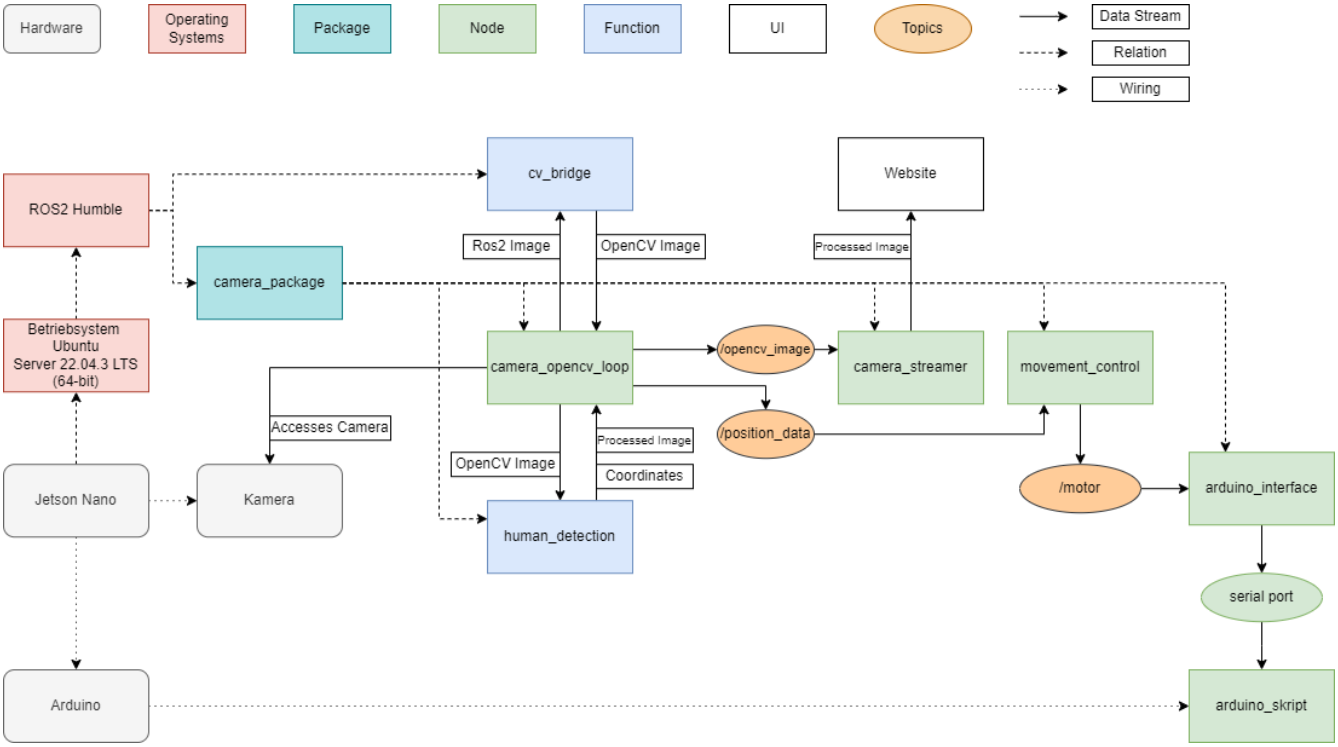

Figure 13. Systemarchitekturdiagramm für Entwurf 7

Abbildung entspricht keinen Modellierungsstandards und ist eher als Vorentwurf zu betrachten.

