

# **Loadlifter**

## **Ein autonomer Lagerroboter**

**Maturaarbeit**

Daniel Würmli



Betreut durch Sven Nüesch

Informatik

Kantonsschule Frauenfeld

20.10.2025



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung.....</b>	<b>1</b>
<b>2</b>	<b>Theorieteil.....</b>	<b>2</b>
2.1	Robotik Grundlagen.....	2
2.1.1	Was ist ein Roboter .....	2
2.1.2	Autonome Roboter.....	2
2.2	Sensoren.....	4
2.2.1	LiDAR-Sensoren .....	4
2.2.2	Kamera (visuelle Sensorik).....	6
2.2.3	Ultraschallsensoren .....	7
2.3	Aktuatoren.....	8
2.3.1	Motoren .....	8
2.3.2	Servomotoren .....	9
2.4	Programmiersprachen in eingebetteten Systemen.....	9
2.4.1	C++.....	10
2.4.2	Python .....	10
2.5	Bibliotheken .....	12
2.5.1	OpenCV (cv2) .....	12
2.5.2	ultralytics (YOLO, Python).....	12
2.5.3	threading (Python) .....	13
2.5.4	sys (Python) .....	14
2.5.5	time (Python).....	14
2.5.6	argparse (Python) .....	15
2.5.7	math (Python) .....	15
2.5.8	pathlib (Python).....	16
2.5.9	termios (Python).....	17
2.5.10	flask (Python) .....	17
2.5.11	smbus2 (Python).....	18
2.6	Frameworks und Middleware .....	18
2.6.1	ROS 2 .....	19
2.6.2	Orocoss.....	19
<b>3</b>	<b>Analyse .....</b>	<b>20</b>
3.1	Namenswahl .....	20
3.2	Projektauftrag.....	20
3.2.1	Must-Haves .....	20
3.2.2	Nice-to-Haves .....	21
3.2.3	Abgrenzung.....	22
3.3	Recherche.....	22
3.3.1	Hardware-Recherche.....	22
3.3.2	Software-Stack-Recherche.....	24

3.4	Zielgruppe & Einsatzumgebung .....	26
3.4.1	Zielgruppe .....	26
3.4.2	Stakeholder.....	26
3.4.3	Einsatzumgebung (Lagermodell) .....	27
3.4.4	Warum diese Umgebung?.....	28
3.5	IST-Ausstattung .....	29
3.5.1	Hardware (IST) .....	29
3.5.2	Software (IST).....	29
3.6	Use-Cases .....	30
3.6.1	UC-01: Autonom auf vordefiniertem Pfad bewegen .....	30
3.6.2	UC-02: Objekt greifen und rechts ablegen (reine Arm-Sequenz).....	31
3.6.3	UC-03: Zielobjekt erkennen (Kamera/ML) .....	32
3.6.4	UC-04: Zielobjekt während der autonomen Fahrt erkennen, greifen und zur Basis transportieren & ablegen .....	33
3.6.5	UC-05: Live-Überwachung .....	33
3.7	Eigene Entscheide.....	34
3.7.1	Sensorwahl: 2D-LiDAR statt Ultraschall oder 3D-LiDAR.....	34
3.7.2	Technologie-Stack: Python statt C++.....	35
3.7.3	Technologie-Stack: Bare Python vs. ROS 2 (mit Python) .....	36
3.7.4	Kameraauswahl: USB-UVC statt MIPI/CS .....	37
3.7.5	Objekterkennung: YOLOv8n .....	37
3.7.6	DIY-Montage & Verkabelung.....	37
3.8	Test-Snippets .....	38
3.9	Aufwand, Zeitplanung & Kostenschätzung .....	47
<b>4</b>	<b>Design .....</b>	<b>48</b>
4.1	Systemübersicht .....	48
4.2	Control-Ebene.....	48
4.3	High-Level-Ebene .....	49
4.4	Low-Level-Ebene und I/O-Abstraktion .....	51
4.5	Peripherie-Subsysteme .....	52
4.6	Sequenzdiagramme.....	52
4.6.1	FollowRoute .....	52
4.6.2	DefinedRouteGetObjectTop .....	54
<b>5</b>	<b>Implementation .....</b>	<b>55</b>
5.1	Log-Buch: Januar – Februar (Vorbereitung & erste Fahrtests) .....	55
5.2	Log-Buch: Mai – Juni (Hardware/Code festigen) .....	55
5.3	Log-Buch: Juli (Kalibrierung & LiDAR Wandfolge .....	56
5.4	Log-Buch: August (Lagermodellbau & erste Modi) .....	56
5.5	Log-Buch: September (Kamera läuft, Streaming über Flask).....	57
5.6	Log-Buch: Oktober (Machine-Learning: Training & Integration).....	57
<b>6</b>	<b>Diskussion .....</b>	<b>58</b>

6.1	Zielerreichung gegenüber der Analyse .....	58
6.2	Verifikation: Use-Cases .....	59
6.3	Grenzen, offene Punkte, Known-Bugs.....	59
6.4	Genauigkeit (praktisch beobachtet).....	59
<b>7</b>	<b>Inbetriebnahme .....</b>	<b>60</b>
7.1	Hardware.....	60
7.2	Software-Umgebung.....	60
7.3	Installation auf dem Raspberry Pi und PC/Laptop .....	60
<b>8</b>	<b>Persönliche Erkenntnisse.....</b>	<b>61</b>
<b>Tabellen- und Abbildungsverzeichnis .....</b>		<b>63</b>
Tabellen.....		63
Abbildungen .....		63
<b>Quellenverzeichnis .....</b>		<b>64</b>
<b>Hilfsmittelverzeichnis .....</b>		<b>66</b>
<b>Selbständigkeitserklärung .....</b>		<b>67</b>
<b>Anhang .....</b>		<b>68</b>

## 1 Einleitung

In einer zunehmend automatisierten Welt gewinnen autonome Systeme in Industrie und Alltag stark an Bedeutung. Besonders in der Intralogistik zeigt sich, dass Roboter längst nicht mehr nur einfache Handhabungsaufgaben übernehmen, sondern zunehmend selbstständig komplexe Abläufe ausführen und damit neue Anwendungsfelder erschließen. Ziel dieser Maturaarbeit ist die Entwicklung des autonomen Roboters Loadlifter, der sich in einer simulierten Lagerumgebung orientieren und definierte Transport- sowie Greifaufgaben zuverlässig bewältigen kann. Im Zentrum steht die Frage, wie ein mobiler Roboter so programmiert und ausgestattet werden kann, dass er sich in einer strukturierten Umgebung selbstständig zurechtfindet und seine Aufgaben zuverlässig und konsistent ausführt.

Die Aufgabenstellung beinhaltet folgende Schwerpunkte:

- Auswahl und Inbetriebnahme einer geeigneten Hardwareplattform (Hiwonder ArmPi Pro)
- Ansteuerung und Kalibrierung von Chassis, Roboterarm, Sensorik und Kamera
- Integration der Subsysteme zu einem autonomen Gesamtsystem
- Entwicklung eines zentralen Steuerungssystems in Python
- Test und Optimierung von Navigation und Objektinteraktion

Das angestrebte Einsatzszenario ist bewusst konkret: Loadlifter wartet in der Basisstation auf einen Auftrag, fährt anschließend den Hauptgang entlang in den Regalgang, identifiziert dort das gesuchte Objekt, greift es mit dem Arm und kehrt zur Basis zurück, wo das Objekt abgelegt wird.

Der Schwerpunkt der Arbeit liegt auf der Softwareentwicklung. Die Hardwarebasis, der Hiwonder ArmPi Pro, wurde als fertige Plattform übernommen. Montage, Verkabelung, DIY-Anpassungen wie die Befestigung von Kamera und LiDAR sowie die Inbetriebnahme benötigten dennoch einen substanzuellen Teil der Projektzeit. Diese Sorgfalt war notwendig, weil stabile Mechanik, eine zuverlässige Stromversorgung und saubere Schnittstellen die Voraussetzungen für reproduzierbare Softwaretests bilden. Obwohl die Hardware nicht den inhaltlichen Kern der Maturaarbeit ausmacht, ist sie als tragende Grundlage zu verstehen. Erst die solide Integration der Komponenten ermöglicht es, die autonome Steuerung programmatisch umzusetzen und die Funktionsfähigkeit im beschriebenen Szenario belastbar nachzuweisen.

## 2 Theorieteil

### 2.1 Robotik Grundlagen

#### 2.1.1 Was ist ein Roboter?

Ein Roboter ist ein programmierbares mechatronisches System, das Material, Werkstücke, Werkzeuge oder Spezialgeräte bewegen kann. Durch frei programmierbare Bewegungsabläufe ist er vielseitig einsetzbar und kann unterschiedliche Aufgaben übernehmen.<sup>1</sup>

Man unterscheidet grundsätzlich zwischen stationären und mobilen Robotern:

- Stationäre Roboter sind fest installiert, somit können sie ihren Standort nicht durch einen eigenen Antrieb wechseln. Sie sind für eindeutige und repetitive Arbeitsabläufe ausgelegt, beispielsweise beim Schweißen oder Montieren.<sup>2</sup>
- Mobile Roboter besitzen Aktuatoren, um ihre Position in der Umwelt zu verändern. Solche Roboter müssen in bekannten oder unbekannten Umgebungen navigieren können.<sup>3</sup>

#### 2.1.2 Autonome Roboter

Der Begriff «autonom» stammt aus dem Griechischen und setzt sich aus autos (selbst) und nomos (Gesetz) zusammen. Wörtlich bedeutet er also «sich selbst ein Gesetz gebend». Übertragen auf die Robotik bedeutet Autonomie, dass ein Roboter ohne externe Unterstützung in seiner Umgebung funktionieren kann. Ein autonomer Roboter ist somit nicht auf direkte Steuerung durch einen Menschen angewiesen, sondern trifft Entscheidungen basierend auf den Informationen, die er über Sensoren wahrnimmt.<sup>4</sup>

Die Autonomie von Robotern ist jedoch nicht absolut, sondern lässt sich in verschiedene Grade einteilen. Je nach Einsatzgebiet reicht sie von teilweiser Autonomie (z. B. mit vorprogrammierten Abläufen und eingeschränkter Entscheidungsfreiheit) bis hin zu hoher Autonomie, bei der komplexe Umgebungen selbstständig analysiert und auf unerwartete Situationen reagiert wird.<sup>5</sup>

---

<sup>1</sup> Vgl. Dr. Mohamed Oubbat, Robotik, Abschnitt 1.2.1

<sup>2</sup> Vgl. Dr. Mohamed Oubbat, Robotik, Abschnitt 1.3.1

<sup>3</sup> Vgl. Dr. Mohamed Oubbat, Robotik, Abschnitt 1.3.2

<sup>4</sup> Vgl. Dr. Mohamed Oubbat, Robotik, Abschnitt 1.2.2

<sup>5</sup> Vgl. Dr. Mohamed Oubbat, Robotik, Abschnitt 1.2.2

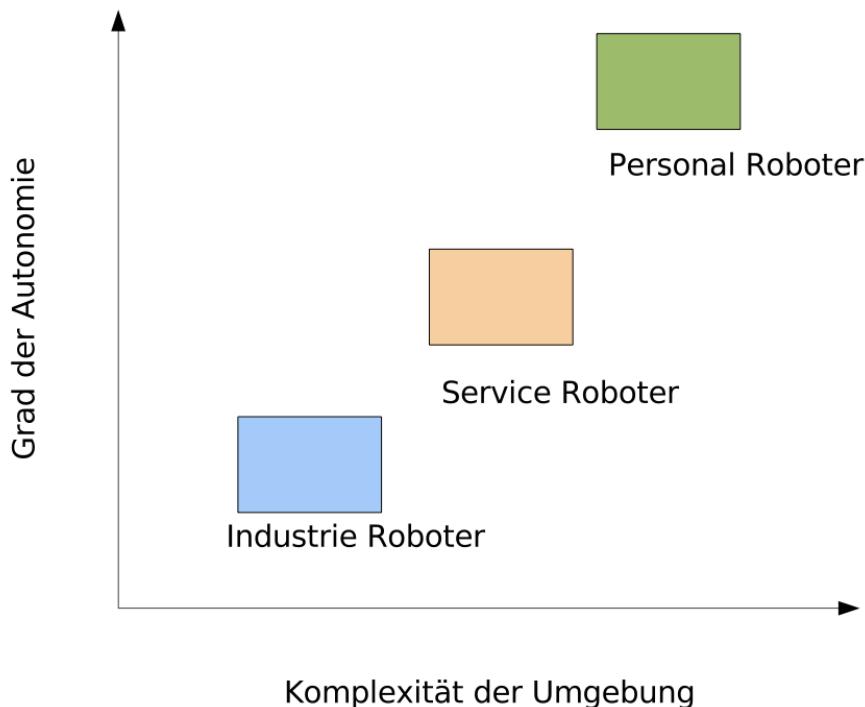


Abb. 1: Autonomie eines Roboters

Quelle: Oubbati, M. (2007). Robotik. Uni Ulm.

Die Abbildung 1 zeigt die Unterschiede verschiedener Robotertypen anhand ihres Autonomiegrades. Industrie Roboter, wie sie in Produktionsanlagen eingesetzt werden, besitzen in der Regel einen sehr geringen Grad an Autonomie, da sie für wiederholende, klar definierte Aufgaben programmiert sind. Service Roboter hingegen, beispielsweise in der Logistik oder im Haushalt, verfügen über eine höhere Autonomie, da sie sich auf wechselnde Umgebungen einstellen müssen. Am höchsten ist der Grad der Autonomie bei Personal Robotern, die eng mit dem Menschen interagieren und deshalb besonders flexibel und anpassungsfähig sein müssen.<sup>6</sup>

Typischerweise bestehen autonome Roboter aus drei zentralen Komponenten:

- Sensoren: Erfassen Informationen aus der Umgebung (z. B. Abstände oder Hindernisse)
- Aktuatoren: Führen Bewegungen aus (z. B. Motoren im Fahrwerk oder im Roboterarm)
- Steuerungseinheit: Verarbeitet die Sensordaten, plant Handlungen und steuert die Aktuatoren entsprechend an.

<sup>6</sup> Vgl. Dr. Mohamed Oubbati, Robotik, Abschnitt 1.2.2

## 2.2 Sensoren

Sensoren sind ein zentrales Element in autonomen Robotersystemen. Sie ermöglichen dem Roboter Informationen aus seiner Umgebung zu erhalten und Entscheidungen zu treffen. Ohne Sensorik wären die gezielte Bewegung, Hinderniserkennung und Objektinteraktion nicht möglich.

Ein Sensor wandelt dabei ein physikalisches Signal in ein elektronisches Ausgangssignal um<sup>7</sup> (siehe Abb. 2).



*Abb. 2: Grundprinzip eines Sensors*

Quelle: Oubbati, M. (2007). Robotik. Uni Ulm.

### 2.2.1 LiDAR-Sensoren

LiDAR steht für Light Detection and Ranging und ist eine aktive Fernerkundungstechnologie. Anders als bei passiven Methoden wie Kameraaufnahmen feuert der Sensor aktiv Laserlicht ab und misst die Laufzeit, den sogenannten Time-of-Flight-Effekt (siehe Formel unten), um daraus die Entfernung vom Sensor zum Objekt zu berechnen.<sup>8</sup>

Ein LiDAR sendet kurze Laserimpulse aus. Die Wellenlänge variiert je nach Anwendung. Meistens kommen Grünlicht (~532 nm, sichtbar) oder Nahinfrarot (~905-1550 nm, unsichtbar) zum Einsatz. Grünlicht wird für bathymetrisches LiDAR eingesetzt (also Messungen unter Wasser, Küsten, Flüssen), weil Grünlicht im Wasser deutlich tiefer eindringt als ein rotes oder infrarotes Licht. Nahinfrarot wird häufig bei Topografien, autonomen Fahrzeugen und Robotik genutzt. Seine Vorteile sind vielfältig. Nahinfrarot wird gut reflektiert, ist für das menschliche Auge sicher (vor allem im Bereich von 1550 nm) und hat in der Luft eine hohe Reichweite. Jeder ausgesandte Laserimpuls wird reflektiert und gelangt zum Sensor zurück. Die gemessene Time of Flight wird wie folgt berechnet:<sup>9</sup>

$$\Delta x = \frac{c \cdot t}{2}$$

<sup>7</sup> Vgl. Dr. Mohamed Oubbati, Robotik, Abschnitt 2.1

<sup>8</sup> Vgl. Leah A. Wasser, The Basics of LiDAR

<sup>9</sup> Vgl. Leah A. Wasser, The Basics of LiDAR

Es stehen  $c$  für die Lichtgeschwindigkeit,  $t$  für die gemessene Laufzeit des Impulses und  $\Delta x$  für die gemessene Distanz zwischen Sensor und Objekt. Der Faktor  $\frac{1}{2}$  berücksichtigt den Hin- und Rückweg des Lasers.

LiDAR-Systeme werden grundsätzlich in 2 Kategorien unterteilt:

- 2D-LiDAR-System: Dieses scannt eine einzelne Ebene. Typische Anwendung ist die Hinderniserkennung und die Distanzmessung in der Robotik oder in Sicherheitssystemen. Solche Sensoren sind kostengünstig und liefern schnelle Ergebnisse, erfassen jedoch keine vollständige räumliche Umgebung.
- 3D-LiDAR-System: Dieses erfasst ein ganzes Volumen, indem mehrere Ebenen oder rotierende Spiegel genutzt werden. Dabei entstehen sogenannte Punktwolken, die die Umgebung detailliert in drei Dimensionen abbilden. Diese Systeme sind Standard in autonomen Fahrzeugen, Drohnen und Mapping.

LiDAR-Technologie bietet eine Reihe von Vorteilen, die sie für Anwendungen wie Vermessung, Robotik und autonome Fahrzeuge besonders attraktiv machen.<sup>10</sup>

Ein entscheidender Vorteil von LiDAR ist die hohe Genauigkeit. Durch die kurzen Wellenlängen des Laserlichts können selbst kleine Objekte erfasst und in exakten 3D-Modellen dargestellt werden. Damit lassen sich Objekte wie Bäume, Personen oder Wände zuverlässig unterscheiden.<sup>11</sup>

Ein weiterer Vorteil liegt in der Geschwindigkeit der Datenerfassung. LiDAR-Sensoren senden und empfangen Laserimpulse im Nanosekundenbereich, wodurch grosse Flächen in kurzer Zeit mit hoher Punktedichte erfasst werden können.<sup>12</sup>

Zudem ermöglicht die Technologie die Erfassung von schwer zugänglichen Gebieten. Regionen wie dichte Wälder, Gebirgszüge oder unwegsames Gelände lassen sich problemlos kartieren, was mit traditionellen Methoden oft nur schwer oder gar nicht möglich ist.<sup>13</sup>

Trotz dieser Vorteile gibt es auch Einschränkungen. Der Betrieb von LiDAR erfordert hohes Fachwissen in der Vermessungstechnik. Kenntnisse über Kontrollmessungen, Basisstationen und Referenzpunkte sind notwendig, um zuverlässige Daten zu erhalten.<sup>14</sup>

---

<sup>10</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

<sup>11</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

<sup>12</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

<sup>13</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

<sup>14</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

Ein weiterer Nachteil ist der hohe Investitionsaufwand für hochwertige Sensoren. Besonders für Unternehmen, die eigene Systeme aufbauen wollen, können die Anschaffungskosten der Geräte sehr hoch ausfallen.<sup>15</sup>

## 2.2.2 Kamera (visuelle Sensorik)

Visuelle Sensorik nutzt Kameras als passive Sensoren, die einfach das vorhandene Licht aufnehmen. Es wird kein Laser wie beim LiDAR ausgesendet. Sie erfassen sehr viele 2D-Bilder von verschiedenen Winkeln, die später mit Software zu 3D-Informationen umgerechnet werden. Dieser Prozess ist auch bekannt als Fotogrammetrie.

Ein zentraler Vorteil der Fotogrammetrie liegt in den geringen Kosten. Im Vergleich zu LiDAR sind Kameras deutlich günstiger, wodurch sich Fotogrammetrie besonders für Projekte mit knappen Budgets eignet.<sup>16</sup>

Darüber hinaus liefert Fotogrammetrie umfangreiche visuelle Informationen. Neben der Geometrie der Umgebung werden auch Texturen und Farben erfasst, was eine realistische Darstellung ermöglicht. Dies ist insbesondere für Anwendungen relevant, bei denen visuelle Qualität von Bedeutung ist, wie etwa Architektur oder virtuelle Simulationen.<sup>17</sup>

Allerdings weist Fotogrammetrie auch Einschränkungen auf. Ein zentrales Problem ist die geringere Genauigkeit. Faktoren wie Sensorgröße, Blendenöffnung, Auflösung, Brennweite sowie Flughöhe beeinflussen die Präzision erheblich. Für hohe absolute Genauigkeit ist meist der Einsatz von GCPs erforderlich.<sup>18</sup> Ground Control Points (GCPs) sind Referenzpunkte am Boden, deren geografische Koordinaten exakt bekannt sind.<sup>19</sup>

Zudem ist Fotogrammetrie anfällig für Vegetationsabschattung. Dichte Baumkronen oder Büsche können Strukturen verdecken, da die Methode anders als LiDAR nicht durch die Vegetation hindurchmessen kann.<sup>20</sup>

Zusätzlich werden Kameras in der Robotik und Computer Vision nicht nur zur Rekonstruktion von 3D-Strukturen eingesetzt, sondern auch für die Objekterkennung. Mithilfe von Bildverarbeitungsbibliotheken wie OpenCV lassen sich Objekte anhand von Farbmerkmalen, Kanten oder Konturen identifizieren. So kann ein Roboter zum Beispiel Objekte erkennen und

---

<sup>15</sup> Vgl. Flyguys, Advantages and disadvantages of LiDAR technology

<sup>16</sup> Vgl. Jouav, LiDAR vs Photogrammetry, Abschnitt Pros and cons

<sup>17</sup> Vgl. Jouav, LiDAR vs Photogrammetry, Abschnitt Pros and cons

<sup>18</sup> Vgl. Jouav, LiDAR vs Photogrammetry, Abschnitt Pros and cons

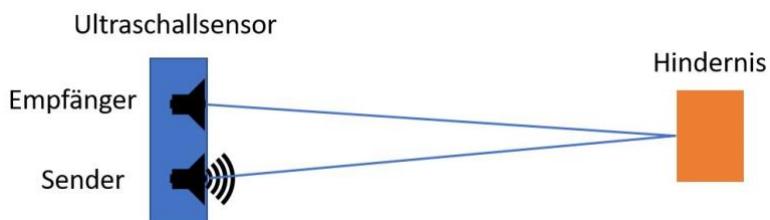
<sup>19</sup> Vgl. DJI-Enterprise, Ground Control Points, Abschnitt What are ground control points?