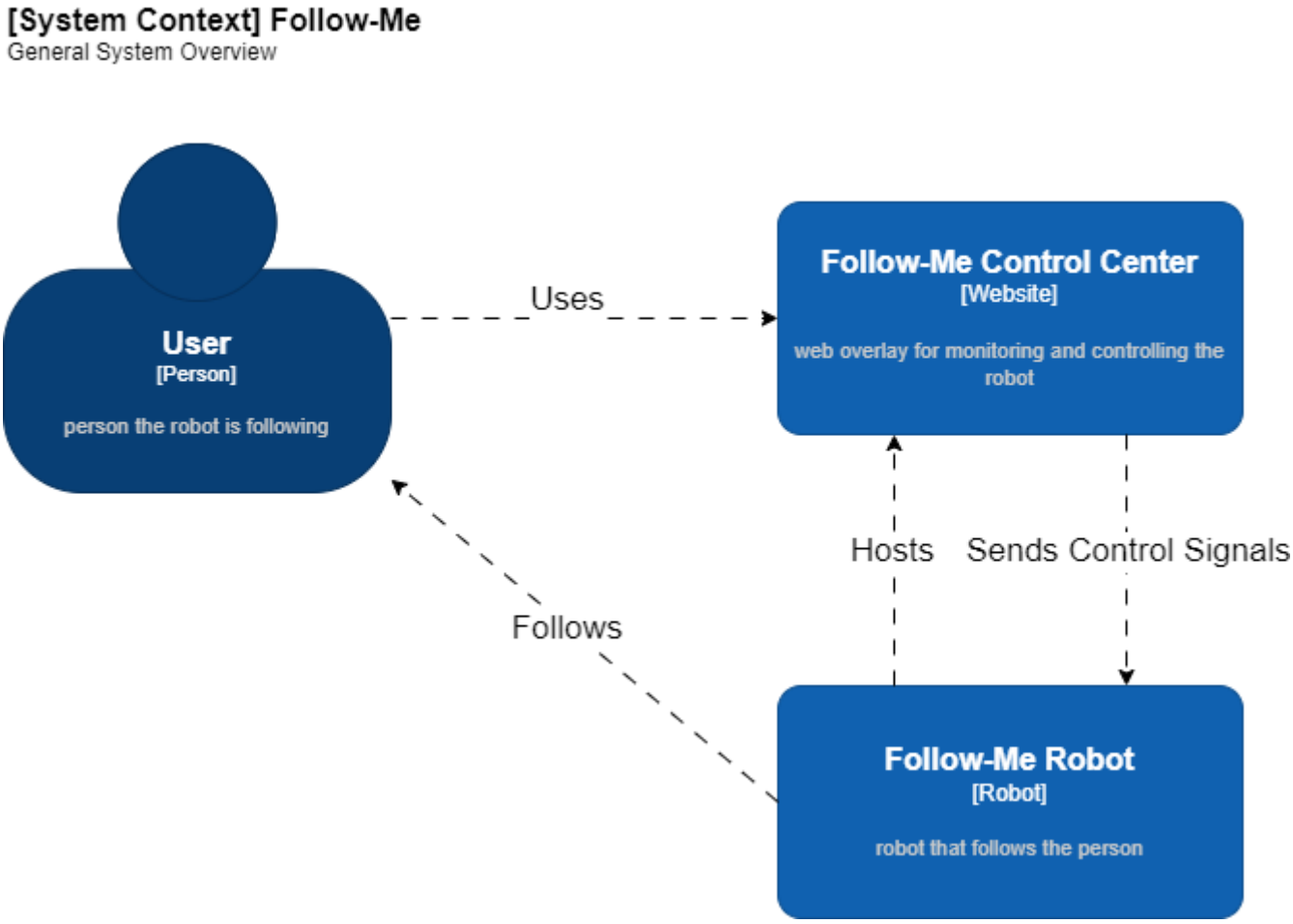

Figure 14. C4-Context Entwurf 7

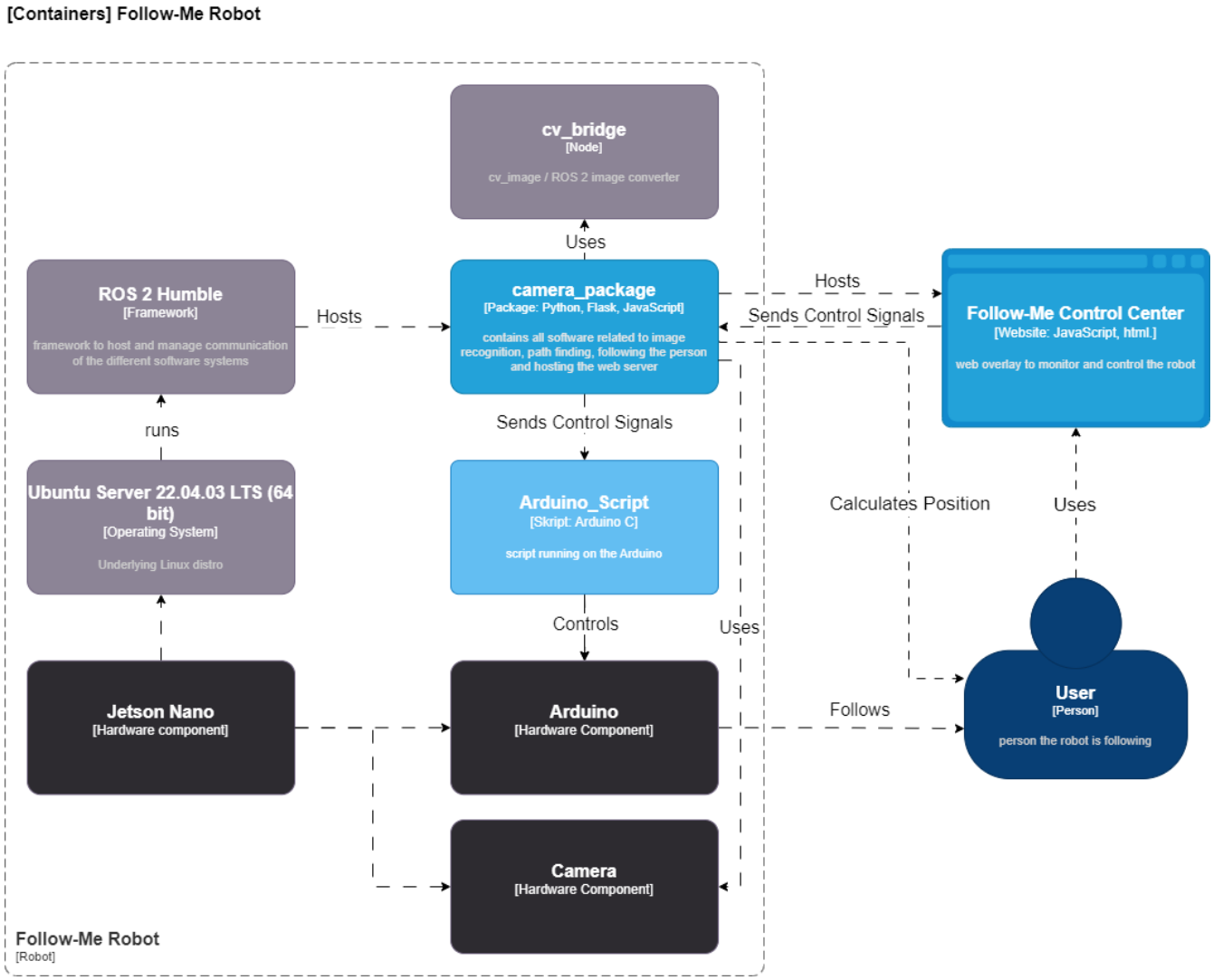

Figure 15. C4-Container Entwurf 7

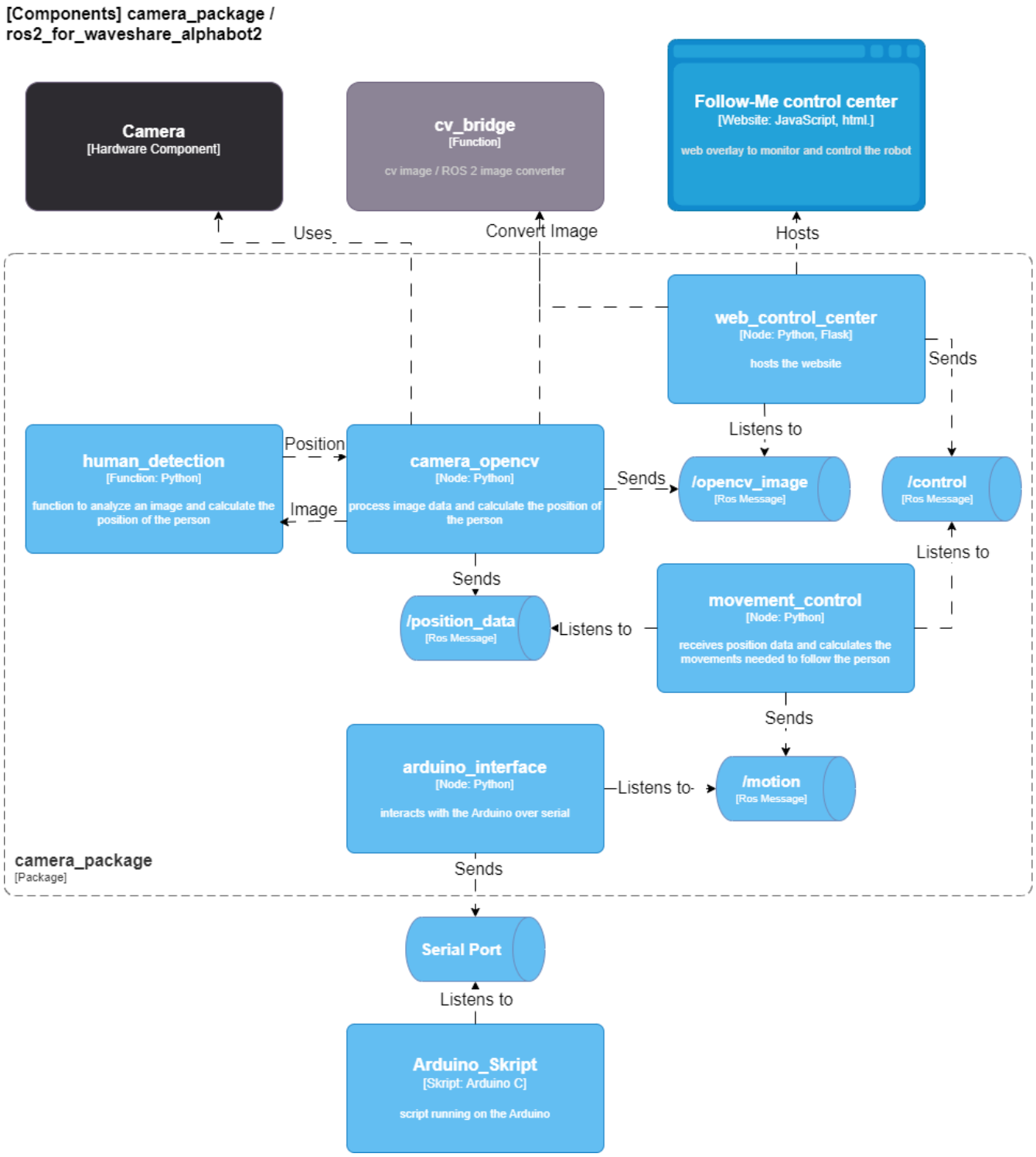

Figure 16. C4-Component Entwurf 7

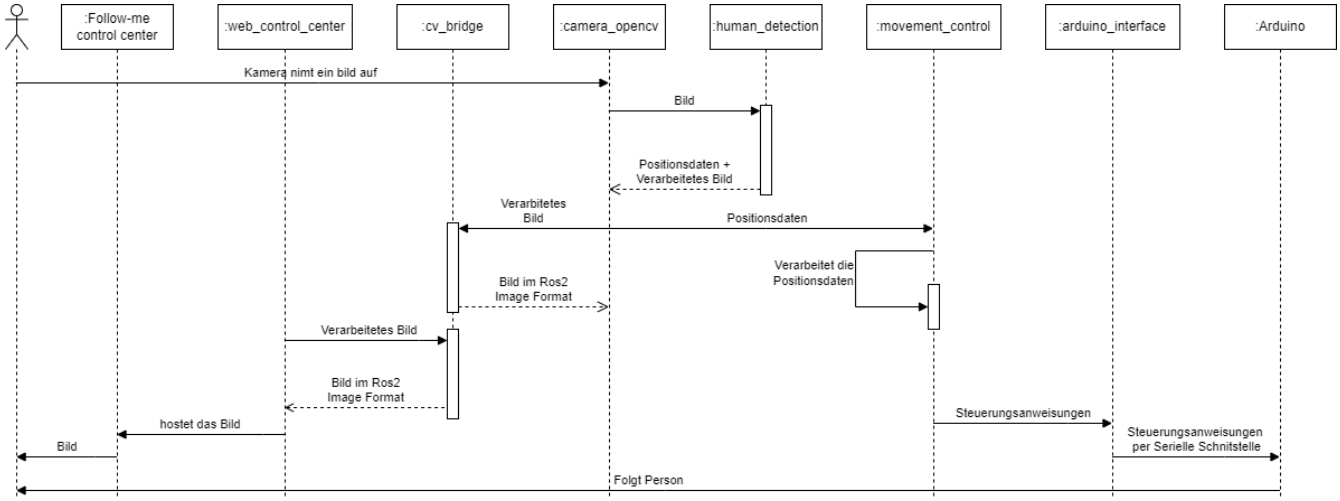

Figure 17. Sequenzdiagramm Entwurf 7

4.7.1. Änderung zum letzten Entwurf

- Wechsel von Raspberry Pi 4 auf Nvidia Jetson Nano **der Raspberry Pi 4 hat nicht genug Leistung für YOLO geboten
- Wechsel von Alphabot 2 auf Arduino Uno und Adafruit Motor Shield v2.3
  - der Alphabot 2 ist zu klein, um den Jetson Nano zu tragen
  - der Alphabot 2 ist schlecht erweiterbar
- Entfernung des ros2_for_waveshare_alphabot2-Pakets
  - wird nicht mehr benötigt