<sup>20</sup> Vgl. Jouav, LiDAR vs Photogrammetry, Abschnitt Pros and cons

lokalisieren. Moderne Verfahren gehen darüber hinaus und nutzen künstliche Intelligenz (KI). Vor allem Deep Learning mit Convolutional Neural Networks (CNNs) wird genutzt, um komplexe Objekte oder sogar Personen zuverlässig zu erkennen.<sup>21</sup>

### 2.2.3 Ultraschallsensoren

Ultraschallsensoren dienen der berührungslosen Abstandsmessung. Sie arbeiten nach dem Prinzip des Aussendens und Empfangens von hochfrequenten Schallwellen von 20 kHz bis 300-500 kHz, die vom Objekt reflektiert und vom Sensor wieder aufgenommen werden.<sup>22</sup> Frequenzen oberhalb von ca. 500 kHz werden in der Luft kaum mehr eingesetzt, da die Dämpfung stark zunimmt.<sup>23</sup> Auf diese Weise lassen sich Informationen wie Distanz, Geschwindigkeit, Position oder auch Materialeigenschaften erfassen.<sup>24</sup>



*Abb. 3: Funktionsprinzip eines Ultraschallsensors*

*Quelle: Inf-Schule (2024). US-Sensoren am Auto.*

Die Distanzberechnung erfolgt nach dem Time-of-Flight-Prinzip. Die Zeit zwischen Aussenden und Empfangen des Signals wird gemessen und mithilfe der Schallgeschwindigkeit wird die Distanz berechnet. Die Formel lautet:<sup>25</sup>

$$\Delta x = \frac{c \cdot t}{2}$$

Es stehen  $t$  für die gemessene Laufzeit,  $c$  für die Schallgeschwindigkeit (343 m/s in Luft bei 20 °C) und  $\Delta x$  für die gemessene Distanz zwischen Sensor und Objekt.<sup>26</sup> Der Faktor  $\frac{1}{2}$  berücksichtigt den Hin- und Rückweg des Schalls.

<sup>21</sup> Vgl. LeCun, Bengio & Hinton, Deep Learning, Abschnitt Abstract

<sup>22</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 2

<sup>23</sup> Vgl. Qiu, Lu & Qiu, Review of Ultrasonic Ranging Methods And Their Current Challenges, Abschnitt 2.3

<sup>24</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 2

<sup>25</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 2

<sup>26</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 2

Ein Vorteil von Ultraschallsensoren ist ihre Einsatzfähigkeit unter schwierigen Bedingungen wie Staub, Rauch oder starker Helligkeit, wo optische Sensoren oft versagen. Sie finden Anwendung in industrieller Automatisierung, robotischer Navigation, Fahrzeugtechnik, Füllstandmessung und sogar in der medizinischen Bildgebung.<sup>27</sup>

Trotz vieler Vorteile bestehen Einschränkungen: Die Reichweite ist typischerweise auf den Nahbereich begrenzt, und Umweltfaktoren wie Regen, Schnee oder Temperaturschwankungen können die Genauigkeit beeinträchtigen. Daher werden Ultraschallsensoren häufig in Kombination mit anderen Sensortypen eingesetzt.<sup>28</sup>

## 2.3 Aktuatoren

Aktuatoren sind Schnittstellen zwischen der Steuerungseinheit des Roboters und seiner physischen Umwelt. Während Sensoren physikalische Größen in elektrische Signale umwandeln, wandeln Aktuatoren elektrische Signale in mechanische Arbeit um (siehe Abb. 4).<sup>29</sup> Damit übernehmen sie die Funktion der «Muskeln» des Roboters und ermöglichen jegliche Handlungen.



*Abb. 4: Grundprinzip eines Aktuators*

Quelle: Oubbati, M. (2007). *Einführung in die Robotik*. Uni Ulm.

### 2.3.1 Motoren

Die am häufigsten eingesetzten Aktuatoren in der mobilen Robotik sind elektrische Motoren. Dazu gehören vor allem Gleichstrommotoren (DC-Motoren) und bürstenlose Gleichstrommotoren (BLDC-Motoren). DC-Motoren lassen sich einfach über Versorgungsspannung oder eine Pulswidmenmodulation (PWM) in ihrer Drehzahl steuern: kurze/kleine Pulsdauer → tiefere mittlere Spannung → langsamere Drehzahl. Lange/große Pulsdauer → höhere mittlere Spannung → schnellere Drehzahl. Sie sind robust, günstig und deshalb ideal für den Antrieb von Rädern. BLDC-Motoren sind effizienter und langlebiger, benötigen jedoch eine spezielle Elektronik zur

<sup>27</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 2

<sup>28</sup> Vgl. Wei, Applications of Ultrasonic Sensors: A Review, Abschnitt 1 & 3.1.4

<sup>29</sup> Vgl. Dr. Mohamed Oubbati, Einführung in die Robotik, Abschnitt Einführung

Kommutierung (das ist die zeitliche richtige Umschaltung des Stroms in den Motorwicklungen, damit das Magnetfeld den Rotor kontinuierlich mitnimmt. Bei BLDC übernimmt das ein Regler, meist mit Hall-Sensoren oder sensorlos über die Gegen-EMK).<sup>30</sup>

Schrittmotoren bilden eine Sonderform: Sie bewegen sich in festen Schritten und können auch ohne Feedback eine Position anfahren. Damit eignen sie sich für präzise, aber eher langsame Bewegungen, solange keine hohen Lasten auftreten.<sup>31</sup>

### 2.3.2 Servomotoren

Servos sind Motoren, die zusammen mit einem Sensor (z. B. Encoder oder Potentiometer) und einer Steuerung eine geschlossene Regelung bilden. Dadurch können sie gezielt eine bestimmte Position oder Geschwindigkeit halten.<sup>32</sup> Sobald Spannung anliegt, wirken sie «steif» und halten ihre Stellung, wenn die Spannung weg fällt, verlieren sie ihre Kraft.

In der Praxis gibt es zwei Varianten:

- Industrielle Servoantriebe, die über Feldbus und präzise Encoder geregelt werden.
- Kompakte RC-Servos, wie sie in Robotergelenken verwendet werden. Letztere werden über ein PWM-Signal mit etwa 50 Hz angesteuert, wobei die Pulsbreite (z. B. 1-2 ms) direkt den gewünschten Winkel vorgibt. Damit lassen sich Roboterarme einfach und zuverlässig bewegen.<sup>33</sup>

## 2.4 Programmiersprachen in eingebetteten Systemen

Die Wahl der Programmiersprache ist ein entscheidender Faktor bei der Entwicklung eingebetteter Systeme und robotischer Anwendungen. Sie beeinflusst nicht nur die Ausführungszeit und Ressourcenverbrauch, sondern auch die Verständlichkeit, Wartbarkeit und Geschwindigkeit der Entwicklung. Einige Sprachen wie C oder C++ erlauben direkten Hardwarezugriff und eine sehr effiziente Ausführung, während Sprachen wie Python eine deutlich einfachere und schnellere Umsetzung ermöglichen, allerdings meist auf Kosten der Performance.

---

<sup>30</sup> Vgl. Siciliano & Khatib, Springer Handbook of Robotics, Abschnitt Mechanisms and Actuation

<sup>31</sup> Vgl. Anaheim Automation, Stepper Motors versus Servo Motors

<sup>32</sup> Vgl. Omron, Technical Explanation for Servomotors and Servo Drives

<sup>33</sup> Vgl. Adafruit, Using Servos with CircuitPython – Low Level Servo Control

### 2.4.1 C++

C++ ist eine kompilierte Sprache, deren Quellcode durch einen Compiler in Maschinencode übersetzt wird. Dadurch lassen sich Programme sehr effizient ausführen, was insbesondere in ressourcenbeschränkten Umgebungen etwa in eingebetteten Systemen von Vorteil ist. Ein wesentliches Merkmal von C++ ist die Möglichkeit der direkten Hardwarekontrolle sowie der manuellen Speicherverwaltung, wodurch sich sehr leistungsfähige und deterministische Anwendungen realisieren lassen. Diese Eigenschaften sind gerade in der Robotik von zentraler Bedeutung, wenn es beispielsweise um Motorsteuerung, Hindernisvermeidung oder signalverarbeitende Algorithmen geht, die in Echtzeit reagieren müssen.<sup>34</sup>

Eine in der Literatur dokumentierte Vergleichsstudie verdeutlicht diesen Vorteil auch in praktischen Messungen. Dabei wurden verschiedene Algorithmen, darunter Fast Fourier Transform (FFT), Cyclic Redundancy Check (CRC-32), Secure Hash Algorithm (SHA-256), Infinite Impulse Response (IIR) und Finite Impulse Response (FIR), in mehreren Programmiersprachen implementiert und hinsichtlich ihrer Ausführungszeit getestet. Die Ergebnisse zeigen, dass C und C++ in den meisten Fällen die besten Laufzeiten erzielen und somit als Referenz dienen. Im Durchschnitt ist C++ also die effizienteste Lösung.<sup>35</sup>

Die Untersuchung macht zudem deutlich, dass die Unterschiede zu modernen Alternativen in vielen Fällen nur wenige Mikrosekunden betragen. Dennoch gilt C++ aufgrund seiner langjährigen Reife, seiner breiten Hardwareunterstützung und seiner hohen Effizienz weiterhin als Goldstandard in eingebetteten Anwendungen, bei denen Leistung und Echtzeitfähigkeit oberste Priorität haben.<sup>36</sup>

### 2.4.2 Python

Python unterscheidet sich grundlegend von C++, da es sich um eine interpretierte Sprache handelt. Der Quellcode wird nicht vorab in Maschinencode übersetzt, sondern erst während der Laufzeit durch einen Interpreter verarbeitet. Dies macht die Sprache für Entwicklerinnen und Entwickler besonders attraktiv: Sie ist leicht erlernbar, verfügt über eine klare und kompakte

---

<sup>34</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 2.1 & 2.2

<sup>35</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 3 & 4

<sup>36</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 4 & 5

Syntax und ermöglicht eine schnelle Umsetzung von Ideen. Gerade für Prototyping, Ausbildung oder Projekte mit geringen Leistungsanforderungen wird Python deshalb häufig eingesetzt.<sup>37</sup>

Auf eingebetteten Systemen kommt Python meist in Form von MicroPython zum Einsatz. Diese Version enthält ein für ressourcenarme Umgebungen angepasstes Subset der Python-Funktionalität. Sie muss aber zuerst als Interpreter auf das Gerät übertragen werden. MicroPython unterstützt viele typische Peripherien, darunter auch Wi-Fi, arbeitet jedoch mit automatischer Speicherverwaltung und Garbage Collection. Letztere führt zu zusätzlichen Laufzeitunterbrechungen, die in Echtzeitanwendungen problematisch sein können.<sup>38</sup>

Die empirischen Ergebnisse der erwähnten Studie belegen die Nachteile von Python in Bezug auf Effizienz. Im Test der vorher erwähnten Algorithmen war MicroPython um ein Vielfaches langsamer als C/C++. Die Autorinnen und Autoren führen dies auf den erheblichen Overhead des Interpreters zurück, der im Vergleich zu kompilierten Sprachen massive Leistungseinbussen verursacht. Teilweise konnte der Interpreter nicht einmal den vollständigen Benchmark laden, sodass Workarounds notwendig waren.<sup>39</sup>

Trotz dieser Einschränkung bleibt Python eine wichtige Sprache in der Entwicklung. Sie erlaubt eine sehr schnelle und unkomplizierte Programmierung, was sie insbesondere für den Einstieg, für Lehrzwecke und für weniger strenge Anwendungsbereiche attraktiv macht. Die Studie hebt hervor, dass MicroPython vor allem für Studierende und Lernende geeignet ist. Mit ihr sind Entwicklungen erheblich einfacher und zugänglicher zu gestalten als in C++. Für hochleistungs- oder echtzeitsensitive Aufgaben ist Python hingegen weniger geeignet.<sup>40</sup>

---

<sup>37</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 2.2

<sup>38</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 2.1 & 2.2

<sup>39</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 2.4 & 3

<sup>40</sup> Vgl. Plauska, Liutkevičius & Janavičiūtė, Performance Evaluation of C/C++, MicroPython, Rust and TinyGO Programming Languages on ESP32 Microcontroller, Abschnitt 4 & 5

## 2.5 Bibliotheken

Bibliotheken bilden in der Informatik die Grundlage, um komplexe Aufgaben effizient umzusetzen. Sie enthalten wiederverwendbare Funktionen, Klassen und Module, die Programmierer\*innen in eigenen Projekten einbinden können, anstatt alles von Grund auf neu entwickeln zu müssen. Besonders in der Robotik sind Bibliotheken entscheidend. So können komplexe Funktionen wie Bildverarbeitung, Pfadplanung oder Kommunikationsmechanismen modular genutzt und kombiniert werden.

### 2.5.1 OpenCV (cv2)

OpenCV (cv2) ist eine offene, modulare Computer-Vision-Bibliothek für Echtzeit-Bild- und Videoverarbeitung. Die Kern-API ist in C++ mit weit verbreiteten Bindings für Python und Java. Die Bibliothek ist plattformübergreifend einsetzbar (u. a. Windows, Linux, macOS).<sup>41</sup>

OpenCV (cv2) ist in funktionale Module gegliedert, z. B. core (Grundfunktion), imgproc (Bildverarbeitung, imgcodecs (Formate), videoio (Video-I/O), highgui (GUI), video (Videoanalyse) und calib3d (Kamerakalibrierung/3D-Rekonstruktion. Diese Modularisierung erleichtert in der Robotik die gezielte Nutzung von Bildverarbeitung, Kamera-I/O und 3D-Geometrie.<sup>42</sup>

### 2.5.2 ultralytics (YOLO, Python)

Die ultralytics-Bibliothek ist ein Python-Paket zur schnellen Nutzung moderner YOLO-Modelle für Objekterkennung, Segmentierung, Klassifikation, Pose-Schätzung und Oriented Bounding Boxes (OBB). Es stellt eine leichtgewichtige API und ein Command Line Interface (CLI) bereit und lässt sich plattformübergreifend in bestehende Robotik-Pipelines integrieren.<sup>43</sup>

Die Bibliothek arbeitet aufrufgesteuert: Entwickler importieren die YOLO-Klasse und rufen Methoden wie `model.train()` oder `model.predict()` auf. Funktional ist sie in Modi gegliedert (u. a. Train, Predict, Val, Export, Track, Benchmark), was eine klare Trennung zwischen Training, Inferenz, Validierung, Deployment und Tracking erlaubt.<sup>44</sup>

Training erfolgt auf eigenen Datensätzen über den Train-Modus. Zentrale Einstellungen (z. B. Daten-/Modell-Konfiguration) werden über YAML/CFG gesteuert. So kann die Bibliothek schnell

---

<sup>41</sup> Vgl. OpenCV Documentation, OpenCV modules, Abschnitt Introduction

<sup>42</sup> Vgl. OpenCV Documentation, OpenCV modules, Abschnitt main modules

<sup>43</sup> Vgl. Ultralytics Docs, Ultralytics YOLO Docs, Abschnitt Home; Quickstart; Python Usage & Tasks

<sup>44</sup> Vgl. Ultralytics Docs, Ultralytics YOLO Docs, Abschnitt Python Usage & Modes

an domänenspezifische Aufgaben angepasst werden, ohne dass die Projektarchitektur verändert werden muss.<sup>45</sup>

Für Inferenz bietet der Predict-Modus eine einheitliche API und CLI mit Unterstützung vielfältiger Eingaben (Bildordner, Videos, Streams, Webcam). Das ermöglicht Echtzeit-Anwendungen (z. B. Laufzeit-Checks am Roboter) mit minimalem Boilerplate-Code.<sup>46</sup>

Zum Deployment stellt der Export-Modus gängige Zielformate bereit (u. a. ONNX, TensorRT, OpenVINO, CoreML, TFLite) und erlaubt per Benchmark-Modus den Vergleich von Genauigkeit/Latenz über Formate und Hardware hinweg. Das erleichtert die Auswahl einer performanten Runtime auf Embedded-Systemen.<sup>47</sup>

### 2.5.3 threading (Python)

Die threading-Bibliothek stellt eine höherstufige Schnittstelle bereit, um mehrere Threads innerhalb eines Prozesses gleichzeitig auszuführen. Threads teilen sich den Adressraum, weshalb der Datenaustausch effizient ist. Besonders I/O-gebundene Aufgaben (Date-/Netzwerk-I/O, Kamera-/Sensor-Streams) profitieren, weil Wartezeiten überlappt werden können.<sup>48</sup>

In CPython (der referenziellen Standard-Implementation von Python, in C geschrieben) sorgt der Global Interpreter Lock (GIL) dafür, dass zu einem Zeitpunkt immer nur ein Thread Python-Bytecode ausführt. Darum ist Threading in Python vor allem für I/O-gebundene Workloads sinnvoll, weil Wartezeiten überlappen können. Für CPU-gebundene Aufgaben ist meist multiprocessing die bessere Wahl, da separate Prozesse den GIL nicht teilen und so mehrere Kerne gleichzeitig nutzen.<sup>49</sup>

Threads werden via `threading.Thread(...), .start() und .join()` (Warten auf Abschluss) gesteuert. Das Flag `daemon` definiert, ob ein Thread das Beenden des Programms verhindert: Nur Nicht-Daemon-Threads halten den Prozess offen. Verbleiben nur Daemon-Threads, beendet sich das Python-Programm automatisch.<sup>50</sup>

---

<sup>45</sup> Vgl. Ultralytics Docs, Ultralytics YOLO Docs, Abschnitt Modes

<sup>46</sup> Vgl. Ultralytics Docs, Ultralytics YOLO Docs, Abschnitt Modes

<sup>47</sup> Vgl. Ultralytics Docs, Ultralytics YOLO Docs, Abschnitt Modes/Export-Tutorial

<sup>48</sup> Vgl. Python Docs, Concurrent Execution, Abschnitt threading – Thread-based parallelism

<sup>49</sup> Vgl. Python Docs, Concurrent Execution, Abschnitt threading – Thread-based parallelism

<sup>50</sup> Vgl. Python Docs, Concurrent Execution, Abschnitt threading – Thread-based parallelism

Daniel Würmli

Loadlifter

20.10.2025

Für Messungen verfügt time über mehrere Uhren mit unterschiedlicher Semantik: `time.time()` liefert die (verstellbare) Systemuhr, `time.monotonic()` garantiert eine nie zurückspringende Uhr für Deadlines/Timeouts, `time.perf_counter()` ist ein hochauflösender Performance-Zähler für kurze Laufzeiten/Benchmarks, `time.process_time()` misst CPU-Zeit des Prozesses (ohne Schlaf-/Wartezeit).<sup>57</sup>

`time.sleep(secs)` blockiert den aufrufenden Thread für mindestens die angegebene Dauer. Die tatsächliche Schlafzeit hängt vom OS-Scheduler ab und kann leicht länger sein. In zyklischen Loops kombiniert man daher meist eine monotone Messuhr (`monotonic()`) bzw. für höchste Präzision `perf_counter()` mit `sleep()`, um CPU-schonend und driftarm zu takten.<sup>58</sup>

## 2.5.6 argparse (Python)

Die argparse-Bibliothek ist das Standardmodul zur Kommandozeilenverarbeitung in Python. Es liest Argumente aus `sys.argv`, erzeugt benutzerfreundliche Hilfe-/Usage-Texte und meldet Fehler bei ungültigen Eingaben. Zentrales Objekt ist `ArgumentParser`, über das man eine Kommandozeilenschnittstelle (CLI) definiert.<sup>59</sup>

Argumente werden mit `add_argument()` beschrieben (positionelle Argumente, Optionen mit Werten, Flags wie `--verbose` via `action='store_true'`). Mit `type`, `default`, `choices` etc. steuert man Datentypen, Standardwerte und zulässige Wertebereiche. `parse_args()` liefert einen Namespace mit den geprästen Werten.<sup>60</sup>

Insgesamt ermöglicht argparse, reproduzierbaren und flexibel konfigurierbaren Code zu schreiben, da ein Programm mit denselben Argumenten stets dasselbe Verhalten zeigt.

## 2.5.7 math (Python)

Die math-Bibliothek stellt grundlegende mathematische Funktionen und Konstanten zur Verfügung, die für Berechnungen in Steuerungs-, Navigations- und Sensordatenverarbeitungsaufgaben benötigt werden. Sie bietet u. a. die Konstanten `math.pi`

---

<sup>57</sup> Vgl. Python Docs, Generic Operating System Services, Abschnitt time – Time access and conversions

<sup>58</sup> Vgl. Python Docs, Generic Operating System Services, Abschnitt time – Time access and conversions

<sup>59</sup> Vgl. Python Docs, Command-line interface libraries, Abschnitt argparse – Parser for command-line options, arguments and subcommands

<sup>60</sup> Vgl. Python Docs, Command-line interface libraries, Abschnitt argparse – Parser for command-line options, arguments and subcommands

und `math.e` sowie Funktionen für Potenzen, Wurzeln, Logarithmen und Exponentialberechnungen.<sup>61</sup>

Ein zentraler Bereich sind die trigonometrischen Funktionen (`sin`, `cos`, `tan` sowie deren Inversen `asin`, `acos`, `atan`, `atan2`). Sie ermöglichen die Umrechnung zwischen Längen und Winkeln und sind in der Robotik beispielsweise für Orientierung, Richtungsbestimmung und Distanzberechnung mit Sensordaten unverzichtbar. Ebenso bietet `math.radians()` und `math.degrees()` die Umrechnung zwischen Radiant und Grad.<sup>62</sup>

Für die Analyse und Nachbearbeitung von Sensordaten bietet die Bibliothek zusätzliche Funktionen wie `hypot(x, y)` zur Berechnung von euklidischen Distanzen, `sqrt()` für Wurzeln sowie Rundungsfunktionen wie `floor()`, `ceil()` oder `trunc()`. Auch numerische Hilfsmittel wie `fabs()` (Betrag) oder `isclose()` (Vergleich mit Toleranz) sind verfügbar und helfen bei robusten Berechnungen in gleitender Genauigkeit.<sup>63</sup>

## 2.5.8 pathlib (Python)

Die `pathlib`-Bibliothek stellt eine objektorientierte Schnittstelle für Datei- und Pfadoperationen bereit. Anstatt mit Zeichenketten und Funktionen wie `os.path.join()` oder `os.path.exists()` zu arbeiten, erlaubt `pathlib` die Nutzung der Klasse `Path`, die Pfade als Objekte repräsentiert, und Methoden direkt daran anbietet.<sup>64</sup>

Ein `Path`-Objekt repräsentiert einen Pfad im Dateisystem (absolut oder relativ) und kann plattformunabhängig mit Operatoren kombiniert werden: `Path("data") / "images"` ergibt automatisch den richtigen Pfad, egal ob unter Windows, Linux oder MacOS. Dadurch wird der Code lesbarer und portabler.<sup>65</sup>

Häufig genutzte Funktionen sind z. B. `exists()` (Existenzprüfung), `is_file()/is_dir()` (Typ-Check), `mkdir()` (Verzeichnisse anlegen), `unlink()` (Datei löschen) und `rename()` (umbenennen/verschieben). Über `open()`, `read_text()`, `write_text()` oder `write_bytes()` lassen sich Dateien direkt ansprechen, ohne separate `open()`-Aufrufe.<sup>66</sup>

---

<sup>61</sup> Vgl. Python Docs, Numeric and Mathematical Modules, Abschnitt `math – Mathematical functions`

<sup>62</sup> Vgl. Python Docs, Numeric and Mathematical Modules, Abschnitt `math – Mathematical functions`

<sup>63</sup> Vgl. Python Documentation, Numeric and Mathematical Modules, Abschnitt `math – Mathematical functions`

<sup>64</sup> Vgl. Python Docs, File and Directory Access, Abschnitt `pathlib – Object-oriented filesystem paths`

<sup>65</sup> Vgl. Python Docs, File and Directory Access, Abschnitt `pathlib – Object-oriented filesystem paths`

<sup>66</sup> Vgl. Python Docs, File and Directory Access, Abschnitt `pathlib – Object-oriented filesystem paths`

### 2.5.9 termios (Python)

Die Bibliothek `termios` liefert Zugriff auf POSIX-Terminalsteuerungen unter Unix-Systemen (Linux, macOS). Damit lassen sich u. a. Eingabemodi (zeilenweise *canonical* vs. zeichenweise *raw/CBREAK*), Echo-Verhalten sowie Zeit-/Zeichenlimits für Reads (`VMIN/VTIME`) konfigurieren.

Unter Windows ist `termios` nicht verfügbar.<sup>67</sup>

Zentrale Aufrufe sind `termios.tcgetattr(fd)` (aktueller Attribut-Set lesen) und `termios.tcsetattr(fd, when, attrs)` (Attribute setzen). In der Praxis arbeitet man häufig auf `sys.stdin` (bzw. dessen `fileno()`) und nutzt für bequeme Umschaltungen das Hilfsmodul `tty` mit `tty.setraw(fd)` bzw. `tty.setcbreak(fd)`.<sup>68</sup>

### 2.5.10 flask (Python)

Flask ist ein leichtgewichtiges Webframework (Microframework) für Python zur Erstellung von HTTP-Diensten, Webanwendungen und REST-APIs. Es stellt den Kern für Routing, Request/Response-Verarbeitung und Konfiguration bereit und lässt sich bei Bedarf über Erweiterungen (z. B. Auth, DB, CORS) modular ausbauen.<sup>69</sup>

Das Routing erfolgt über Dekoratoren wie `@app.route("/path")`, welche die URL einer View-Funktion zuordnen und so die HTTP-Endpunkte definieren. Über das `request`-Objekt lassen sich Parameter, Form-Daten und Dateien lesen. `Response`-Objekte geben HTML, JSON oder Dateien zurück. Damit bildet flask den minimalen, aber vollwertigen Rahmen für REST-artige Schnittstellen.<sup>70</sup>

Für Darstellung und Assets bietet flask integrierte Template-Unterstützung (Jinja2) sowie Konventionen für `templates/` und `static/`. Damit können HTML-Seiten serverseitig gerendert und statische Inhalte (CSS/JS/Bilder) effizient ausgeliefert werden. Für APIs steht die bequeme JSON-Rückgabe (z. B. `return jsonify(data)`) zur Verfügung.<sup>71</sup>

Im Entwicklungsbetrieb verfügt flask über einen Debug-Server (Reload, Debugger) und eine klare Konfigurationsmechanik (Umgebungsvariablen, `config`), während im Produktivbetrieb ein WSGI-Server (z. B. `gunicorn`, `uwsgi`) empfohlen wird. Für grössere Projekte unterstützen Application Factory und Blueprints eine saubere Modularisierung.<sup>72</sup>

---

<sup>67</sup> Vgl. Python Docs, Unix-specific services, Abschnitt `termios – POSIX style tty control`

<sup>68</sup> Vgl. Python Docs, Unix-specific services, Abschnitt `termios – POSIX style tty control`

<sup>69</sup> Vgl. Flask Documentation, Welcome/Overview (gesamtes Kapitel)

<sup>70</sup> Vgl. Flask Documentation, Quickstart, Abschnitt `Routing/Accessing Request Data/About Responses`

<sup>71</sup> Vgl. Flask Documentation, Quickstart/Tutorial, Abschnitt `Static Files`

<sup>72</sup> Vgl. Flask Documentation, Deploying to Production (gesamtes Kapitel)

### 2.5.11 smbus2 (Python)

Die Bibliothek smbus2 ist eine Python-Implementierung des SMBus-Protokolls auf Basis des I<sup>2</sup>C-Bus und als Drop-in-Replacement für das ältere smbus gedacht (gleiche Syntax, zusätzliche Features). Ziel ist eine vollständige, erweiterbare Userspace-Anbindung an I<sup>2</sup>C/SMBus ohne eigenen Kernel-Treiber.<sup>73</sup>

Kern der API ist die Klasse SMBus, über die typische SMBus-Operationen bereitstehen: `read_byte`, `write_byte`, `read_word_data`, `write_word_data`, `read_i2c_block_data`, `write_i2c_block_data` sowie erweiterte Aufrufe wie `block_process_call`. Für komplexere Transaktionen (z. B. Repeated-Start) stellt smbus2 die Helfer `i2c_msg` und `i2c_rdwr` zur Verfügung, mit denen sich mehrere Lese-/Schreib-Operationen in einer kombinierten I<sup>2</sup>C-Transaktion ausführen lassen.<sup>74</sup>

Die Nutzung erfolgt über das Linux-I<sup>2</sup>C-Userspace-Interface unter `/dev/i2c-*` (z. B. `/dev/i2c-1`). Dafür muss das Kernel-Modul `i2c-dev` verfügbar sein; Geräteadressen werden über `ioctl I2C_SLAVE` gesetzt (standardmäßig 7-Bit; optional 10-Bit). Die Kernel-Doku beschreibt zusätzlich SMBus-Spezifika wie PEC (Packet Error Checking) und die verfügbaren SMBus-Aufrufe.<sup>75</sup>

## 2.6 Frameworks und Middleware

Frameworks und Middleware stellen in der Robotik eine übergeordnete Schicht gegenüber klassischen Bibliotheken dar. Während Bibliotheken wiederverwendbare Funktionen bereitstellen, bieten Frameworks eine strukturierte Architektur, in die verschiedene Software-Komponenten wie Treiber, Algorithmen und Steuerlogik eingebettet werden können. Middleware ergänzt dies, indem sie die Kommunikation zwischen verteilten Komponenten eines Robotersystems organisiert und Mechanismen wie Nachrichtenübertragung, Namensdienste oder Synchronisation bereitstellt. Dadurch lassen sich komplexe, modulare und skalierbare

---

<sup>73</sup> Vgl. PyPI, smbus2 – A drop-in replacement for smbus-cffi/smbus-python in pure Python, Abschnitt Introduction

<sup>74</sup> Vgl. smbus2 Documentation, smbus2 – A drop-in replacement for smbus-cffi/smbus-python (gesamtes Kapitel)

<sup>75</sup> Vgl. Linux Kernel Documentation, I2C: dev-interface (gesamtes Kapitel)

Systeme entwickeln, in denen mehrere Sensoren, Aktuatoren und Algorithmen parallel zusammenarbeiten.<sup>76</sup>

### 2.6.1 ROS

Das Robot Operating System (ROS) ist ein umfassendes Framework, das sowohl Bibliotheken als auch Middleware-Komponenten umfasst. Es ermöglicht eine modulare Architektur, in der einzelne Funktionseinheiten wie Sensoren, Aktuatoren oder Algorithmen in sogenannten Nodes gekapselt und über Topics, Services oder Actions miteinander verbunden werden. Dadurch können komplexe Roboterarchitekturen flexibel aufgebaut und erweitert werden. Ein weiterer Vorteil ist die grosse Community, die eine Vielzahl von Treibern, Tools und Algorithmen Paketen bereitstellt.<sup>77</sup>

### 2.6.2 Orocос

Orocос (Open Robot Control Software) ist ein spezialisiertes Framework, das sich vor allem auf die Echtzeitsteuerung von Robotern konzentriert. Im Gegensatz zu generischeren Frameworks wie ROS bietet Orocос eine Infrastruktur, die auf harte zeitliche Anforderungen ausgerichtet ist. Damit eignet es sich besonders für industrielle Anwendungen wie Roboterarme oder Manipulatoren, bei denen präzise und deterministische Steuerung unabdingbar ist.<sup>78</sup>

Ein zentrales Merkmal von Orocос ist die Integration mit Echtzeitbetriebssystemen wie RTAI oder Xenomai, wodurch zeitkritische Abläufe zuverlässig umgesetzt werden können. Neben der Kommunikationsinfrastruktur stellt Orocос auch eine Vielzahl von Algorithmen Bibliotheken bereit, etwa für Kinematik, Bewegungssteuerung oder probabilistische Verfahren wie Bayesian Filtering.<sup>79</sup>

Darüber hinaus unterstützt Orocос auch Middleware wie ROS, sodass eine Kombination von Echtzeitsteuerung und flexibler Middleware realisierbar ist.<sup>80</sup>

---

<sup>76</sup> Vgl. Iñigo-Blasco et al, Diaz-del-Rio, Romero-Ternero, Cagigas-Muñiz & Vicente-Diaz, Robotics software

## 3 Analyse

### 3.1 Namenswahl

Der Titel Loadlifter wird gewählt, weil er die Kernfunktion des Systems prägnant beschreibt: Lasten aufnehmen, transportieren und ablegen. Zusätzlich ist er eine bewusste Referenz auf die LoadLifter-Roboter aus Star Wars, die mich in ihren Aufgaben stark an meinen Roboter erinnern und den Namen zugleich einprägsam und ansprechend machen. Der Begriff ist kurz, international verständlich und gut aussprechbar, was ihn auch für Dateinamen, Repositories und UI-Bezeichnungen praktikabel macht. Er bleibt technologienutral, d. h. er bindet nicht an eine bestimmte Hardware oder ein Framework und bleibt damit auch bei Weiterentwicklungen gültig. Zur Vermeidung offensichtlicher Kollisionen wird eine einfache Verfügbarkeitsprüfung (Websuche/GitHub) durchgeführt. Für die Nutzung im Rahmen der Maturaarbeit sind keine Konflikte zu erwarten.

### 3.2 Projektauftrag

Das Ziel meines autonomen Fahrzeugmodells ist es, mit einer KI-basierten Objekterkennung ein gezeigtes Objekt (von vier verschiedenen Objekten) in einer vorgegebenen Zone zu finden und es zu einem festgelegten Zielort zurückzubringen.

#### 3.2.1 Must-Haves

1. Grundlegende Navigation:
  - Programmierung von Bewegungsabläufen (vorwärts, rückwärts, drehen).
  - Fähigkeit, sich autonom auf vordefiniertem Pfad zu bewegen.
2. Objektbeschaffung:
  - Der Roboter kann ein Objekt an einem vordefinierten Platz abholen und zur Basis zurückbringen.
3. Grundlegende Sensorik:
  - Nutzung von Sensoren zur Erkennung von Wänden.
  - Verarbeitung der Sensordaten zur Entscheidungsfindung.
4. Hardware-Integration:
  - Verbindung des Raspberry Pi mit Aktuatoren & Sensoren.

- Sicherstellung einer stabilen Stromversorgung und Signalübertragung zwischen den Modulen
5. Programmierung in einer Hochsprache:
- Verwendung von einer Hochsprache zur Entwicklung effizienter und stabiler Software.
  - Strukturierter und kommentierter Code für bessere Wartbarkeit.
6. Mechanischer Arm:
- Entwicklung und Integration eines funktionierenden Greifarms.
  - Programmierung der Greiffunktion für das Aufheben und Ablegen von Objekten.
7. Dokumentation:
- Ausführliche Beschreibung des Projektverlaufs und der technischen Umsetzung.
  - Erklärung der Herausforderungen und deren Lösungen.

### 3.2.2 Nice-to-Haves

1. Kameraintegration:
  - Nutzung einer Kamera zur Bildverarbeitung.
  - Objekterkennung mittels Computer Vision (z. B. OpenCV).
2. Objektsuche, Erkennung und -beschaffung:
  - Suche und Erkennung (4 Farben) eines Objekts mittels Computer Vision.
  - Ergreifen und zurückbringen des Objektes zur Basis.
3. Benutzeroberfläche:
  - Entwicklung einer GUI zur Überwachung und Steuerung des Roboters.
  - Anzeige von Sensordaten und Systemstatus in Echtzeit.
4. Netzwerkkommunikation:
  - Fernsteuerung oder -überwachung über WLAN (SSH-Protokoll).
  - Möglichkeit zur Aktualisierung der Software über das Netzwerk.
5. Energieverwaltung:
  - Implementierung von Energiesparmodi.
  - Anzeige des Batteriestatus und Warnung bei niedrigem Ladezustand.
6. Modulares Design:
  - Aufbau des Roboters in modularer Bauweise für einfache Erweiterungen.
  - Dokumentation für zukünftige Anpassungen oder Verbesserungen.
7. Sicherheitsfunktionen:
  - Not-Aus-Schalter und Sicherheitsprotokolle.

- Einhaltung von Sicherheitsstandards für Robotik-Anwendungen.

Die Must- und Nice-to-Haves wurden im Verlauf der Arbeit mit dem Einverständnis des Betreuers angepasst und entsprechen daher dem Vertrag nur noch teilweise.

### 3.2.3 Abgrenzung

- Keine Hinderniserkennung und -umfahrung.
- Kein dedizierter Not-Stopp/Sicherheitskreis.
- Kein 3D-LiDAR und kein SLAM.
- Kein ROS-2-Stack.
- Keine harten Echtzeitanforderungen (C++)

## 3.3 Recherche

Die Vorbereitung der praktischen Umsetzung des Projekts Loadlifter basiert auf einer umfassenden und zielgerichteten Recherche. Durch das intensive Studium von Herstellerdokumentationen, Beispielcode und das Durchführen kleiner Tests lassen sich Unsicherheiten frühzeitig eliminieren. Die Recherche ist in zwei Teile unterteilt. Die Hardware-Recherche sowie die Software-Stack-Recherche.

### 3.3.1 Hardware-Recherche

Ein Schwerpunkt der Recherche liegt auf der verwendeten Hardware. Zunächst wurde der Mecanum-Fahrantrieb mit vier Motoren analysiert. Hierbei hilft der Quellcode des Herstellers Hiwonder, der den Armpi Pro-Roboterarm und das Chassis bereitstellt. Aus dem Code und der Dokumentation geht hervor, dass der mitgelieferte 4-Kanal-Motorcontroller über I<sup>2</sup>C angesprochen wird. Insbesondere ist die I<sup>2</sup>C-Adresse auf 0x34 voreingestellt und die Geschwindigkeit der vier Motoren wird über die Register 51 bis 54 kontrolliert. Diese Informationen sind entscheidend, um die Motorsteuerung mit Python (über die Bibliothek smbus2) korrekt zu implementieren. Schon in kleinen Tests auf dem Raspberry Pi konnte ich so die Motoren gezielt ansteuern. Ohne dieses Vorwissen aus der Dokumentation verliere ich sonst Zeit mit der Suche nach der richtigen I<sup>2</sup>C-Adresse oder Registerbelegung. Ebenso zeigt der Hersteller-Code, wie man den Fahrmixer für die Mecanum-Räder umsetzt, um Vorwärts-/Rückwärtsfahren, Seitwärtsbewegungen und Drehungen zu kombinieren. Diese Logik übernehme ich in mein eigenes Steuermodul, was Fehlversuche vermeidet.

Auch die Servoansteuerung des Greifarms analysiere ich vorab theoretisch. Der Arm verfügt über mehrere Servomotoren, die für die Gelenke und den Greifer zuständig sind. Hiwonder setzt hier auf sogenannte Bus-Servos, die über ein serielles Protokoll angesteuert werden. Im Hersteller-Code findet sich beispielsweise ein Aufruf `bus_servo_control.set_servos(...)`, mit dem sich mehrere Servos synchron auf Zielpositionen bewegen lassen. Anhand dieses Beispiels habe ich verstanden, wie die Parameter (Servonummern und Winkelwerte) übergeben werden und wie Timing-Parameter (z. B. Bewegungsdauer) wirken. Die Recherche macht deutlich, dass mit dem Bus-Servo-System komplexe Bewegungen des Arms koordiniert werden können, ohne jeden Servo einzeln und sequenziell steuern zu müssen. Dieses Verständnis ermöglicht es mir, meinen eigenen Code so zu strukturieren, dass alle Arm-Servos reibungslos zusammenspielen. Mögliche Stolpersteine, wie etwa die falsche Adressierung eines Servos oder unpassende Geschwindigkeiten, vermeide ich von vornherein.

Ein weiterer kritischer Hardware-Aspekt ist der LiDAR-Sensor (Orbbec Oradar MS200). Da dieser Sensor ein zentrales Element zur Umgebungswahrnehmung darstellt, untersuche ich die vom Hersteller bereitgestellten Unterlagen und Beispielprogramme gründlich. Die Orbbec-Dokumentation zeigt zunächst die technischen Schnittstellen: der LiDAR kommuniziert über eine serielle UART-Verbindung mit 230400 Baud (8 Datenbits, keine Parität). Mit diesem Wissen kann ich den Sensor für die Anwendung konfigurieren.

Schnittstelle passend konfigurieren. Wita( )-761kvomple(e)11(r)-7( )] TJETQEMC /Span #MCID873/Lang

Sensors) durch vorhandene Dokumentationen erheblich vereinfacht werden können, sofern man sich intensiv einliest.

Nicht zuletzt bereite ich die Kameraintegration vor, da der Loadlifter mit einer HD-Weitwinkelkamera ausgestattet ist. Hier stütze ich mich vor allem auf die offizielle OpenCV-Dokumentation, um zu verstehen, wie man unter Python auf die Kamera zugreift und Bilddaten verarbeitet. Gemäss den OpenCV-Dokumenten wird typischerweise zunächst ein VideoCapture-Objekt erstellt, dem entweder eine Kamera-ID (wie 0 für die Standard-Kamera) oder ein Videodateipfad übergeben wird. Anschliessend liest man in einer Schleife frameweise die Bilder aus (`cap.read()`) und prüft, ob der Lesevorgang erfolgreich ist. Dieses Muster setze ich in einem kleinen Testprogramm auf dem Raspberry Pi um, um die korrekte Funktion der Kamera zu prüfen. Dabei stelle ich auch fest, dass die USB-Kamera des Roboters hohe Auflösungen unterstützt. Durch weitere Recherche (u.a. in Foren und OpenCV-Beispielcode) lerne ich einen Tipp: Wenn man den FourCC-Code des VideoCaptures auf MJPEG stellt, kann die Bildübertragung effizienter erfolgen. In meinem Test bestätigt sich dies, der Kamerastream läuft deutlich flüssiger, nachdem ich das Capture-Format entsprechend angepasst habe. Zusätzlich arbeite ich mich in OpenCV-Funktionen zur Bildverarbeitung ein, zum Beispiel das Umwandeln in Graustufen und das Zeichnen von Markierungen, da diese später für die Objekterkennung wichtig werden. Die OpenCV-Doku ist auch hier ein verlässlicher Leitfaden, der mich Schritt für Schritt durch die nötigen Funktionen führt und Best Practices (wie z. B. das Freigeben der Kamera mit `cap.release()` am Ende) vermittelt. Durch diese Vorarbeit kann ich die Kameraanbindung im Hauptprojekt praktisch ohne Schwierigkeiten umsetzen.

### 3.3.2 Software-Stack-Recherche

Parallel zur Hardware vertiefe ich mich in den Software-Stack, der im Projekt zum Einsatz kommt. Da ich entscheide, einen Grossteil der Logik in Python zu entwickeln (vor allem wegen der guten Bibliotheksunterstützung für Computer Vision und die schnelle Entwicklungszeit), richte ich die Python-Umgebung ein und lerne die relevanten Libraries kennen. Beispielsweise muss ich mit der Bibliothek `smbus2` den I<sup>2</sup>C-Bus des Raspberry Pi ansprechen, um Hardwarekomponenten wie Motorcontroller oder eventuell I<sup>2</sup>C-basiertes Servo-Board zu steuern. Ein Blick in den Hiwonder-Python-Code zeigt den Umgang mit `smbus2` sehr deutlich: Dort wird etwa ein SMBus-Objekt geöffnet und mittels `write_i2c_block_data` an Adresse 0x34 ein Wert an ein Register gesendet, um die Geschwindigkeit eines bestimmten Motors zu setzen. Diesen Ansatz übernehme ich. Auch hier geben kleine Probeprogramme mir die Rückmeldung, dass ich die Technik verstehe. So lässt sich z. B. ein einzelnes Rad gezielt vor- und zurückdrehen, indem ich

entsprechende Werte an das passende Register schicke. Neben smbus2 steht auch WiringPi/Pigpio zur Diskussion, insbesondere falls ich auf niedriger Ebene GPIO-Pins steuern oder PWM-Signale für Servos generieren muss. Durch Forenbeiträge und die Raspberry-Pi-Dokumentation finde ich jedoch heraus, dass für meine Zwecke smbus2 und die vorhandenen Boards ausreichen, da komplexe Aufgaben (wie die PWM-Erzeugung für die Servos) bereits von der mitgelieferten Elektronik übernommen wird. Somit kann ich mich auf höherer Ebene mit Python darauf konzentrieren, die Hardware anzusteuern, ohne selbst zeitkritische Signalroutinen entwickeln zu müssen. Insgesamt nimmt mir die Auseinandersetzung mit dem Software-Stack die Scheu vor der Vielfalt der Technologien: anstatt unvorbereitet vor einer Fülle von unbekannten Bibliotheken zu stehen, weiss ich nun genau, welche Pakete ich brauche und wie ich sie einbinde und nutze.