- Wechsel zum Jetson Nano - Ubuntu 20.04-Image
  - der Jetson Nano hat aktuell keine offizielle Unterstützung für Ubuntu 22.04

4.7.2. Technische Entscheidung

- Wechsel zum Nvidia Jetson Nano
  - Pro:
    - deutlich mehr Leistung als der Raspberry Pi 4
    - bessere Unterstützung für YOLO
  - Contra:
    - höherer Stromverbrauch
    - höheres Gewicht
- Wechsel zum Arduino Uno und Adafruit Motor Shield v2.3
  - Pro:
    - bessere Erweiterbarkeit
    - deutlich einfachere Motorensteuerung
    - Testung ohne den Jetson Nano möglich, über serielle Schnittstelle
  - Contra:

        - Kommunikation muss über serielle Schnittstellen stattfinden
        - komplexere Systemstruktur
- Wechsel auf Jetson Nano - Ubuntu 20.04 image
    - Pro:
        - offizielle Unterstützung
        - bessere Kompatibilität
    - Contra:
        - ältere Ubuntu Version
        - weniger Software verfügbar

5. Formeln zur Berechnung von Steuersignalen aus Bilddaten

5.1. Berechnung

5.1.1. Winkel zur Person

Der Winkel zur Person wird berechnet, indem man den Maximalwinkel der Kamera mit der Dimension des Bildes gleichsetzt und dann mittels Dreisatz den Winkel zur Person aus der Position im Bild berechnet.

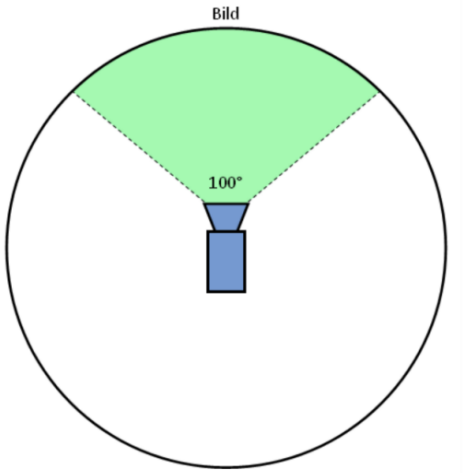

Figure 18. Visualisierung Kamerawinkel

$P \ldots Person$
$K \ldots Kammera$

Ges:
$P.wx \ldots Winkel\ zur\ Person\ in\ x\ richtung$
$P.wy \ldots Winkel\ zur\ Person\ in\ y\ richtung$

Geg:
$P.lx \ldots Länge\ der\ Person\ in\ x\ richtung$
$P.ly \ldots Länge\ der\ Person\ in\ y\ richtung$
$P.cx \ldots x\ Cordinate\ der\ Person$
$P.cy \ldots y\ Cordinate\ der\ Person$

$K.wx \ldots Maximalwinkel\ der\ Kammera\ in\ x\ richtung$
$K.wy \ldots Maximalwinkel\ der\ Kammera\ in\ y\ richtung$
$K.lx \ldots Länge\ des\ Bildes\ in\ x\ richtung$
$K.ly \ldots Länge\ des\ Bildes\ in\ y\ richtung$

Formel:

$$P.w = \frac{P.c}{K.l} * K.w$$

Figure 19. Winkel zur Personenberechnung

5.1.2. Geschwindigkeitsberechnung

Die Geschwindigkeit im Verhältnis zum ersten Rad wird berechnet, sodass der Roboter einen bestimmten Winkel fährt. Somit kann man den Roboter mithilfe von Winkeln steuern.