Ein besonderes Feature des Projekts ist die Objekterkennung mittels KI, wofür ich YOLOv8n von Ultralytics einsetze. Da ich zuvor noch nie ein eigenes neuronales Netz trainiert habe, ist hier eine besonders gründliche Recherche nötig. Glücklicherweise stellt Ultralytics eine ausführliche Dokumentation und Tutorials bereit, die ich Schritt für Schritt durchgehe und auf YouTube gibt es genug Tutorials. Zunächst lerne ich, welche Vorbereitungen für das Training eines eigenen Objekterkennungsmodells nötig sind. Gemäss Ultralytics muss ich einen annotierten Datensatz meiner Zielobjekte erstellen (d.h. Bilder aufnehmen und mit passenden Bounding Boxes labeln), dann eine YAML-Konfigurationsdatei schreiben, die Klassen und Pfade definiert, und schliesslich das Training mit dem entsprechenden Befehl starten. Konkret legte ich beispielsweise eine YAML-Datei an, die meine Objektklasse (für den Greifgegenstand, den der Loadlifter erkennen soll) und die Pfade zu Trainings- und Testbildern enthält. Die Ultralytics-Doku liefert hierfür ein Template, an dem ich mich orientiere. Anschliessend ist es dank der Dokumentation recht einfach, den Trainingsprozess auszuführen. Entweder über den Kommandozeilenauftruf `yolo detect train ...` mit den gewünschten Parametern oder direkt über das Python-API mittels `YOLO().train(...)`. Ich informiere mich im Vorfeld auch darüber, welche Modellvarianten von YOLOv8n verfügbar sind (nanos, small, medium, etc.) und entscheide mich aus Performancegründen für eine leichtere Variante, da die Inferenz später auf dem Raspberry Pi erfolgen soll. Ein wichtiger Lernpunkt aus der Recherche ist zudem der Umgang mit den Ergebnissen des Trainings: Die Dokumentation erläutert, wie man das trainierte Modell abspeichert und anschliessend für die Inferenz, also die Laufzeiterkennung nutzt. Hier kommt mir meine OpenCV-Erfahrung zugute, denn YOLOv8n lässt sich mit wenigen Zeilen Python-Code integrieren, um z. B. ein Kamerabild einzulesen und an das Netzwerk zu übergeben, woraufhin dieses erkannte Objekte samt Positionen zurückliefert. Ultralytics stellt hierfür Codebeispiele

bereit, die ich adaptieren kann. Natürlich gab es auch Stolpersteine: Die enorme Daten- und Rechenintensität des Trainings zwingen mich etwa dazu, dieses auf einem leistungsstarken Rechner durchzuführen, da der Raspberry Pi selbst zu langsam ist. Ich nutzte für das Machine Learning meinen MacBook Pro mit dem M3 Prozessor und 24GB RAM. Alles in allem erlange ich durch die Beschäftigung mit YOLOv8n ein gutes Verständnis für den Workflow von Datenerfassung über Training bis Deployment. Dieses Wissen verhindert, dass ich mich naiv in ein KI-Abenteuer stürze, stattdessen kann ich von Anfang an systematisch vorgehen und typische Fehler (wie falsch formatierte Labels oder eine unpassende Netzwerkarchitektur) vermeiden. Schlussendlich entscheide ich mich jedoch dafür, dass YOLOv8n auf meinem MacBook Pro läuft und die erkannten Objekte an den Raspberry Pi übermittelt. Der Grund dafür ist, dass der Raspberry Pi nicht genügend Rechenleistung besitzt, um gleichzeitig einen flüssigen Videostream aufrechtzuhalten und parallel Inferenzen durchzuführen.

## 3.4 Zielgruppe & Einsatzumgebung

### 3.4.1 Zielgruppe

Primär richtet sich Loadlifter an schulische Demonstrationen und Prototyping im Kontext der Intralogistik: Lehrpersonen/Prüfer:innen, Studierende sowie interessierte Dritte, die autonome Grundfunktionen (Fahren, Finden, Greifen, Rückkehr) nachvollziehen wollen.

Sekundär adressiert das System die Intralogistik in Unternehmen. Von KMU bis hin zu grösseren Firmen als Konzeptstudie (Proof-of-Concept) für einfache Transportaufgaben in klar strukturierten Gängen.

### 3.4.2 Stakeholder

- Operator (Start/Stop, Missionswahl, manueller Abbruch bei Test)
- Beobachter (Demo-Publikum, Lehrpersonen)
- Entwickler (ich) für Betrieb, Debugging und Updates
- Umgebung (Lagermodell mit Regalen, Zielobjekten, Basisstation)

### 3.4.3 Einsatzumgebung (Lagermodell)

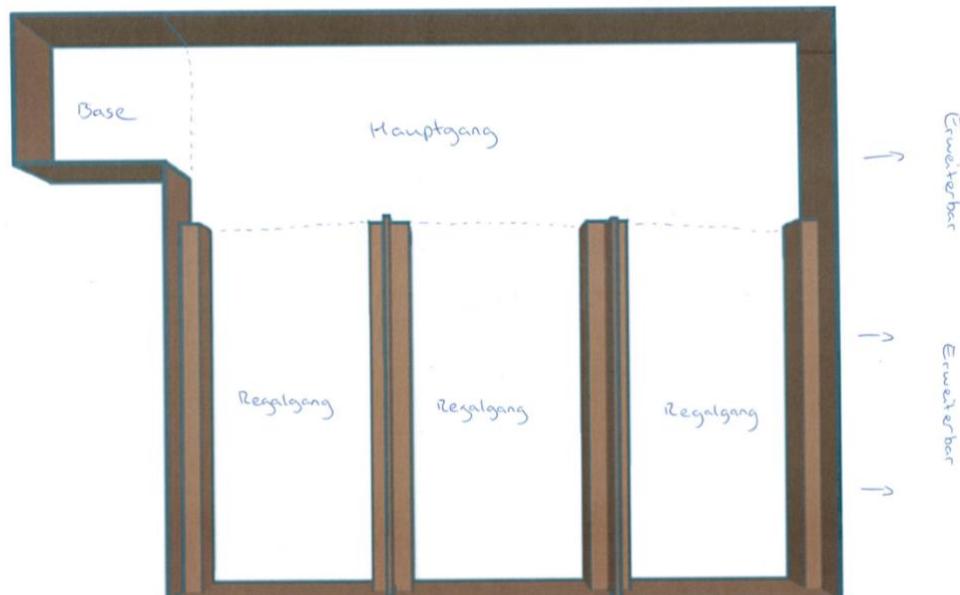


Abb. 5: Lagerplan

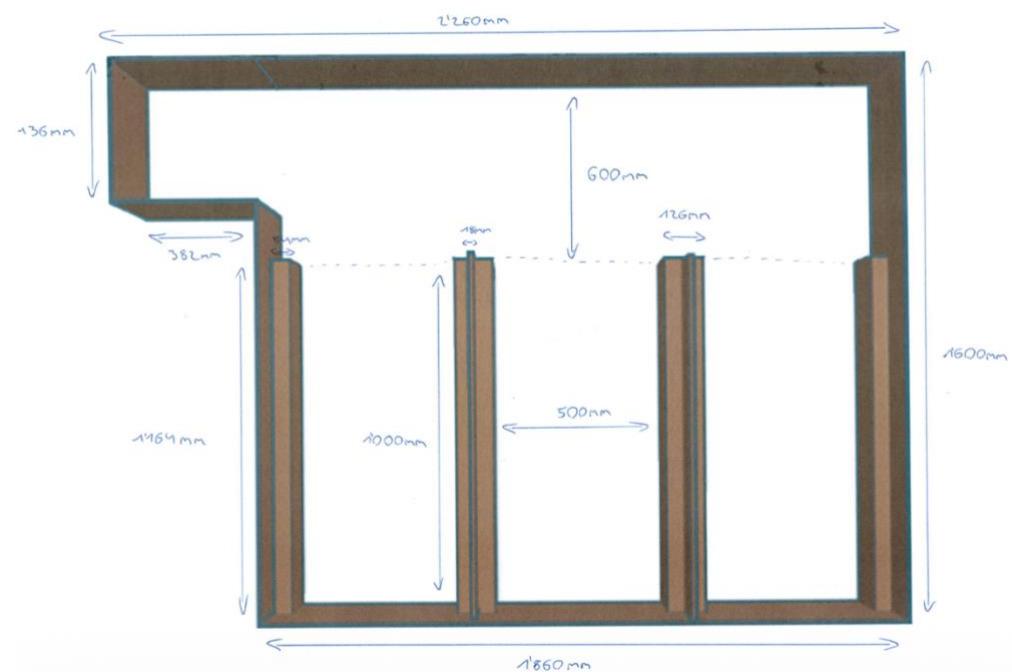


Abb. 6: Ansicht mit Längen und Breiten

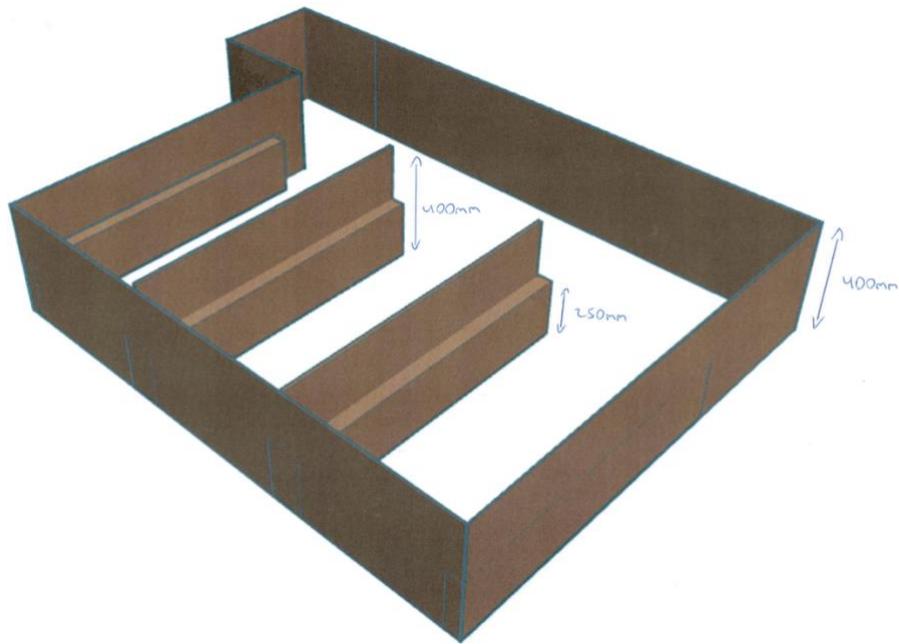


Abb. 7: Ansicht mit Höhen

### 3.4.4 Warum diese Umgebung?

Die strukturierte, kontrollierte Indoor-Umgebung reduziert Störvariablen und erlaubt es, die Kernfrage (autonomer Transport inkl. Greifen auf gegebener Plattform) sauber nachzuweisen, ohne den Umfang durch SLAM/Safety zu sprengen. Gleichzeitig ist das Setting realitätsnah für Lagergänge, sodass Ergebnisse auf einfache Intralogistik-Szenarien übertragbar sind.

## 3.5 IST-Ausstattung

Dieses Kapitel zeigt, was heute real vorhanden ist (IST).

### 3.5.1 Hardware (IST)

- Rechnerplattform: Raspberry Pi 4 (4 GB RAM) mit ArmPi-Pro Board; SD-Karte (32 GB).
- Antrieb/Chassis: Mecanum-Antrieb (Links/rechts Rad), Hiwonder Motor-Treiberplatine.
- Manipulator: Hiwonder Greifarm mit 6 Servos, 1 x Hiwonder LX-225, 4 x Hiwonder LX-15D, 1 x Hiwonder HTS-16L; Ansteuerung erfolgt direkt über das ArmPi-Pro Board (integrierte PWM-Kontrolle, keine separate Servoplatine erforderlich).
- Sensorik:
  - Kamera: Waveshare IMX335 (5MP USB), bis 2K Video bei 30fps, Weitwinkel; DIY-Montage über dem Greifer.
  - 2D-LiDAR: Orbbee Oradar MS200; DIY-Frontmontage.
- Strom: Akkuversorgung (12 V).

### 3.5.2 Software (IST)

- Betriebssystem: Ubuntu 22.04 LTS (64-Bit), Linux Kernel 5.x.
- Zugriff: SSH-Terminal für Deployment/Debug.
- Steuerungs-Software (Python):
  - Core-Module: `threading`, `time`, `sys`, `argparse`, `math`, `pathlib`, `termios`.
  - Peripherie/Bus: `smbus2` (`I2C-Userspace`).
  - Vision/ML: `OpenCV` (`cv2`), `ultralytics` (YOLOv8).
  - Web-UI/Endpoints: `flask`.
- Prozesse/Threads:
  - Navigation-Loop (Pfadfolge, Taktung).
  - Arm-Control (Greifersequenz).
  - Sensor-Reader (LiDAR/Kamera-Grabber).
  - Vision-Worker (Inference auf Frames).
  - Logger

### 3.6 Use-Cases

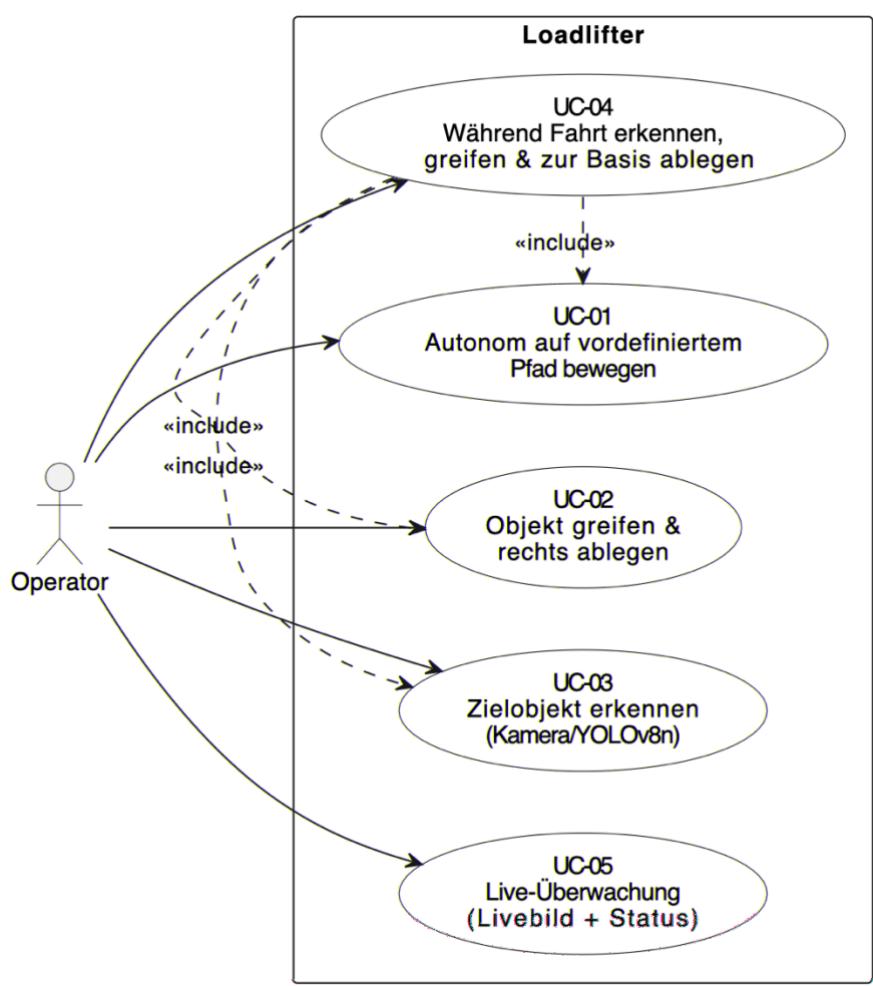


Abb. 8: Use-Case-Diagramm

Die folgenden Unterkapitel beschreiben die einzelnen Use-Cases im Detail.

#### 3.6.1 UC-01: Autonom auf vordefiniertem Pfad bewegen

**Ziel:** Den hinterlegten Pfad (Basis → Hauptkanal → Lagerkanal → ... → zurück zur Basis) zuverlässig abfahren.

**Akteur:** Operator (Start/Stopp).

**System:** Loadlifter

**Vorbedingungen:**

- Pfad liegt vor.
- Akku ausreichend.
- Antrieb, Encoder/Motortreiber initialisiert.

- LiDAR sendet Messwerte.
- Relevante Laufzeitparameter sind gesetzt (robot\_config.json).

**Auslöser:** Startbefehl (SSH).

**Standardablauf:**

1. Initialisieren: Motor- und Sensor-Subsysteme hochfahren Startpose setzen.
2. Segment laden: Nächstes Pfad-Segment aktivieren.
3. Fahren: Vortrieb mit `forward_mm_s`.
4. Online-Korrektur: Solange  $|Yaw| \leq \text{MAX\_YAW}$ , wird mit kleiner Korrektur entdrifft; Querzentrierung via Center-Gain (`Kp_center`).
5. Segment-Ende erkennen: LiDAR-Trigger vor Wand/Stopper bei `front_stop_mm`.
6. Bremsen: Geschwindigkeit runter.
7. Nächstes Segment: ggf. Drehung/Orientierung, weiter mit Schritt 3.
8. Pfad fertig: Endzustand erreicht, Buzzer (`buzzer_end_s`).

**Alternativen/Fehler:**

- Pfad unvollständig → kontrollierter Abbruch.
- LiDAR/Encoder fehlen → Stopp und Meldung.
- Remote-Abort (Operator) → sofortiger Stopp.

**Nachbedingungen:**

- Roboter steht stabil in der Basis.
- Status im Log persistiert.

**Erfolgskriterien:**

- Positionsgenauigkeit: End-Stopp pro Segment innerhalb `TOL_MM` (5mm).
- LiDAR-Stopp: Fall Wand-Stopp aktiv, Anhalten bei `front_stop_mm` ± 1-2 cm.
- Dynamik: Segmentlaufzeit in erwartetem Korridor (Konstanttempo `forward_mm_s`).

### 3.6.2 UC-02: Objekt greifen und rechts ablegen (reine Arm-Sequenz)

**Ziel:** Ein Objekt mechanisch aufnehmen, rotieren und rechts ablegen.

**Akteur:** Operator (Start/Stopp).

**Vorbedingungen:**

- Servos initialisiert.
- Roboter steht still.
- Objekt liegt greifbar.
- Um den Roboter ist alles frei.

**Auslöser:** Startbefehl (SSH).

**Standardablauf:**

1. Arm in Homeposition.
2. Arm in aufrechter Position.
3. Greifer öffnen.
4. Arm beugt sich zum Objekt.
5. Greifer schliessen.
6. Arm wieder in aufrechte Position.
7. Arm-Basis rotiert 90° nach rechts.
8. Arm beugt sich.
9. Greifer öffnen.
10. Arm in Homeposition.

**Alternativen/Fehler:** Objekt nicht sicher gefasst → Objekt besser platzieren.

**Nachbedingungen:**

- Objekt in Ablagezone.
- Arm in Homeposition wieder bereit.

**Erfolgskriterien:** Sauberer Greif- und Ablagevorgang.

### 3.6.3 UC-03: Zielobjekt erkennen (Kamera/ML)

**Ziel:** Laufenden Videostream mit YOLOv8n inferieren und Zielobjekte (4 Klassen/Farben) erkennen.

**Akteur:** Operator.

**Vorbedingungen:**

- Kamera verbunden/aktiv; Belichtung ausreichend.
- Modellgewichte geladen; Inferenz-Pipeline lauffähig.

**Auslöser:** Startbefehl (SSH)

**Standardablauf:**

1. Videostream öffnen und Frames beziehen.
2. Pro Frame Inferenz (YOLOv8n).
3. Treffer nach Confidence-Schwelle filtern; beste Box je Klasse wählen.
4. Ergebnis publizieren (Klasse, Konfidenz, Bounding Box) und optional Overlay anzeigen/loggen.

**Alternativen/Fehler:**

- Kamera nicht verfügbar → Retry/Fehlermeldung.

- Zu dunkle Szene → Auto-Gain/Belichtung erhöhen, Confidence anpassen.
- Modell nicht ladbar → Abbruch & Log.

**Nachbedingungen:**

- Kamera freigegeben.
- Modell entladen.
- Logs gespeichert.

**Erfolgskriterien:** Korrekte Klassifikation  $\geq 95\%$  (Confidence-Wert muss  $\geq 80\%$  sein, dass ein Objekt als Lego-Duplo-Stein durchgeht).

**3.6.4 UC-04: Zielobjekt während der autonomen Fahrt erkennen, greifen und zur Basis transportieren & ablegen**

**Ziel:** Während der Fahrt ein Zielobjekt per Kamera/ML erkennen, anfahren, greifen und zur Basis bringen.

**Akteure:** Operator.

**Vorbedingungen:** Kamera-Stream aktiv; Inferenz lauffähig; Lagermodell keine Hindernisse.

**Auslöser:** Startbefehl (SSH).

**Standardablauf:**

1. Pfad geführt fahren; parallel Frames inferieren (YOLO).
  2. Objekt erkannt → Zielpunkt/Andockposk
- I°Zit  |p

2. Optional: Start/Stopp-Kommandos senden.

#### Alternativen/Fehler:

- Stream bricht ab.
- Logs stimmen nicht.

**Nachbedingungen:** Keine Zustandsänderung am Roboter.

#### Erfolgskriterien:

- Flüssiger Stream (für Raspberry-Pi-Verhältnisse).
- Saubere Logs.

## 3.7 Eigene Entscheide

### 3.7.1 Sensorwahl: 2D-LiDAR statt Ultraschall oder 3D-LiDAR

#### Ausgangslage:

Für die Pfadfolge und das zuverlässige Stoppen an definierten Punkten brauche ich exakte Distanzen in einer Ebene (Lagergang). Aus dem Theorieteil: LiDAR misst per Time-of-Flight präzise und schnell; Ultraschall ist im Nahbereich robust, aber weniger genau und störanfällig; 3D-LiDAR liefert Punkt wolken, ist aber teuer und rechenintensiv.

#### Nutzwertanalyse:

Kriterium	Gewicht	2D-LiDAR	m. G.	Ultraschall	m. G.	3D-LiDAR	m. G.
Messgenauigkeit	4	8	32	4	16	10	40
Robustheit ggü. Störungen	2	8	16	5	10	9	18
Reichweite	2	7	14	4	8	9	18
Integrationsaufwand (SW/SDK)	2	7	14	8	16	4	8
Kosten	3	6	18	9	27	2	6
Daten-/Rechenaufwand	1	7	7	9	9	3	3
Verfügbarkeit/Support	1	7	7	9	9	4	4
Montage/Platzbedarf	1	7	7	9	9	5	5
<b>Summe</b>			<b>115</b>		<b>104</b>		<b>102</b>

**Entscheid:** 2D-LiDAR gewinnt (115-Punkte), weil er eine sehr gute Genauigkeit hat bei moderatem Aufwand und Preis, ideal für strukturierte Gänge. Ultraschall wäre günstiger, liefert aber zu wenig metrische Genauigkeit. 3D-LiDAR wäre Overkill (Kosten, Rechenlast) für dieses MA-Scope.

### 3.7.2 Technologie-Stack: Python statt C++

#### Rahmenbedingungen:

Hiwonder-Quellcode bestand zu 100 % aus Python mit ROS 2, LiDAR-Quellcode zu 100 % aus C++. Ziel der MA: funktionierender Prototyp, kurze Iterationszyklen, viel Testen.

#### Nutzwertanalyse:

Kriterium	Gewicht	Python	m. G.	C++	m. G.
Entwicklungszeit/Produktivität	3	9	27	5	15
Treiber/Beispiele für meine HW	2	9	18	5	10
Performance/Echtzeitfähigkeit	2	5	10	9	18
Lernkurve/Komplexität	3	9	27	4	12
CV/ML-Integration (OpenCV/YOLO)	2	9	18	7	14
Stabilität/Deployment auf dem Pi	1	7	7	8	8
Risiko/Fehlerquellen	1	7	7	6	6
<b>Summe</b>			<b>114</b>		<b>83</b>

#### Entscheid:

Bare Python gewinnt klar, weil es schnelleres Prototyping ermöglicht, reichlich vorhandene Bibliotheken und Beispiele für die Hiwonder-Plattform hat. Mit den Trade-offs wie weniger Roh-Performance als C++ kann man leben.

### 3.7.3 Technologie-Stack: Bare Python vs. ROS 2 (mit Python)

#### Rahmenbedingungen:

ROS 2 ist ein leistungsfähiges Middleware-Framework, das speziell für modulare und skalierbare Robotersysteme entwickelt wurde. Es bietet Komponenten wie Topics, Nodes, Services, Parameter-Server und eine standardisierte Kommunikation zwischen Modulen, was ideal für grosse, verteilte Systeme ist.

Für ein einzelnes System wie den Loadlifter (Raspberry Pi 4 + Hiwonder-Chassis) stellt sich jedoch die Frage, ob der zusätzliche Architektur-Overhead gerechtfertigt ist.

Das Ziel der Maturaarbeit ist ein funktionsfähiger, stabiler Prototyp mit überschaubarem Entwicklungsaufwand und klarer Dokumentation. Daher steht die Entscheidung zwischen einer reinen („bare“) Python-Implementierung und ROS 2 mit Python-Bindings (`rclpy`) im Fokus.

#### Nutzwertanalyse:

Kriterium	Gewicht	Bare Python	m. G.	ROS 2 (Python)	m. G.
Entwicklungsaufwand/Einstieg	3	9	27	5	15
Stabilität auf dem Raspberry Pi (OS/Memory)	2	8	16	6	12
Kommunikation/Modularität	3	6	18	9	27
Debugging/Fehleranalyse	2	9	18	6	12
Performance/Latenz	2	8	16	7	14
Wartbarkeit/Erweiterbarkeit	1	7	7	9	9
Abhängigkeit/Installationskomplexität	1	9	9	5	5
<b>Summe</b>			<b>111</b>		<b>94</b>

#### Entscheid:

Bare Python gewinnt mit 111 Punkten gegenüber ROS 2 (Python, 94 Punkte).

Für das Ziel der Maturaarbeit, eine lauffähige, übersichtliche Robotersteuerung mit klar nachvollziehbarer Logik, ist eine direkte Python-Implementierung ohne ROS 2-Schichten die sinnvollere Wahl. Sie reduziert Setup-Zeit, Fehlerquellen und Ressourcenverbrauch, was auf dem Raspberry Pi besonders relevant ist.

ROS 2 wäre dann interessant, wenn das System in Zukunft erweitert oder vernetzt werden soll (z. B. mehrere Roboter, zentrale Kommunikation über Topics). Für einen einzelnen Roboter im Laborumfeld überwiegt jedoch der Nutzen einer leichten, direkten Architektur in Python.

### 3.7.4 Kameraauswahl: USB-UVC statt MIPI/CS

Die mitgelieferte Kamera ging kaputt. Ich habe sie durch eine Waveshare IMX335 USB-Kamera ersetzt.

#### Gründe:

- Plug-and-play (UVC): läuft sofort als `/dev/video*`, problemlos mit `cv2.VideoCapture(...)`.
- Direkter USB-A-Anschluss: kein Adapter/Flachband, einfache DIY-Montage über dem Greifer.
- Verfügbarkeit/Lieferzeit: zügige Beschaffung (lokaler Händler).
- Bildqualität: 5MP und 2K@30 fps → super für YOLO-Detektion.

### 3.7.5 Objekterkennung: YOLOv8n

Ich nutze YOLOv8n, weil es auf dem Raspberry Pi echtzeitnah läuft und auch super auf dem MacBook Pro funktioniert, wenn man auf dem Raspberry Pi noch Performance freihalten will für andere Prozesse. Außerdem ist es einfach zu trainieren (eigenes Dataset mit Labels) und es bietet gute Genauigkeit für vier Zielklassen (Duplo-Farben).

### 3.7.6 DIY-Montage & Verkabelung

- LiDAR (Frontmontage, lösbar): Mit doppelseitigem Klebeband auf Fronträger befestigt. Hält super, bleibt aber rückstandsfrei demontierbar (Sensorwechsel/Justage). Frontposition bietet ein freies Sichtfeld entlang des Ganges (ca. 270°).
- Kamera (über dem Greifer): Auf dem Originalhalter montiert, da dieser zu gross war, wurden neue Befestigungsbohrungen gesetzt.
- Kabelmanagement & Stromversorgung: Motorverkabelung unten bündig und fixiert geführt. LiDAR-USB nahe beim Raspberry Pi verlegt. Kamerakabel entlang des Arms mit Service-Loop an Gelenken, dann unter der Basis zum Pi. Akku (12 V) seitlich unten mit Klettstreifen befestigt → schnell entnehmbar zum Laden.

### 3.8 Test-Snippets

```
#!/usr/bin/env python3
# test_chassis.py
# turn the wheel (ID 1) forward at a speed of 40

import time
from smbus2 import SMBus

I2C_PORT = 1          # bus
ADDR = 0x34           # adress for motorcontroller
REG_MOTOR1 = 51        # register of the wheel
TEST_PULSE = 40        # forward momentum
DURATION_S = 5.0       # duration in seconds

def main() -> None:
    print("[INFO] Starting the test (Motor 1).")
    try:
        with SMBus(I2C_PORT) as bus:
            bus.write_i2c_block_data(ADDR, REG_MOTOR1, [TEST_PULSE &
0xFF])
            time.sleep(DURATION_S)
            bus.write_i2c_block_data(ADDR, REG_MOTOR1, [0])
    except FileNotFoundError:
        print("[ERROR] I2C-Device not found (/dev/i2c-1).")
    except PermissionError:
        print("[ERROR] No permission for I2C-Access.")
    except Exception as exc:
        print(f"[ERROR] Error while writing to the motorcontroller: {exc}")
    else:
        print("[INFO] Test finished, motor stopped.")

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

```

#!/usr/bin/env python3
# test-LiDAR.py
# Reads frames, parses angle/distance/intensity, checks CRC, prints
# the first few points.

import serial
import struct
import time
from collections import deque

PORT = "/dev/ttyUSB0"      # USB serial device
BAUD = 230400              # baudrate
TIMEOUT = 0.05

# CRC8 table copied from the original C++ driver (same ordering)
CRC8_TABLE = [
    0x00, 0x4d, 0x9a, 0xd7, 0x79, 0x34, 0xe3, 0xae, 0xf2, 0xbf, 0x68, 0x25, 0x8b, 0xc6,
    0x11, 0x5c, 0xa9, 0xe4, 0x33, 0x7e, 0xd0, 0x9d, 0x4a, 0x07, 0x5b, 0x16, 0xc1, 0x8c,
    0x22, 0x6f, 0xb8, 0xf5, 0x15, 0x58, 0x8f, 0xc2, 0x6c, 0x21, 0xf6, 0xbb, 0xe7, 0xaa,
    0x7d, 0x30, 0x9e, 0xd3, 0x04, 0x49, 0xbc, 0xf1, 0x26, 0x6b, 0xc5, 0x88, 0x5f, 0x12,
    0x4e, 0x03, 0xd4, 0x99, 0x37, 0x7a, 0xad, 0xe0, 0x2a, 0x67, 0xb0, 0xfd, 0x53, 0x1e,
    0xc9, 0x84, 0xd8, 0x95, 0x42, 0x0f, 0xa1, 0xec, 0x3b, 0x76, 0x83, 0xce, 0x19, 0x54,
    0xfa, 0xb7, 0x60, 0x2d, 0x71, 0x3c, 0xeb, 0xa6, 0x08, 0x45, 0x92, 0xdf, 0x3f, 0x72,
    0xa5, 0xe8, 0x46, 0x0b, 0xdc, 0x91, 0xcd, 0x80, 0x57, 0x1a, 0xb4, 0xf9, 0x2e, 0x63,
    0x96, 0xdb, 0x0c, 0x41, 0xef, 0xa2, 0x75, 0x38, 0x64, 0x29, 0xfe, 0xb3, 0x1d, 0x50,
    0x87, 0xca, 0x54, 0x19, 0xce, 0x83, 0x2d, 0x60, 0xb7, 0xfa, 0xa6, 0xeb, 0x3c, 0x71,
    0xdf, 0x92, 0x45, 0x08, 0xfd, 0xb0, 0x67, 0x2a, 0x84, 0xc9, 0x1e, 0x53, 0x0f, 0x42,
    0x95, 0xd8, 0x76, 0x3b, 0xec, 0xa1, 0x41, 0x0c, 0xdb, 0x96, 0x38, 0x75, 0xa2, 0xef,
    0xb3, 0xfe, 0x29, 0x64, 0xca, 0x87, 0x50, 0x1d, 0xe8, 0xa5, 0x72, 0x3f, 0x91, 0xdc,
    0x0b, 0x46, 0x1a, 0x57, 0x80, 0xcd, 0x63, 0x2e, 0xf9, 0xb4, 0x7e, 0x33, 0xe4, 0xa9,
    0x07, 0x4a, 0x9d, 0xd0, 0x8c, 0xc1, 0x16, 0x5b, 0xf5, 0xb8, 0x6f, 0x22, 0xd7, 0x9a,
    0x4d, 0x00, 0xae, 0xe3, 0x34, 0x79, 0x25, 0x68, 0xbf, 0xf2, 0x5c, 0x11, 0xc6, 0x8b,
    0x6b, 0x26, 0xf1, 0xbc, 0x12, 0x5f, 0x88, 0xc5, 0x99, 0xd4, 0x03, 0x4e, 0xe0, 0xad,
    0x7a, 0x37, 0xc2, 0x8f, 0x58, 0x15, 0xbb, 0xf6, 0x21, 0x6c, 0x30, 0x7d, 0xaa, 0xe7,
    0x49, 0x04, 0xd3, 0x9e
]

def crc8(data: bytes, init_val=0x00):
    crc = init_val
    for b in data:
        crc = CRC8_TABLE[crc ^ b]
    return crc

def read_exact(ser, n):
    buf = bytearray()
    while len(buf) < n:

```

```
chunk = ser.read(n - len(buf))
if not chunk:
    return None
buf.extend(chunk)
return bytes(buf)

def sync_to_header(ser):
    while True:
        b = ser.read(1)
        if not b:
            return False
        if b[0] == 0x54:
            return True

def parse_frame(rest: bytes):
    # buffer starts with ver_len
    ver_len = rest[0]
    count = ver_len & 0x1F  # low 5 bits = number of points
    if count == 0 or count > 40:
        return None

    need = 1 + 2 + 2 + count * 3 + 2 + 2 + 1
    if len(rest) < need:
        return None

    speed = struct.unpack_from("<H", rest, 1)[0] / 100.0
    start_raw = struct.unpack_from("<H", rest, 3)[0]
    start_angle = start_raw / 100.0

    off = 5
    dist_mm = []
    inten = []
    for _ in range(count):
        d = struct.unpack_from("<H", rest, off)[0]
        dist_mm.append(d)
        inten.append(rest[off + 2])
        off += 3

    end_raw = struct.unpack_from("<H", rest, off)[0]
    end_angle = end_raw / 100.0
    off += 2
```

```
timestamp = struct.unpack_from("<H", rest, off)[0]
off += 2

crc_byte = rest[off]

# CRC check includes header byte (0x54) plus all payload bytes
before CRC
calc = crc8(bytes([0x54]) + rest[:-1])

# angle interpolation
if count > 1:
    diff = end_angle - start_angle
    # handle wrap-around
    if diff < -180:
        diff += 360
    elif diff > 180:
        diff -= 360
    step = diff / (count - 1)
else:
    step = 0.0

angles = [(start_angle + i * step) % 360 for i in range(count)]

return {
    "speed_dps": speed,
    "start_angle": start_angle,
    "end_angle": end_angle,
    "angles_deg": angles,
    "dist_mm": dist_mm,
    "intensity": inten,
    "timestamp_ms": timestamp,
    "crc_ok": (calc == crc_byte),
    "points": count,
}

def main():
    print(f"[INFO] -> Opening {PORT} @ {BAUD} baud ...")
    ser = serial.Serial(PORT, BAUD, timeout=TIMEOUT)
    time.sleep(0.2)

    frames = 0
    last_log = time.time()
```

```

recent = deque(maxlen=3)

try:
    while True:
        if not sync_to_header(ser):
            print("[ERROR] No header found (timeout).")
            continue

        # read minimal header to discover point count
        head_rest = read_exact(ser, 1 + 2 + 2)  # ver_len + speed
+ start angle
        if head_rest is None:
            continue

        ver_len = head_rest[0]
        count = ver_len & 0x1F
        need = count * 3 + 2 + 2 + 1
        body = read_exact(ser, need)
        if body is None:
            continue

        frame = parse_frame(head_rest + body)
        if not frame:
            continue

        frames += 1
        recent.append(frame)

        if time.time() - last_log >= 0.5:
            f = recent[-1]
            print(
                f"[INFO] \nFrame #{frames} | pts={f['points']:02d}"
                f" | speed={f['speed_dps']:.1f}°/s | "
                f"[INFO] {f['start_angle']:.2f}° ->"
                f"{f['end_angle']:.2f}° | CRC={'ok' if f['crc_ok'] else 'BAD'}"
            )
            for i in range(min(6, f['points'])):
                print(
                    f"[INFO] {i:02d}:"
                    f"angle={f['angles_deg'][i]:6.2f}° dist={f['dist_mm'][i]:5d} mm"
                    f"I={f['intensity'][i]:3d}"
                )
            last_log = time.time()

```

```
finally:  
    ser.close()  
  
if __name__ == "__main__":  
    main()
```

```
#!/usr/bin/env python3
# test_arm.py
# opening, closing and opening the gripper again

import time
import serial

PORT = "/dev/serial0"
BAUD = 115200
SERVO_ID = 1           # gripper
MOVE_TIME_MS = 600
GRIPPER_OPEN_PULSE = 32
GRIPPER_CLOSE_PULSE = 607

def _checksum(buf: bytearray) -> int:
    total = sum(buf) - 0x55 - 0x55
    return (~total) & 0xFF

def build_move_packet(pulse: int, duration_ms: int) -> bytes:
    pulse = max(0, min(1000, pulse))
    duration_ms = max(0, min(30000, duration_ms))
    frame = bytearray([
        0x55,
        0x55,
        SERVO_ID & 0xFF,
        7,
        1,   # LOBOT_SERVO_MOVE_TIME_WRITE
        pulse & 0xFF,
        (pulse >> 8) & 0xFF,
        duration_ms & 0xFF,
        (duration_ms >> 8) & 0xFF,
    ])
    frame.append(_checksum(frame))
    return bytes(frame)

def send_gripper_command(pulse: int) -> None:
    pkt = build_move_packet(pulse, MOVE_TIME_MS)
    with serial.Serial(PORT, baudrate=BAUD, timeout=0.5,
write_timeout=0.5) as ser:
        ser.write(pkt)
        time.sleep(MOVE_TIME_MS / 1000.0 + 0.2)
```

```
def main() -> None:
    print("[INFO] starting.")
    try:
        print("[INFO] -> opening...")
        send_gripper_command(GRIPPER_OPEN_PULSE)
        print("[INFO] -> closing...")
        send_gripper_command(GRIPPER_CLOSE_PULSE)
        print("[INFO] -> opening again...")
        send_gripper_command(GRIPPER_OPEN_PULSE)
    except serial.SerialException as exc:
        print(f"[ERROR] Serial connection failed: {exc}")
    except Exception as exc:
        print(f"[ERROR] Error sending gripper command: {exc}")
    else:
        print("[INFO] Gripper test completed.")