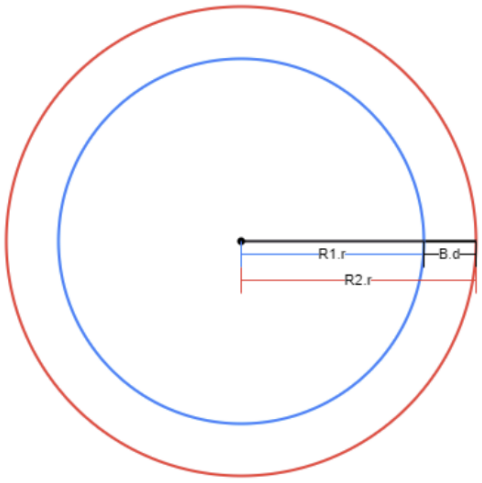

Figure 20. Visualisierung Geschwindigkeitsberechnung

$B.rpm \dots basis\ Geschwindigkeit\ in\ RPM$
$B.d \dots Radabstand\ in\ cm$
$B.r \dots Rad\ Radius\ in\ cm$
$B.v \dots basis\ Geschwindigkeit\ in \frac{cm}{s}$

$R1.r \dots Radius\ Kreis\ 1$
$R2.r \dots Radius\ Kreis\ 2$
$R1.u \dots Umfang\ Kreis\ 1$
$R2.u \dots Umfang\ Kreis\ 2$
$R1.ut \dots teil\ Umfang\ Kreis\ 1$
$R2.ut \dots teil\ Umfang\ Kreis\ 2$
$R1.v \dots Geschwindigkeit\ Rad\ 1\ in \frac{cm}{s}$
$R2.v \dots Geschwindigkeit\ Rad\ 2\ in \frac{cm}{s}$
$R1.rpm \dots Geschwindigkeit\ Rad\ 1\ in\ rpm$
$R1.rpm \dots Geschwindigkeit\ Rad\ 2\ in\ rpm$
$R.w \dots Winkel$
$R.t \dots Bewegumgsdauer\ in\ s$

Figure 21. Geschwindigkeitsberechnung 1

Geg:
$B.rpm$
$B.d$
$B.r$
$R1.r$
$R.w$

Ges:
$R2.rpm$
$R.t$

Grundlagen:

$R1.u = 2\pi * R1.r \dots Kreisformel$
$R1.u = R1.v * R.t$

$$R1.r = \frac{R1.v * R.t}{2\pi}$$
$$R2.v = \frac{2\pi * (R1.r + B.d)}{R.t}$$

$$R2.v = \frac{2\pi * \left(\frac{R1.v * R.t}{2\pi} + B.d\right)}{R.t}$$

Es existiert Abhängigkeit zwischen R1.v und R2.v über den Radabstand

Figure 22. Geschwindigkeitsberechnung 2

Formeln:

1. *Umrechnung in cm/s*

$$B.v = \frac{2 * \pi * B.r}{60} * B.rpm$$

2. *Teilumfang Berechnen*

$$R1.ut = 2 * \pi * R1.r * \frac{R.w}{360}$$

$$R2.r = R1.r + B.d$$
$$R2.ut = 2 * \pi * R2.r * \frac{R.w}{360}$$

3. *Bewegungsdauer Berechnen*

$$R.t = \frac{R1.ut}{B.v}$$

4. *Rad Geschwindigkeit Berechnen*

$$R1.rpm = B.rpm$$

$$R2.rpm\ (R2.ut = 0) = B.rpm$$
$$R2.rpm\ (R2.ut \neq 0) = \frac{R2.ut}{R.t} * \frac{1}{0{,}1047 * B.r}$$

Figure 23. Geschwindigkeitsberechnung 3