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

```
#!/usr/bin/env python3
# test_camera_stream.py
# Simply shows the camera live stream.

import cv2

def main():
    cap = cv2.VideoCapture(0)  # 0 = first camera
    if not cap.isOpened():
        print("Could not open camera!")
        return

    print("[INFO] Live stream running - press 'q' to quit.")

    while True:
        ret, frame = cap.read()
        if not ret:
            print("[WARN] No frame captured...")
            continue

        cv2.imshow("Camera Live Stream", frame)

        # Check key
        if cv2.waitKey(1) & 0xFF == ord("q"):
            break

    cap.release()
    cv2.destroyAllWindows()

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

### 3.9 Aufwand, Zeitplanung & Kostenschätzung

Ausgangsbasis ist eine erste Gesamtschätzung von 120 Stunden. Die Verteilung nach Arbeitspaketen:

Phase A – Analyse	20 h
Phase B – HW-Integration inkl. DIY	10 h
Phase C – Implementation Navigation	30 h
Phase D – Implementation Arm	10 h
Phase E – Implementation Vision, ML	20 h
Phase F – Tests & Inbetriebnahme	10 h
Phase G – Dokumentation	20 h
<b>Summe</b>	<b>120 h</b>

Berechnung mit 120 h und dem typischen Junior Software Engineer Stundensatz von CHF 80  
brutto:  $120 \text{ h} \times \text{CHF } 80 = \text{CHF } 9'600$ .

## 4 Design

Dieses Kapitel beschreibt die Architektur von Loadlifter und begründet den modularen Aufbau in klar getrennten Schichten. Ziel ist hohe Wartbarkeit, Testbarkeit und Erweiterbarkeit bei minimaler Kopplung zwischen Hardwaredtreibern und Logik.

### 4.1 Systemübersicht

Das System ist in logisch getrennte Pakete gegliedert: cli, control, control.modes, high\_level, low\_level, low\_level.io, camera\_bridge und lidar\_stream. Die Übersicht zeigt nur Beziehungen zwischen Subsystemen. Klassen und Methoden sind bewusst ausgeblendet, um die kognitive Last gering zu halten.

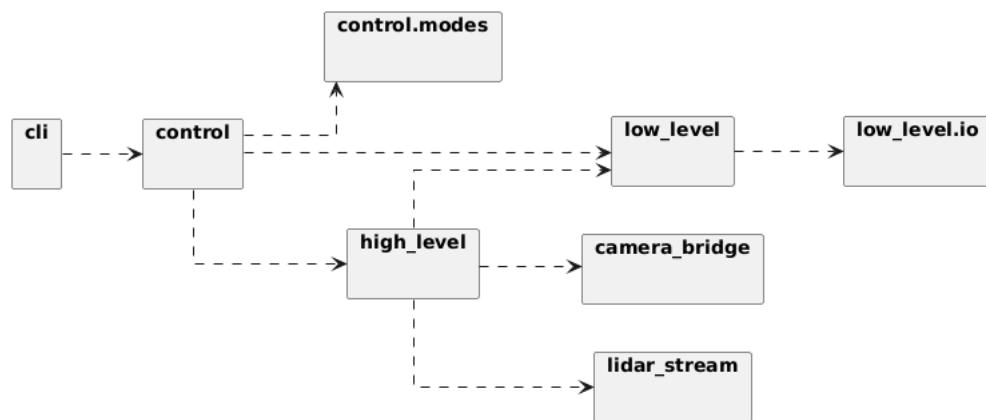


Abb. 9: Systemübersicht

**Entwurfsentscheid:** High-Level-Logik bleibt stabil, auch wenn Sensoren oder Aktuatoren ausgetauscht werden. Die Kommunikation erfolgt über schmale und stabile Schnittstellen.

### 4.2 Control-Ebene

Das ControlSystem ist die zentrale Orchestrierungsschicht. Es liest Konfigurationen, initialisiert Subsysteme und startet Betriebsmodi wie RemoteMode, FollowWallMode, FollowRouteMode und DefinedRouteGetObjectTopMode. Die Modi kapseln jeweils eine Aufgabe (Single-Responsibility) und interagieren mit High-Level-Systemen ausschliesslich über deren öffentliche APIs.

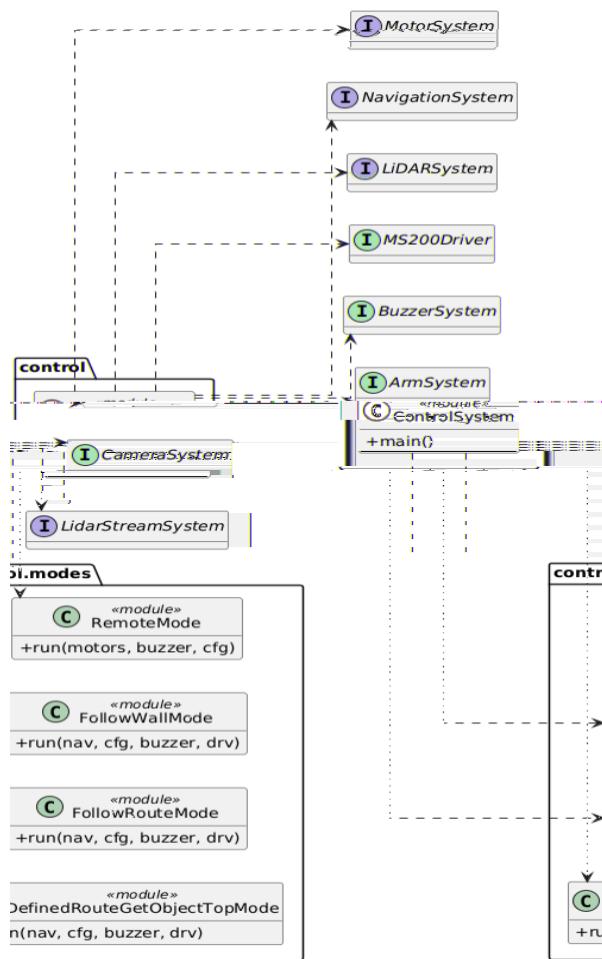


Abb. 10: Control-Ebene

### Vorteile:

- Erweiterbarkeit: Neue Modi können hinzugefügt werden, ohne Kernlogik zu ändern.
- Testbarkeit: Jeder Modus ist isoliert startbar und kann mit Mocks verifiziert werden.

## 4.3 High-Level-Ebene

Die High-Level-Schicht stellt domänen spezifische Funktionen bereit und vermeidet direkte Hardwarezugriffe:

- ArmSystem: Greif- und Posenlogik (z. B. `open_gripper`, `move_pose_deg`, `home`).
- MotorSystem: Fahrmanöver mit Mecanum-Rädern sowie stopp.
- NavigationSystem: Wandfolge, Zentrierung, Rotationen, zeitgesteuertes Fahren. Nutzt nur Distanz- und Punktwolkenabfragen.
- LiDARSystem: Distanzfunktionen (Front/Links/Rechts), Zugriff auf Punktwolken.

- CameraSystem: Start/Stopp von Streams, CLI-Integration.
- BuzzerSystem: akustische Signale (Für Start/Stopp Signal).
- LidarStreamSystem: Netzwerk-Streaming für Telemetrie/Debug.

Zwischen den High-Level-Komponenten bestehen nur fachlich notwendige Abhängigkeiten, z. B. NavigationSystem → MotorSystem und NavigationSystem → LiDARSystem.

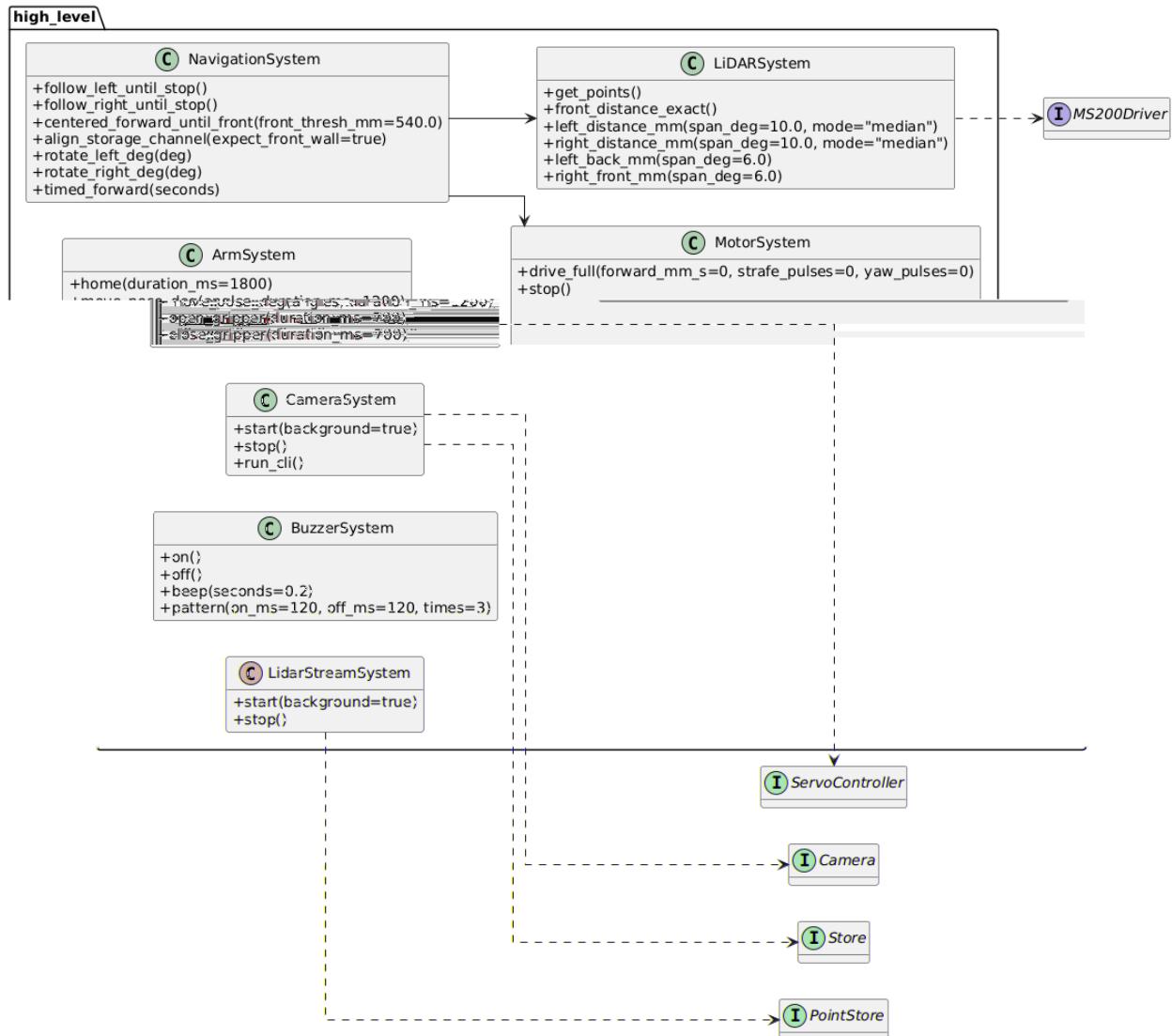


Abb. 11: High-Level-Ebene

**Prinzip:** High-Level kennt keine Hardwaredetails; Low-Level wird über klar definierte Interfaces angesprochen (z. B. ServoController, MS200Driver). Dadurch bleiben Austauschbarkeit und Mocking einfach.

## 4.4 Low-Level-Ebene und I/O-Abstraktion

Die Low-Level-Treiber sprechen direkt mit der Hardware:

- MotorController: Basiskommandos für Fahren, Drehen, Strafen.
- ServoController: Ansteuerung der Arm-Servos (Move/Stop/Read/Load/Unload).
- MS200Driver: LiDAR-Zugriff über UART; liefert Punktwolken/Distanzwerte.
- BuzzerDriver: GPIO/PWM-basierte Signale.

Die IO-Abstraktionen in `low_level.io` (UARTPort, GPIOPin, PWMChannel) entkoppeln die Treiber von der konkreten Plattform. So kann die gleiche Software auf unterschiedlichen Raspberry-Pi-Generationen oder alternativen Boards laufen.

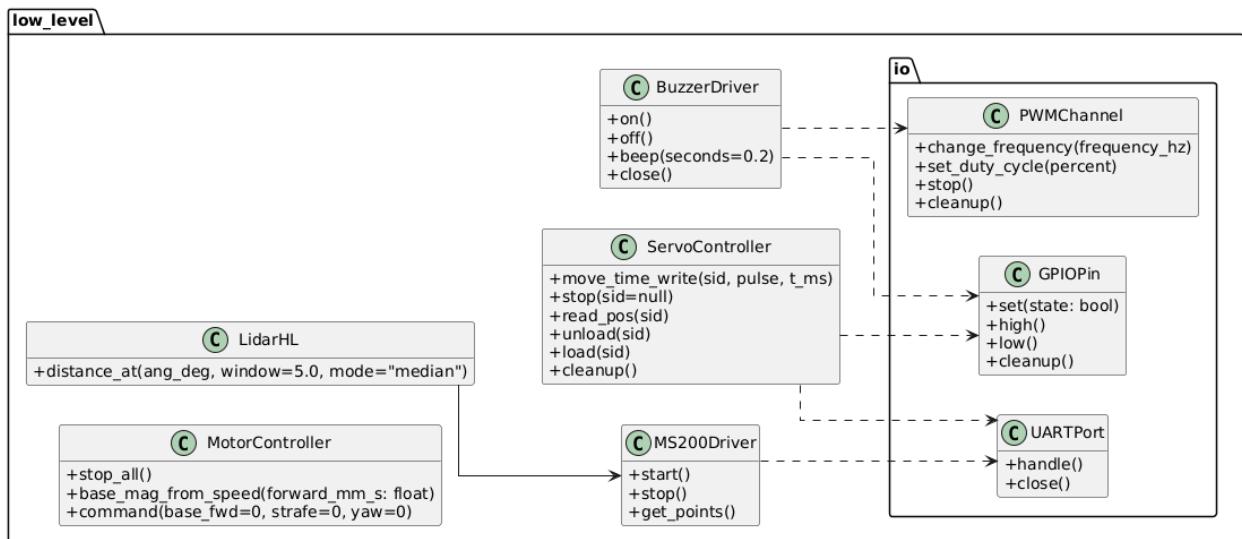


Abb. 12: Low-Level-Ebene und I/O-Abstraktion

## 4.5 Peripherie-Subsysteme

camera\_bridge stellt eine OpenCV-basierte Camera sowie einen Store zum Puffern von JPEG-Overlays bereit. lidar\_stream liefert mit PointStore eine periodisch aktualisierte Punktwolken-Pufferung für Streaming und Remote-Debugging.

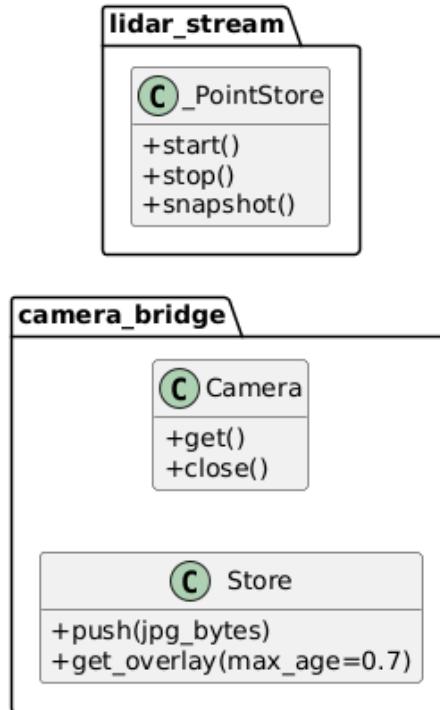


Abb. 13: Peripherie-Subsysteme

Diese Subsysteme bilden die Brücke zwischen Sensorik und High-Level-Logik und ermöglichen paralleles Monitoring (z. B. via Web-Overlay) ohne die Missionslogik zu blockieren.

## 4.6 Sequenzdiagramme

### 4.6.1 FollowRoute

Der Modus führt den Roboter entlang einer vordefinierten Route. Er kombiniert LiDAR-basierte Zentrierung/Kollisionserkennung mit expliziten Rotationen und Zeitfahrten. Ereignisse wie `front_stop` oder `right_open` beeinflussen die nächsten Schritte (z. B. Korrigieren, Zentrieren, Fortsetzen).

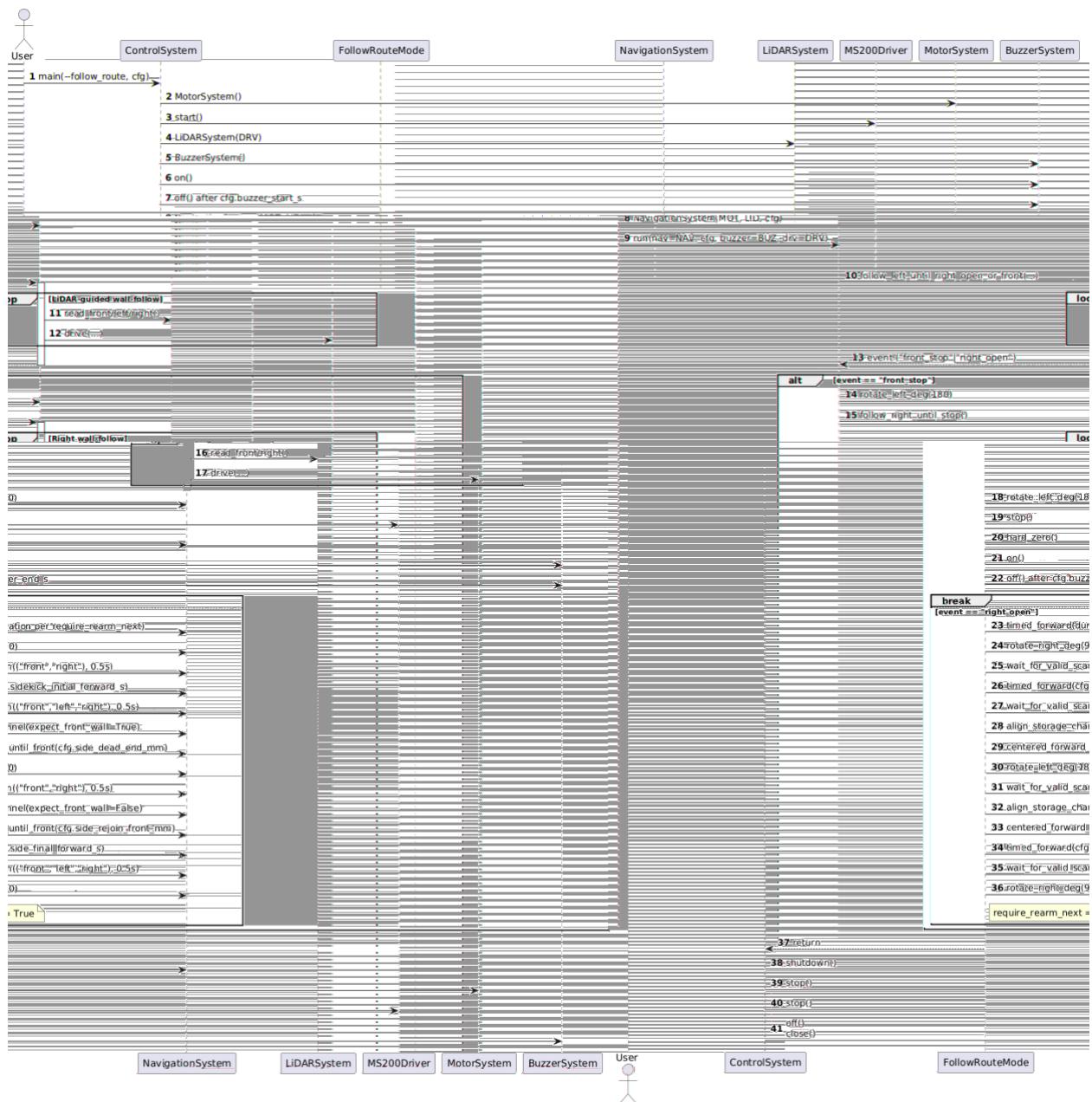


Abb. 14: FollowRoute Sequenz

#### 4.6.2 DefinedRouteGetObjectTop

Dieser Modus fährt eine Route ab, richtet sich am Lagerkanal aus, führt eine Greifsequenz aus (Öffnen → Anfahren Top-Pose → Schliessen → Anheben) und legt das Objekt an Zielposition ab, bevor der Arm auf home fährt.

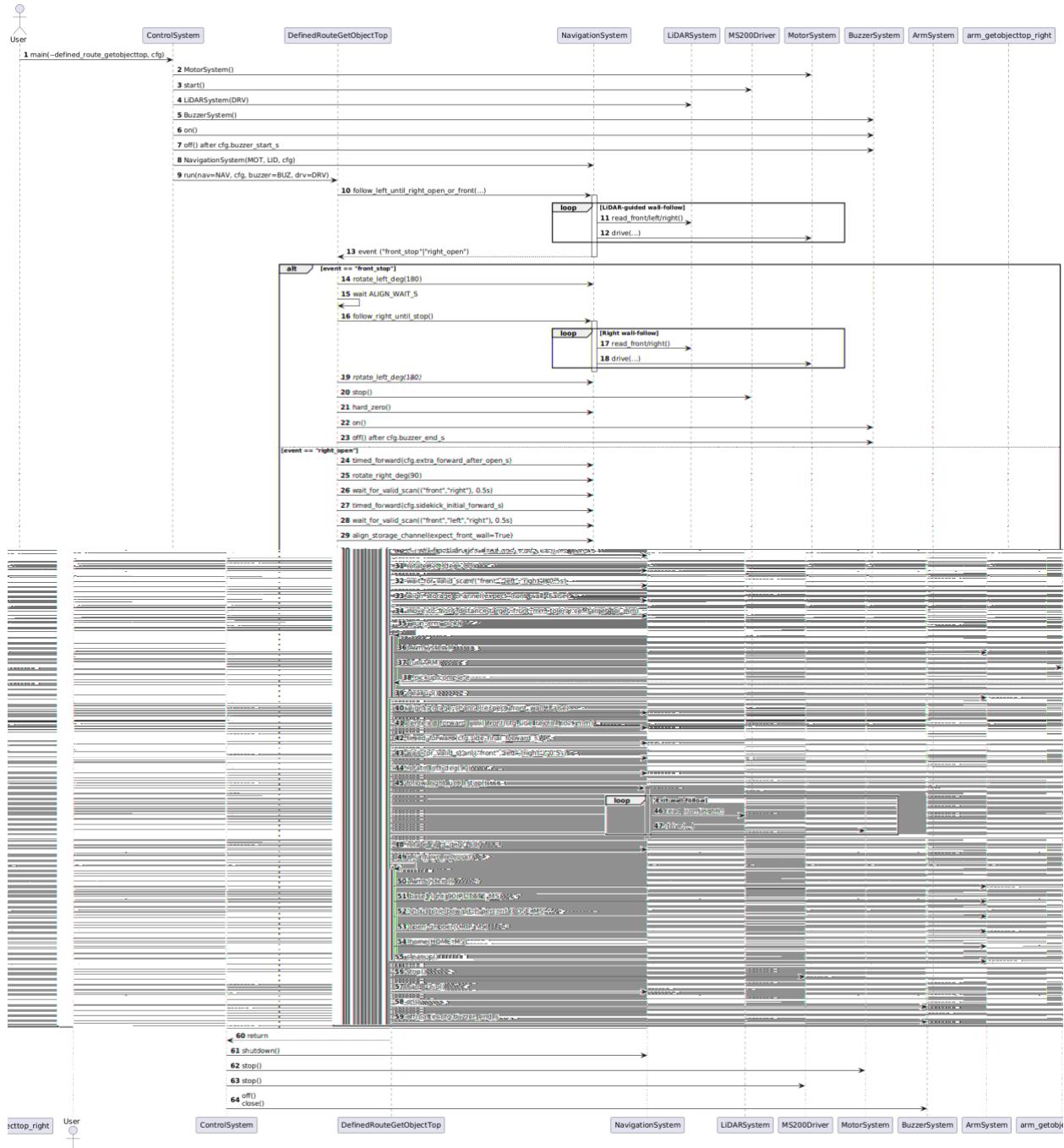


Abb. 15: `DefinedRouteGetObjectTop` Sequenz

## 5 Implementation

Dieses Kapitel ist bewusst als einfaches Logbuch aufgebaut. Eine ausführliche Erläuterung zu jedem einzelnen Schritt würde den Rahmen sprengen, in dieser Form bleibt es jedoch am übersichtlichsten. Es zeigt den Ablauf meiner praktischen Arbeit klar und chronologisch. Ausserdem täuscht die Länge dieses Kapitels, da der eigentliche Arbeitsaufwand deutlich höher ist. Das Erarbeiten der Theorie aus der Analysephase und die Implementierung der neuen Erkenntnisse sind sehr zeitintensiv.

### 5.1 Log-Buch: Januar – Februar (Vorbereitung & erste Fahrtests)

#### 12.01. – Raspberry-Pi Setup

- Ubuntu 22.04 auf Raspberry Pi 4 eingerichtet, remote via SSH/VNC lauffähig.

#### 14.01. – Antrieb verstehen

- Grundlagen zu Motor-/Servotreibern aufgearbeitet. Klarheit: separate Treiber notwendig, GPIO allein reicht nicht.

#### 16.01 – 18.01. – I<sup>2</sup>C & Geradeauslauf

- Motorcontroller auf I<sup>2</sup>C 0x34 verifiziert → erste Fahrtests.
- Rechtsdrift kompensiert: PWM-Mikroausgleich für Motor 4 (10/11 im 10-ms-Takt).
- Klassen-Skelett gestartet: RobotSystem, NavigationSystem, MotorController, Motor.

#### 10.02. – 24.02. – Tooling & Fahrverhalten

- Remote-Entwicklung stabilisiert (CLion DIE) → Projekt neu laden & Cache löschen
- Drift bei Mecanum bestätigt → Basis für spätere LiDAR-Zentrierung gelegt.

### 5.2 Log-Buch: Mai – Juni (Hardware/Code festigen)

#### 05.05. – 15.05. – Fokus Hardwareteile & Tests

- LiDAR kurz evaluiert (Protokoll komplex → später sauber parsen).
- Remote-Dev fix: stabiler Workflow für Build & Run. (CLion DIE)

#### 05.06. – 20.06. – Umgebung & Dokumentation

- VNC-Tuning (leichtes Desktop-Setup, Fake-HDMI) für flüssiges Remote-Arbeiten. (nicht praktisch)

- Motor-Mapping dokumentiert (w/a/s/d ↔ Rad 1–4), Grundlage für spätere Korrekturen.

### 5.3 Log-Buch: Juli (Kalibrierung & LiDAR-Wandfolge)

#### 23.07. – 25.07. – Armsteuerung funktionsfähig

- 6-DOF-Arm in Python: open, close, move\_pose, home.
- Kalibrierung: Gripper 200–700 PU (Objekt ~590 PU), Wrist auf 500 PU, Base-Referenzwinkel.

#### 27.07. – 30.07. – LiDAR Parser & erste Wandfolge

- MS200 seriell geparsst (Winkel/Distanz/Intensität).
- Wall-Follow links: Yaw-Korrektur, Ramping; Vorzeichenfix für Strafing.

#### 31.07. – Kamerawechsel vorbereitet

- Defekt erkannt → Umstieg auf USB-UVC (Waveshare IMX335) geplant.

### 5.4 Log-Buch: August (Lagermodellbau & erste Modi)

#### 09.08. – 13.08. – Lagermodell aufgebaut

- Zuschnitt, Feinschliff, Montage. Klettbänder für modulare Gänge.

#### 13.08. – Fahrlogik verfeinert von der ersten Wandfolge

- Rotation an Abzweig erst nach ~25 cm. Strafen (Driften) im Hauptgang deaktiviert.

#### 17.08. – 25.08. – Betriebsmodi & Kennwerte

- Modi: --getobjecttop, --remote (Tastatur), --follow\_route.  
([youtube.com/@danielrmv](https://youtube.com/@danielrmv))
- Test-Defaults: left\_target=180 mm, front\_stop=120 mm, forward=60 mm/s.
- Remote-Ruckler beseitigt (Start/Release-Bridging), Buzzer stabil.
- Ergebnis: Seitenabstand ~176–179 mm, Front-Stop ~118 mm (nahe Soll).

#### 28.08. – Zentrale Konfiguration

- robot\_config.json als Single-Source-of-Truth eingeführt. CLI-Flags nur temporär.  
([gitlab.com/danielrmv/Loadlifterdroid/-/tree/main/src/utils/robot\\_config.json](https://gitlab.com/danielrmv/Loadlifterdroid/-/tree/main/src/utils/robot_config.json))

## 5.5 Log-Buch: September (Kamera läuft, Streaming über Flask)

### 14.09. – OpenCV-Stream & Flask-MJPEG

- USB-UVC Kamera (Waveshare IMX335) mit OpenCV (`cv2.VideoCapture`) erfolgreich geöffnet.
- Flask-basierten MJPEG-Stream implementiert, damit der Live-Feed im Browser sichtbar ist (praktisch bei SSH-Workflows).
- Capture-Auflösung (z. B. 2592×1944) von der Stream-Auflösung getrennt (z. B. 960 px Breite) → deutlich flüssigeren Stream (Raspberry-Pi-Verhältnisse).
- Fehlerbehandlung & Cleanup: sauberes `cap.release()`. Stream bleibt stabil, CPU-Last ok.

## 5.6 Log-Buch: Oktober (Machine-Learning: Training & Integration)

### 01.10 – 02.10. – Datensatz aufnehmen & Labeln

- Ca. 700 Bilder (600 davon mit Objekten) aufnehmen auf dem Raspberry Pi mit dem `capture_data.py` Skript.
- Ca. 600 Bilder labeln.

### 03.10 – 05.10. – Datensatz organisieren, YOLOv8n trainieren & validieren

- YAML mit Klassen/Pfaden erstellt. YOLOv8n Training auf MacBook Pro (M3, 24 GB).
- Lernkurven super. Artefakte unter `runs/detect/....`

(Einsicht in den Ordner `~/data` nur auf Anfrage: [daniel.wuermli@icloud.com](mailto:daniel.wuermli@icloud.com))

### 06.10. – Modell-Export & Test auf dem Raspberry Pi

- `best.pt` auf den Raspberry Pi kopiert und mit laufendem Kamerastream getestet.
- Ergebnis: Inferenz ruckelig, hohe CPU-Last, spürbare Latenz im Stream (Pi am Limit).

### 07.10. – Off-board Inferenz (Mac) & Overlay-Pipeline

- `client_overlay.py` so erstellt, dass er Frames aus dem Pi-Stream (OpenCV/MJPEG) zieht.
- MacBook Pro lädt `best.pt` (YOLOv8n) und führt Inferenz dort aus.
- Detections (Klasse, Confidence, Bounding Box) werden zurück an den Pi geschickt (HTTP).
- Overlay wird auf dem Pi in den bestehenden Flask-Stream gerendert.
- Ergebnis: Stream deutlich flüssiger.

## 6 Diskussion

Dieses Kapitel hält den Endzustand fest: Was wurde aus der Analyse umgesetzt, was getestet und verifiziert, was funktioniert (stabil), was funktioniert noch nicht oder hat Grenzen. Ausserdem liste ich Known-Bugs und Genauigkeiten/Fehlerquellen auf.

### 6.1 Zielerreichung gegenüber der Analyse

#### Must-Haves (alle erreicht)

1. Grundlegende Navigation – vorwärts/rückwärts/drehen implementiert. Wandfolge/Front-Stop stabil.
2. Autonom auf vordefiniertem Pfad – Follow-Route/Follow-Wall Modi vorhanden.
3. Objektbeschaffung (Greifen/Ablegen) – Arm kalibriert. GetObject Modi vorhanden.
4. Grundlegende Sensorik – 2D-LiDAR MS200 für Wand Erkennung.
5. Hardware-Integration – Pi ↔ Motoren/Servos/LiDAR/Kamera lauffähig.
6. Programmierung in Hochsprache – Python, strukturierte Module, kommentiert.
7. Dokumentation – Design + Implementation (Logbuch) + Diagramme vorhanden.

#### Nice-to-Haves

1. Kameraintegration – erreicht (OpenCV, Flask-Stream).
2. Objekterkennung (CV/ML) – teilweise erreicht: YOLOv8n erkennt die Zielobjekte zuverlässig. Suche & automatische Beschaffung während der Fahrt aus Zeit- und Knowhow-Gründen nicht umgesetzt.
3. Benutzeroberfläche – erreicht: zwei Web-UIs (LiDAR-Daten, Kamera-Livestream).
4. Netzwerkkommunikation – erreicht: SSH für Start/Stopp/Deployment; Code-Sync vom Mac (Resync-Workflow).
5. Energieverwaltung – nicht erreicht: Spannungsabfrage über I<sup>2</sup>C nicht verfügbar.
6. Modulares Design – erreicht: Sensoren/Module tauschbar, mechanisch modular. Akku per Klett.
7. Sicherheitsfunktionen – teilweise: Software-Stopp via SSH/CTRL-C vorhanden; kein dedizierter Not-Aus/Sicherheitskreis.

## 6.2 Verifikation: Use-Cases

- UC-01 Pfad fahren (autonom) – mehrfach getestet, reproduzierbar bestanden.
- UC-02 Greifen & rechts ablegen – Sequenz stabil, Posen kalibriert.
- UC-03 Zielobjekt erkennen (ML) – läuft off-board (Mac) mit Overlay zurück auf Pi.
- UC-04 Erkennen während Fahrt, greifen, zurück – nicht umgesetzt (Zeit & Knowhow fehlen).
- UC-05 Live-Überwachung – Browser-Stream (MJPEG) + Browser-Stream LiDAR-Daten + Status/Logs ok.

## 6.3 Grenzen, offene Punkte, Known-Bugs

- UC-04 nicht implementiert (Zeit & Knowhow fehlen).
- Energie/Spannung: keine Messung verfügbar → kein Spannungswert.
- Kein Hardware-E-Stop (nur Software-Stop via SSH).
- Mechanik: Rad ist immer wieder abgefallen → jetzt geklebt, alles funktioniert.
- On-Pi-Inferenz: ruckelig, deshalb bewusst Off-board gelöst.

## 6.4 Genauigkeit (praktisch beobachtet)

- Seitenabstand zur Wand:  $\pm 4$  mm um 180 mm (Einzelstrahl-Modus).
- Front-Stop:  $\pm 2$  mm um 120 mm.

Bei Aufgaben, wo es auf den Millimeter darauf ankommt (z. B. Objekt holen auf einem vordefiniertem Pfad), kann man durch extrem langsames Fahren die Genauigkeit erhöhen.

## 7 Inbetriebnahme

Grundvoraussetzung sind Grundkenntnisse in Linux. Ausserdem werden hier aus Sicherheitsgründen keine Passwörter genannt.

### 7.1 Hardware

- Rechner: Raspberry Pi 4 (4 GB), 12-V Akku, microSD ( $\geq$ 32 GB).
- Chassis: Mecanum-Antrieb mit Hiwonder-Treiberplatine.
- Arm: Hiwonder Greifarm mit Bus-Servos.
- Sensorik: Orbbec Oradar MS200 (2D-LiDAR) per USB-UART, USB-UVC-Kamera (Waveshare IMX335).
- Verkabelungen:
  - 2D-LiDAR → USB am Pi.
  - Kamera → USB am Pi.
  - Treiberplatinen/Servos → gemäss Anleitung am ArmPi-Board.
  - Akku sicher befestigt (Klett).

### 7.2 Software-Umgebung

- Raspberry Pi OS: Ubuntu 22.04 LTS (64-bit).
- SSH: aktiv (`ssh username@<pi-ip>`).

### 7.3 Installation auf dem Raspberry Pi und PC/Laptop

Die detaillierte Installationsanleitung befindet sich auf GitLab, da sie dort übersichtlicher dargestellt ist und die einzelnen Schritte mit Befehlen ergänzt werden. Dadurch wird die Einrichtung deutlich vereinfacht und klar strukturiert, ausserdem können die Befehle direkt kopiert und im Terminal eingefügt werden.

([gitlab.com/danielrmv/Loadlifterdroid/-/blob/main/README.md](https://gitlab.com/danielrmv/Loadlifterdroid/-/blob/main/README.md))

## 8 Persönliche Erkenntnisse

Die Arbeit an Loadlifter war für mich eine extrem intensive, aber auch unglaublich lehrreiche Erfahrung. Ich habe in dieser Zeit mehr über Programmierung, Hardware und Systemdenken gelernt als in jedem anderen Schulprojekt zuvor. Besonders spannend war die Kombination aus den vielen Teilbereichen der Informatik, die jedoch auch ihren Preis hatte: Die anfängliche Aufwandsschätzung von 120 Stunden erwies sich als zu optimistisch, am Ende investierte ich deutlich mehr Zeit. Am meisten gelernt habe ich im Bereich Python. Von der Syntax über Klassenstrukturen bis hin zu komplexen Bibliotheken wie OpenCV, Flask, Ultralytics und smbus2. Gleichzeitig habe ich ein tiefes Verständnis für Robotik-Algorithmen, Modularisierung von Code und Linux-basierte Systeme gewonnen. Auch der Umgang mit Git und Versionskontrolle ist für mich inzwischen selbstverständlich geworden, da ich es fast wöchentlich mit meinem Repository gearbeitet habe.

Was ich das nächste Mal anders machen würde:

- Konsequenter dranbleiben. Dadurch hätte ich am Ende mehr Zeit gehabt, um zusätzliche Features umzusetzen.
- Die Datei `robot_config.json` hätte ich direkt zu Beginn des Projekts implementieren sollen. Sie hat sich als enorm nützlich erwiesen, um grosse und modulare Systeme flexibel zu konfigurieren.
- Ich würde von Anfang an auf einen 3D-LiDAR oder eine Kamera mit Tiefensensorik setzen. Damit wäre die Kombination aus autonomem Fahren, Objektsuche und präzisem Greifen (Nice-to-Haves) deutlich einfacher umsetzbar gewesen, da man die Objektposition durch ML direkt räumlich bestimmen könnte.

Die grösste Herausforderung war, alles gleichzeitig im Blick zu behalten. Oft funktionierte ein Teil perfekt, während ein anderer unerwartet ausstieg, das machte das Testen manchmal mühsam, aber auch spannend. Ich habe gelernt, strukturiert zu debuggen, Logs konsequent zu nutzen und Probleme schrittweise einzugrenzen.

Meine grössten Highlights waren die Momente, in denen nach geduldigem Tüfteln endlich etwas funktionierte, genauso, wie ich es mir vorgestellt hatte. Besonders erinnere ich mich an die Abende und Nächte, in denen ich stundenlang im Keller an meinem Laptop sass und am Roboter gearbeitet habe, bis endlich die Bewegungen oder das Greifen perfekt liefen. Dieses Gefühl, wenn etwas nach vielen Fehlversuchen plötzlich funktioniert, war unbeschreiblich motivierend und ein willkommener Glücksmoment.

Insgesamt war das Projekt für mich eine persönliche Weiterentwicklung. Ich habe in den Bereichen Machine Learning, Robotik und Elektronik enorm viel dazu gelernt. Ausserdem habe

ich viel in Dokumentation, Projektarbeit, im Zusammenspiel von Hardware und Software, im Projektcontrolling und im langfristigen Arbeiten über mehrere Monate gelernt, ebenso im Umgang mit Rückschlägen und darin, mich selbst immer wieder zu motivieren, auch wenn etwas nicht sofort funktioniert hat.

An dieser Stelle möchte ich mich noch herzlich bei allen bedanken, die mich während der Entstehung meiner Maturaarbeit unterstützt haben.

Ein besonderer Dank gilt Herrn Sven Nüesch, meinem betreuenden Lehrer, für seine aussergewöhnliche Unterstützung und Geduld. Er hat mir nicht nur in der Schule, sondern auch ausserhalb viel Zeit gewidmet und mir bei komplexen Hardware- und Softwareproblemen geholfen. Durch seine fachliche Erfahrung, seine gute Erreichbarkeit sowie seine zeitnahen Rückmeldungen konnte ich technische Hürden überwinden und das Projekt erfolgreich umsetzen. Ein ebenso grosser Dank gilt meinen Eltern, die mich während der gesamten Projektphase tatkräftig unterstützt und immer wieder moralisch ermutigt haben. Ohne ihre Hilfe und Geduld wäre die Umsetzung in dieser Form nicht möglich gewesen.

Allen, die mich auf meinem Weg begleitet, ermutigt und unterstützt haben, gilt mein aufrichtiger Dank.

## Tabellen- und Abbildungsverzeichnis

### Tabellen

- Tab. 1:** Nutzwertanalyse, S. 35
- Tab. 2:** Nutzwertanalyse, S. 36
- Tab. 3:** Nutzwertanalyse, S. 37
- Tab. 4:** Zeitplanung mit Aufwand, S. 48

### Abbildungen

- Abb. 1:** Autonomie eines Roboters, S. 3
- Abb. 2:** Grundprinzip eines Sensors, S. 4
- Abb. 3:** Funktionsprinzip eines Ultraschallsensors, S. 7
- Abb. 4:** Grundprinzip eines Aktuators, S. 8
- Abb. 5:** Lagerplan, S. 28
- Abb. 6:** Ansicht mit Längen und Breiten, S. 28
- Abb. 7:** Ansicht mit Höhen, S. 29
- Abb. 8:** Use-Case-Diagramm, S. 31
- Abb. 9:** Systemübersicht, S. 49
- Abb. 10:** Control-Ebene, S. 50
- Abb. 11:** High-Level-Ebene, S. 51
- Abb. 12:** Low-Level-Ebene und I/O-Abstraktion S. 52
- Abb. 13:** Peripherie-Subsysteme S. 53
- Abb. 14:** FollowRoute Sequenz, S. 54
- Abb. 15:** DefinedRouteGetObjectTop Sequenz, S. 55

## Quellenverzeichnis

- OpenAI. Titelbild der Maturaarbeit. Generiert mit ChatGPT. Abgerufen am 02.08.2025
- Oubbati, M. (2007). Robotik. Uni Ulm. Abgerufen am 26.08.2025 von [https://www.uni-ulm.de/fileadmin/website\\_uni\\_ulm/iui.inst.130/Arbeitsgruppen/Robotics/Robotik-Skript\\_07-08.pdf](https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.130/Arbeitsgruppen/Robotics/Robotik-Skript_07-08.pdf)
- Wasser, L. A. (2024, 13. September). The Basics of LiDAR - Light Detection and Ranging - Remote Sensing. National Ecological Observatory Network (NEON). Abgerufen am 28.08.2025 von <http://fhgr.ch/studium/bachelorangebot/medien-technik-und-it/mobile-robotics/>
- Flyguys. (2020, 5. Dezember). Advantages and disadvantages of LiDAR technology. Abgerufen am 28.08.2025 von <https://flyguys.com/advantages-disadvantages-lidar-technology/>
- Jouav. (2025, 6. Februar). LiDAR vs Photogrammetry: The Ultimate Showdown for 3D Mapping. Abgerufen am 30.08.2025 von [https://www.jouav.com/blog/lidar-vs-photogrammetry.html?utm\\_source=chatgpt.com#jouav-scrollspy-anchor-5](https://www.jouav.com/blog/lidar-vs-photogrammetry.html?utm_source=chatgpt.com#jouav-scrollspy-anchor-5)
- DJI-Enterprise. (2022, 29. März). Ground Control Points. Abgerufen am 30.08.2025 von <https://enterprise-insights.dji.com/blog/ground-control-points>
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. Abgerufen am 30.08.2025 von <https://www.nature.com/articles/nature14539>
- Wei, Y. (2024). Applications Of Ultrasonic Sensors: A Review. Abgerufen am 04.09.2025 von [https://www.researchgate.net/publication/386118233\\_Applications\\_of\\_Ultrasonic\\_Sensors\\_A\\_Review](https://www.researchgate.net/publication/386118233_Applications_of_Ultrasonic_Sensors_A_Review)
- Inf-Schule. (2024, 5. Dezember). Ultraschallsensoren am Auto Abgerufen am 09.10.2025 von [https://inf-schule.de/informatiksysteme/calliope/projekte/Ultraschallsensor\\_Auto/lernstrecke/Auto](https://inf-schule.de/informatiksysteme/calliope/projekte/Ultraschallsensor_Auto/lernstrecke/Auto)
- Qiu, Z., Lu, Y. & Qiu, Z. (2022). Review Of Ultrasonic Ranging Methods And Their Current Challenges. *Micromachines*, 13(4), 520. Abgerufen am 05.09.2025 von <https://pmc.ncbi.nlm.nih.gov/articles/PMC9025471/#sec3-micromachines-13-00520>
- Siciliano, B., & Khatib, O. (Hrsg.). (2016). Springer Handbook of Robotics (2. Aufl.). Kapitel: Mechanisms and Actuation. Abgerufen am 06.09.2025 von <https://link.springer.com/book/10.1007/978-3-540-30301-5>

- Anaheim Automation. (2024, Juli). Stepper Motors versus Servo Motors. Abgerufen am 06.09.2025 von <https://anaheimautomation.com/blog/post/stepper-motors-versus-servo-motors>
- Omron Corporation. (o. J.) Technical Explanation for Servomotors and Servo Drives. Abgerufen am 06.09.2025 von [https://www.ia.omron.com/data\\_pdf/guide/14/servo\\_tg\\_e\\_1\\_1.pdf](https://www.ia.omron.com/data_pdf/guide/14/servo_tg_e_1_1.pdf)
- Adafruit. (2025, 22. Januar). Using Servos with CircuitPython – Low Level Servo Control. Abgerufen am 06.09.2025 von <https://learn.adafruit.com/using-servos-with-circuitpython/low-level-servo-control>
- Plauska, I., Liutkevičius, A. & Janavičiūtė, A. (2023). Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics*, 12(1), 143. Abgerufen am 07.09.2025 von <https://www.mdpi.com/2079-9292/12/1/143>
- OpenCV Documentation. (o. J.). OpenCV modules. Abgerufen am 08.09.2025 von <https://docs.opencv.org/4.x/index.html>
- Ultralytics Docs (o. J.). Ultralytics YOLO Docs. Abgerufen am 12.09.2025 von <https://docs.ultralytics.com>
- Python Docs. (o. J.) Python Standard Library. Abgerufen am 15.09.2025 von <https://docs.python.org/3/library/index.html>
- Flask Documentation. (2010). User's Guide. Abgerufen am 15.09.2025 von <https://flask.palletsprojects.com/en/stable/>
- PyPI. (o. J.). smbus2 – A drop-in replacement for smbus-cffi/smbus-python in pure Python. Abgerufen am 17.09.2025 von <https://pypi.org/project/smbus2>
- smbus2 Documentation. (o. J.). smbus2 – A drop-in replacement for smbus-cffi/smbus-python. Abgerufen am 17.09.2025 von <https://smbus2.readthedocs.io/en/latest/>
- Linux Kernel Documentation. (o. J.). I2C : dev-interface. Abgerufen am 17.09.2025 von <https://www.kernel.org/doc/Documentation/i2c/dev-interface>
- Iñigo-Blasco, P., Diaz-del-Rio, F., Romero-Ternero, M. C., Cagigas-Muñiz, D., & Vicente-Diaz, S. (2012). Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, 60(6), 803–821. Abgerufen am 18.09.2025 von <https://www.sciencedirect.com/science/article/abs/pii/S0921889012000322> (Inhalt siehe Anhang)

## Hilfsmittelverzeichnis

ChatGPT (OpenAI), Version GPT-5, genutzt über ChatGPT-App (Desktop & Mobile)

Codex (OpenAI). Modell code-davinci-002, genutzt über die Codex CLI im Terminal (macOS)

Grammarly (Grammarly Inc.), genutzt über die Web-App (Browser).

Aktiv-online. (2024). Intralogistik: Das ändert sich durch Automatisierung | aktiv online [Video]. YouTube. <https://www.youtube.com/watch?v=LdcWh7wpM9M>

element14 presents. (2021). How Do Ultrasonic Distance Sensors Work? [Video]. YouTube. <https://www.youtube.com/watch?v=2ojWO1QNprw>

System Design School. (2024). Multithreading vs Multiprocessing | System Design [Video]. YouTube. <https://www.youtube.com/watch?v=PgDaJEjlBul>

Edje Electronics. (2025). How to Train YOLO Object Detection Models in Google Colab (YOLO11, YOLOv8, YOLOv5) [Video]. YouTube. <https://www.youtube.com/watch?v=r0RspiLG260>

## Selbstständigkeitserklärung

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die erlaubten und angegebenen Quellen und Hilfsmittel verwendet.

Name	Vorname
Würmli	Daniel
Ort	Datum
Frauenfeld	15.20.2025

Unterschrift



## Anhang

### Code

Die vollständigen Projektdateien, Quellcodes, Test-Snippets sowie ergänzendes Material zur Maturaarbeit «Loadlifter – Ein autonomer Lagerroboter» sind online abrufbar. Alle Repositories sind öffentlich zugänglich.

Die Verzeichnisse enthalten die aktuelle Software, Konfigurationsdateien, ein trainiertes neuronales Netzwerkmodell sowie begleitende Entwicklungsnotizen und Diagramme. Aus Datenschutzgründen sind die Trainingsdaten und weitere private Informationen nicht öffentlich. Anfragen bitte an [daniel.wuermli@icloud.com](mailto:daniel.wuermli@icloud.com).

### Projektverzeichnisse:

- GitHub: <https://github.com/daniel-rmv/Loadlifter>
- GitLab: <https://gitlab.com/danielrmv/Loadlifterdroid>

### Quelle

Auszüge aus Iñigo-Blasco et al. (2012) – Robotics software frameworks for multi-agent robotic systems development.

Enthaltene Abschnitte: 3.2, 4.2, 5.3, 5.7:

CARMEN [77], OPROS [78], CLARATY [79], MARIE [80], Mobile Robots [81], MRPT [82], MSRS [83], Peis-Ecology [84], Pyro [85], Webots [86]. Moreover, while some non-open-source robot software platforms have appeared, such as Microsoft Robotics Studio, the focus of this study is to be on open source systems. This is because two reasons: first, the diversity of non-commercial platforms is very wide, which permits to compare and discuss lots of implementations, analyzing their drawbacks and advantages. Secondly, internal architecture of non-open-source platforms is (due to commercial reasons) less documented.

### 3.2. Aspects focused on by existing RSFs

Despite the varied foci, many common points are shared; in general they are all geared toward contributing solutions for typical problems in robotics. These RSFs are normally centered on one or several of the following areas:

- **Middleware for distributed robotics.** Some RSFs provide a set of tools that enable the architecture of the robotic system to be organized by taking advantage of characteristics that are intrinsically parallel to the robotic system. At runtime, the architecture is organized into nodes which communicate with each other by means of passing messages. This design of robotic systems may be more complicated than monolithic approximations, but the architecture yielded is a lot more flexible thanks to the high degree of uncoupling of the nodes. All the presented applications in Section 2.2 are highly sensitive to the use of properly distributed middleware during the software design and development stages. The use of this kind of middleware is extremely recommended, almost mandatory (as mentioned in Section 2.3). The middleware for distributed robotics tools promotes scalability, reusability, and the high tolerance of errors and also renders parallel development and integration tasks easier. Examples include: ROS, VSLAM, OpenRDK, OpenRTM.
- **Introspection and management tools.** The introspection and management tools permit the monitoring, visualization and analysis of the state of the robotic system to be carried out during execution which enables the tasks of tracking and/or error detection and correction. Tools worth mentioning here include those of logging, management, and system-state monitoring from the graphic interface, the console, or the web. They are especially important in distributed robotics architectures given the difficulty of debugging and testing of such architectures.
- **Advanced development tools.** These enable time and expense to be saved in the implementation and testing stages, are especially important as soon as a complex robotic system development tools include compilers, dependency managers, with other modules or system libraries, and Integrated Development Environments (IDEs). Deployment tools permit scripts and configuration files to be designed which define the startup of the robotic system, the initial values of the parameters, the nodes used, its localization name, and connections among other details, are specified in these files.
- **Robot-hardware interfaces and drivers.** These encapsulate the code of the controller of a specific device (sensor or actuator) behind a well-known and stable programming interface. This interface has to be defined between robot devices and user modules. The objective is to attain reusability of the devices and of the high-level algorithms that they use. It is common for a specific device to offer various functions and particular configuration parameters. When this occurs, the device controller can implement various interfaces according to its characteristics. For example, if a commercial robotic arm offers a system of integrated vision in the end-effector, then the

controller can implement the generic interface of the robotic arm and of the vision. The specific parameters of the device are stated in a configuration file, and thereby the algorithms remain free from coupling with specific devices, and reusability is maintained. In this way, the Open-Close principle of Software Engineering is observed: code is open to extension and closed to modification. The frameworks specialized in this area offer programmers a simple development mechanism for their own drivers. Furthermore, they commonly provide a set of built-in controllers for diverse commercial devices. Player, MOOS, ORCA, and CARMEN are examples of Frameworks that are centered in this area. This aspect will be taken into account in the RSFs surveyed in Sections 4 and 5.

- **Robotics algorithms.** These are often the objective of an RSF: to provide generic and reusable algorithms and functionalities in the field of robotics. They correspond to the functionality and application levels shown in Fig. 2. Those algorithms are encountered at different levels of abstraction: from a low level, such as those related to kinematics, control, robot perception, Bayesian estimation up to others of a high level such as planning, human interaction, robot learning, navigation algorithms, motion planning, Bayesian Filtering, and SLAM. These abstract algorithms are usually built over the Robotic Hardware Interfaces or other low-level robotics algorithms. This software is usually provided in the form of libraries or components that can be instantiated as nodes of the system. Some RSFs, such as Player, do not show a clear line between these algorithms and device drivers since both are usually wrapped behind a stable and known programming interface to promote reutilization.
- **Simulation and modeling.** These permit modeling, prototyping, and simulation of the final system to be generated, thereby saving both time and expense. They also serve as an early test of viability of the solutions, which may prevent situations of mutual function conflict, etc. These tools stand out for their capacity to express mathematical concepts and for the possibility of carrying out simulations on these models, thereby obtaining results which can be interpreted for an improvement in the system design. These tools usually offer the possibility of generating scenarios and environments that can be used to simulate virtual worlds with rigid solid dynamics. Some free tools which deserve a mention include: OpenRAVE, Stage, UsarSim, Gazebo, and Breve.

### 3.3. Existing RSFs overview

The earlier technologies tackled a design of a Robotics Framework following an abstraction-level-based architecture (see Fig. 2). However a proper MARS software architecture should have a strong infrastructure focused on distributed systems where nodes communicate with each other independently of the abstraction level of the task that they perform. To this end, in this survey only those frameworks that fulfil the conditions stated under the heading “Middleware for distributed robotics” of Section 3.2 are to be analyzed.

Table 1 shows a set of existing RSFs and their strong characteristics. Those RSFs selected for a more in-depth analysis are shown in bold font. Some of the discarded frameworks may be excellent, popular and active RSFs. However, they fail to represent the best choice for a complex MARS since they provide an insufficiently powerful distributed middleware mechanism. Other aspects significant in the criteria for MARS, such as introspection tools and development tools, are also taken into consideration. In addition, other selection criteria considered in this study include: characterization with an Open-Source and free-commercial license, proper documentation for evaluation, a



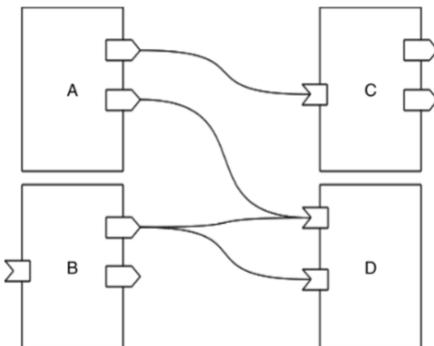


Fig. 3. Port mechanism.

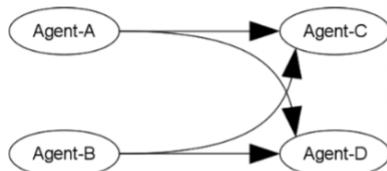


Fig. 5. Topic mechanism implemented over a P2P network protocol (like TCP).

Generally, the RSFs display naming-service mechanisms (also known as White Paper Service). Some RSFs have characteristics of a more advanced nature such as "Name Pushing" and the relative addressing of agents or resources of the MAS. These mechanisms are crucial for the prevention of name conflicts when nodes are instantiated in different spheres of the system. In this case although they can both have the same code, they are in different contexts, which is why a relative-name references different agents or resources.

These complementary mechanisms related to the naming service boost the flexibility of the resources that can be referenced.

- **Renaming (Remapping).** During deployment, this mechanism allows all the references that exist in the logic of a node of the system (resources, nodes, topics, services, etc.) to be substituted by others. It is highly useful in obtaining the names integration modules. For example, if two modules implemented by different development teams were designed to use the same set of topics, an Agent-A could receive a message from one team, while another node would manage its communication. This problem arises in complex hybrid P2P architectures involved. There are two solutions to this problem: change the implementation and change the name of the affected topics, or use the renaming mechanism, which allows the final value of the deployment configuration to be kept in a file which will take the references to the system resources. This last solution is "A deployment-oriented solution".
- **Relative and absolute naming (Namespaces).** This is the capacity to represent hierarchies in the names. It allows the code of a node to be referenced by other resources in an absolute manner (namespace + name) or a relative manner (name). It is an important tool for creating an organized design and clean code. It is mainly useful when it is used in conjunction with Name Pushing.

- **Name pushing.** It is a particular form of naming. It is a mechanism that allows a group of resources (nodes, topics, etc.) to be instantiated in a specific namespace at the moment of deployment. This mechanism allows conflicts in the resource names to be prevented. When name pushing is carried out the interconnections of the nodes can change, since all the references to resources with relative names will only be sought in the present namespace, whereas the absolute references to resources will remain unaffected by name pushing and will continue to be referenced to the same resources. Fig. 6 shows how two groups of resources (Actuator-Agent, Intelligent-Agent and control topic) are instantiated twice and located in two different namespaces. It also shows how the absolute references to the monitor topic and Monitor-Agent resources stay in both groups in spite of name pushing.

#### Lookup service.

This service is peculiar to Hybrid P2P architectures where a central or master node exists which contains special information about the whole system. It is also known as the Yellow Pages service, where the central agent or node acts as a directory of the existing resources. The system agents can consult this directory and look for other agents that offer certain services or that fulfill



Fig. 4. Node interaction through the topic mechanism (logic view).

- **Events.** These are one-to-many asynchronous communication mechanisms that allow a low degree of coupling to be made between resources and agents. The main difference between ports and events is that the connection concept does not exist; an event is implicitly asynchronous, and the subscribed nodes always deal with events by means of callbacks.
- **Services.** This is a communication mechanism that allows the remote execution of a procedure, the remote-procedure call (RPC). Two messages come into play: Request and Response. The message request is sent by the client node and indicates what procedure is desired to be executed and its arguments. The Response message is sent by the server node with the result of the operation. It is a typically synchronous procedure where the client remains blocked while the response is awaited. This mechanism is typically used for Robot Hardware Interfaces (see "Robot Hardware Interfaces" in Section 4.3).
- **Properties.** The nodes can show a set of properties that represent part of their status to the rest of the nodes. Each property is usually managed through a pair of services get/set (get property for the listener, and set property for the node that is going to show a property). However, several RSFs treat these two services independently. In Hybrid P2P architectures, the properties are often stored in the master node; when a case, it is called the "Parameter Server". It is worth mentioning that the pure P2P approach is more desirable. For instance, ROS uses a parameter server, while OROCOS and YARP have real distributed node properties.

#### Naming service

It is any old service typical in traditional networks and was otherwise known as the White Pages service, which allows the localization of nodes, and other global system resources such as topics, from a name. Specific node resources, such as ports, properties, services and events, seldom have a global name managed by the naming service. This service is present in all of the frameworks analyzed, but most of the discarded frameworks of Section 3 fail to provide it.

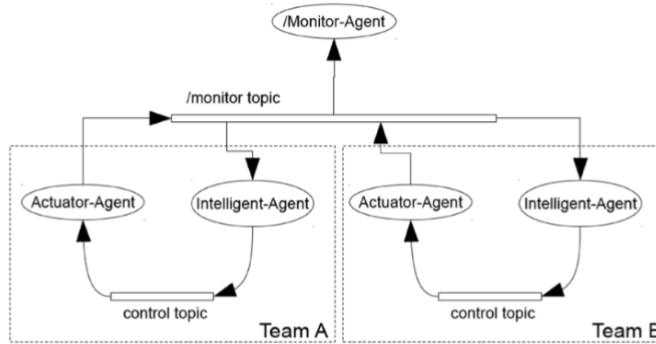


Fig. 6. Absolute and relative names for nodes and topics.

certain properties. It is worth mentioning that the lookup service for multi-agent systems is well defined in the FIPA standard [13].

Not all the RSFs implement the lookup service. This is a fundamental characteristic for the creation of a robust MAS. For example, where certain nodes cease to function or cease to offer a particular service or functionality, the system must be able to replace these nodes by searching for substitutes that offer similar services or functionality.

There are innumerable situations where this service is essential: heterogeneous robots in teams or swarms can obviously benefit from this service (see the discovery service example below). Another example is a complex autonomous robot controlled by a multi-agent control system in an internal network. Each node works on a specific task: some nodes (agents) in a dedicated TEPUs work to handle the obstacle avoidance and localization tasks, while other nodes are dedicated to handling sensors and actuators.

The localization and obstacle avoidance agents need a prediction of the future state based on the odometry system. If odometry sensors fail, the robot does not receive any information about its state, then both agents ask the lookup service whether another agent can provide the odometry service. For instance, some cameras can start working as a visual odometry system although they have other primary tasks. Although camera agents then become overloaded, the system can continue working until a new agent is assigned to the

service makes the MARS more robust. Let us consider a team of fire-fighters as a team of heterogeneous robots [1] that is composed of members with different capacities: fire hose, first-aid, tracking, etc.

When a fire is detected in a forest, the tracking robots could explore to find people in danger or injured. In this case, one tracking robot should look for the nearest and unoccupied first-aid robot and waterspout robot in order to heal and protect such people. All this negotiation and interaction between robots starts with the use of the discovery service asking for the first-aid and fire-hose agent robots.

The discovery service is a very dynamic solution when new robots must be dynamically incorporated into a team, when connectivity problems can occur or when the availability of robots is not ensured, due to robot damage, energy autonomy, etc.

#### Agent mobility

This characteristic defines the capacity by which a software agent can move between network nodes, for example, sensors and robots. Some definitions such as the MASIF standard [88] consider that mobile agents are those that can move, whereas others [17] distinguish between stationary agent and mobile agent. Nevertheless, it is usually implemented in the general purpose MASFs. On the other hand, this feature has not been implemented by any RSFs since robotics architectures are nowadays much more static than other multi-agent systems architectures.

812

P. Iñigo-Blasco et al. / Robotics and Autonomous Systems 60 (2012) 803–821

nodes and high computational resources and are therefore tasks of high energy consumption. Let us consider a "navigator agent" originally located in a mobile robot, which requires the recognition of an unfamiliar face. This navigator agent could travel temporarily to the environment computers to perform such a task there, in order to access a more complete face database stored in the building.

Therefore, if an image-processing task is computationally expensive, an agent could use the powerful computational infrastructure of the intelligent building to save both time and

real-time constraints must seriously take this aspect into account. This is the case of Multi-Agent control systems and MARS where numerous image and point cloud messages are sent over the network.

Each node communication mechanism may be implemented in a different way. For instance the topic communication mechanism may be more properly implemented with a physical bus network and using a multicast protocol. However there are many situations in which this straightforward solution is not possible. Hence, the more traditional approach of using TCP/IP and UDP protocols.



whose constraints are related with communication performance or energetic autonomy. Applications based on mobile robots which work autonomously should especially consider this aspect; mobile sensor networks, swarm robots, and modular robots usually have significant energy autonomy constraints. The use of a binary data format facilitates better performance and energy saving. Furthermore, binary formats focused on performance are custom made and have no standardization. Therefore, introspection tools have to be specifically designed for each binary data format. On the other hand, text-based formats are more adequate for platform-independent introspection tools.

#### Concurrency model

This characteristic indicates if the framework organizes the agents by means of processes or threads. Processes are more flexible and can be distributed between different physical nodes of a network, whereas threads pertain to the same process and are sometimes useful for agents with closely related tasks where the velocity of communication is critical. There are frameworks in robotics that support mixed models. Frameworks that allow the modules to be organized as multi-thread usually internally implement the necessary memory protection and concurrency mechanisms so that the programmer uses this system transparently or semi-transparently.

#### 4.3. Other aspects of robotics frameworks

The general characteristics of the RSFs are briefly described in Section 2 of the present study. In addition to the middleware characteristics of the communications, the RSFs present other characteristics such as the support of hardware abstraction layers, visualization, and presentation.

##### Development tools

Several development tools are available for the construction of software development. Commands for the construction of the system tend to be of a regular type, and can solve problems of dependences on other modules of the framework. On the other hand, some frameworks allow the graphical creation and interconnection of system modules.

##### Deployment tools

Frameworks usually offer infrastructure to facilitate deployment tasks. This infrastructure typically appears in the form of tools and configuration files. Deployment tools are crucial in maintaining the scalability of the distributed and complex robotics architectures. The architecture of the system is defined and the participating nodes are specified in the configuration files. The configuration files can also define boot and connection parameters and mechanisms of communication between the nodes. The tools can be visual or command-lines and allow the start, stop and administration of system nodes, as well as the modification of the initial parameters through configuration files, or the conducting of operations related to the naming service, such as renaming or name pushing.

##### Simulation capabilities

The simulators offer a visual representation of the problem and execution in virtual worlds of rigid solids. The frameworks studied are not focused on simulation capacities; nevertheless some frameworks have implemented modules that allow a rapid integration with simulators such as Stage, Gazebo, OpenRAVE, OcarSIM or Webots. Simulators play a major role by allowing the system to be modeled in the design stage, by finding errors in the early stages of development, and by saving time and money. They are also useful for testing ideas.

##### Robot hardware interfaces

RSFs usually support a set of hardware devices and robots. This is one of the lower-level aspects of the Robotics software Development (see Section 3.2). Most existing RSFs mentioned in

**Table 1** tackle this aspect, and commonly use one of the two following approaches in the implementation of the interface:

- **The API isolation approach:** This approach follows the layered architecture discussed in Section 2.3. An application programming interface (API) is designed in a language such as C and Java. The specific code then implements such interface, usually through inheritance. Programmers are encouraged to make high-level algorithms over these interfaces instead of specific low-level ones. This approach is often referred to as *RPC-style*.

two ways:

- **Dynamic Link Library Isolation:** The driver code is isolated in a dynamic library and loaded by the node at runtime. This is the simplest way of abstraction. It is adequate for rapid software development and higher driver performance. However problems arise when multi-language support or introspection compatibility is required. Clear examples of PCLs with this approach are MPT [22] and Poco [85].
- **RPC-API Server Isolation:** The driver code is isolated in a server that is invoked by a RPC. The API internally calls a remote server which executes the requested function by means of a Client/Server Architecture between layers (see Section 2.3). This mechanism is transparent for the client (upper layer) since the code is coupled over a programming interface. This approach is more flexible and enables the use of distributed processing, various programming languages in each layer, etc. The main drawback of this approach is that the latency of operations may increase and real-time capabilities may be lost, if the correct transport layer and operative system is not used. The most representative example of this approach is the Player/Stage RSF. This approach is typical in RSFs which use CORBA as a transport layer, given that CORBA provides itself with a distributed OOP-mechanism. The multiple language support is achieved through code generation from any Interface Definition Language, such as CORBA IDL. Examples of RSFs using this approach are MIRO and MOOS.

- **The node isolation approach:** This approach is peculiar to P2P architectures (see Section 2.3). The device driver code is isolated in a component that can be instantiated as a node at runtime. This node uses the communication mechanisms explained in Section 4.2 (ports, topics, services, etc). The programmer incorporates these mechanisms explicitly in the node code, and hence the coupling point is located in the message data structure. Programmers are therefore encouraged to use standardized message types for reusability purposes. Messages types include: motor commands, images, clouds of points, and IMU sensor readings. Again the message data type may be problematic in multi-language RSFs. As in the RPC-API isolation approach, the main drawback of this approach is that the use of distributed communication mechanisms may increase the latency of operations and therefore real-time capabilities may be lost.

The difference between the node isolation approach (using a service communication mechanism) and the RPC-API approach are sometimes unclear, but the key differences between them are:

- The RPC-API approach imposes a strict client-server layered architecture for Robot hardware interfaces. In this architecture, the server cannot take the initiative since it only provides a set of services to the upper layer. In the node isolation approach, communication is balanced and both nodes can synchronously communicate to each other.
- The programming style in the node isolation approach with services is much more explicit and the coupling point is in the message data structure, while the coupling point in RPC-API is the provided list of methods. Programming code can be more comfortable over RPC-API than over the service communication mechanism.

of California and the Michigan Technological University, and implements the OMG MASIF standard for multi-agent systems and also, albeit only partially,<sup>3</sup> the FIPA standard, which is why it is similar to JADE in many respects. The most special characteristic of Mobile-C is that allows the creation of mobile agents developed in C/C++. It is a lower level language, which allows greater control over the hardware on the systems in which it is hosted (agents).

It is important to emphasize that C/C++ is a language that, during its process of compilation, remains strongly coupled to a specific hardware architecture and operating system. Nevertheless, Mobile-C is able to implement the mobility of agents. To obtain the independence of the platform, C++ agents do not travel via the already compiled network but in form of source code accompanied with their state variables.

Mobile-C is suitable for real-time temporal requirements (real-time since the C++ interpreter is temporal) and deterministic and is also compatible with real-time operating systems such as QNX. These characteristics can make it suitable for the programming of robotic systems. Mobile-C supports various operating systems in addition to QNX, such as Windows, Linux, OSX, and other UNIX operating systems such as Solaris.

The communication between the mobile agents is implemented by means of message passing and the transport mechanism used is TCP. Mobile-C has a naming service that allows it to locate the different agents; however, it does not allow pushing mechanisms (or namespaces) or absolute and relative names. It also has a powerful lookup service, as defined by the FIPA specification, which allows it to find agents that provide services with specific characteristics.

Mobile-C lacks drivers or generic algorithms for robotics, since its sphere of applications is wider. It does not display compatibility with other RSFs, such as robotics or simulators and lacks a good set of simulation, development, deployment, and monitoring tools, which is why these tasks must be made manually or with off-the-shelf tools.

Mobile-C is used in various situations, such as: multi-sensor data fusion, multi-camera surveillance for intelligent homes, and urban traffic signal control based on MAS.

and service mechanisms. Over these services another complex communication mechanism between nodes is provided for tasks of longer duration called *actions*. All these mechanisms are built over a TCP transport layer using a custom-built binary format of messages. Nonetheless, interactions with the master node are made through XML-RPC/TCP and experimental support for UDP transport is featured. Serial transport implementation (called Rosserial) is also provided especially for use with microcontrollers such as Arduino.

The name service let a hierarchical organization of the network resources like topics, services and nodes. ROS also has the so-called “pushing feature” to organize and identify properly these resources. These resources can be referenced in a relative or absolute manner. thereby promoting reusability and avoiding problems of name conflicts. The master node provides a service lookup system to search any registered service that fulfills requirements for certain characteristics. This allows microcontrollers to belong to the P2P network. These transport capabilities are weak in comparison with other technologies, such as YARP.

ROS allows the use of multiple languages although the main language is C++, it also supports Java, Python, C, C++, MATLAB, Octave, and LabView. Regarding the interoperability between languages, the connection points are the interface messages and services. The key feature that makes ROS work with different languages is the set of tools for message code generation (msg and genmsg). These tools take an interface description language for messages (for topics) or services and generate specific language code.

The system is focused on working in Unix-based systems (Ubuntu, OS X, etc.). As usual, ROS has a library that can be used for AVR microcontrollers using the so-called “Rosserial” mechanism, and support for developing nodes using Ethernet LAN networks is provided. However, ROS is inadequate for real-time applications since neither the underlying operating system nor the main transport mechanism (TCP/IP) are real-time. Other RSFs such as QNX or RTOS are best suited for real-time execution tasks.

This RSF provides a good number of graphical tools: command lines for various purposes: development, deployment management, monitoring and debugging of the distributed system as well as 2D and 3D visualization that allows a virtual view of the system interprets the real environment.

ROS implements the „multi-process“ and the „multi-thread“ concurrent model. The multi-thread concurrent model is used when the transport mechanism becomes problematic due to intensive use of heavy data messages, such as images and clouds of points. Nodes in threads (called nodelets) provide the main advantage of making possible a zero-copy mechanism of message exchange through shared memory. However, they present two important drawbacks: nodelets are only available for the C++ language and shared memory communication has to be explicit in the nodelet code. On the other hand, YARP (which is based on C++)

### 5.3. ROS (Robot Operating System)

This is an RSF that is notable for being generalist and able to integrate with other existing technologies [67]. It arose as a spin-off from Willow Garage in collaboration with the University of Stanford. The latest release is ROS Electric. Emu (30/7/2011), and a new release ROS-Fuerte.Turtle is due in March 2012..

This RSF has a numerous, active and growing community, which probably constitutes the most important factor toward making ROS one of the most complete RSFs today. Several organizations, such as companies and universities, are using or collaborating with ROS in their research, thereby forming a pool of federated repositories not controlled by Willow Garage. It is noteworthy the quality and quantity of its documentation and its fast growth. This has been made possible thanks to the availability of a large number of graphical tools for development, deployment, monitoring and debugging of the distributed system as well as 2D and 3D visualization that allows a virtual view of the system interprets the real environment.

<sup>3</sup> http://www.ros.org/doc/api/mobilec/html/

generate the proxy code from its IDL interface, for each of the services. Furthermore, it has a graphical tool (27) that uses the Java integrated in the Eclipse development environment, which allows the schematic design of the interconnections between the different nodes and also specifies their initial configuration.

Furthermore, OpenRMF has a good number of drivers implemented for different types of robots, sensors and actuators. It is also compatible with simulators such as Stage, Gazebo and OpenHRP. OpenRMF does not implement mobile agent capacities or any agent communication standard.

### 5.7. OROCOS (Open Robot Control Software)

This RSI arose from a project promoted by the European Robotics Research Network (EURO-RO), and was implemented mainly by the KU Leuven in Belgium. OROCOS [66] offers a very specialized framework for the development of industrial robotic systems. The framework focuses on the following aspects: Robot hardware interfaces and drivers of components for real-time applications. To this end, this RSI allows the use of various operating systems, including several with real-time capacities [59] (RTAI, Xenomai), and there are also positive experiences with other systems such as QNX and VxWorks. It also allows generic OSs such as Linux, OSX, Windows, and Windows CE. This framework provides the infrastructure and communication facilities needed to implement distributed real-time systems, and other languages such as Python and Simulink. OROCOS has a series of standard modules for robotics, including the communication and interface between real and virtual machines of each agent.

OROCOS proposes the use of hybrid P2P architecture of nodes that can communicate by means of port, service, event and property mechanisms. The distributed system is based on CORBA as a transport layer. CORBA components are typically presented by using various implementations of CORBA ACE or TAO, which presume real-time capabilities. Other transport mechanisms are supported such as EtherCAT and CANopen with which real-time communications can be achieved.

OROCOS can use either the RPC API (socket or the node isolation mechanism) as robot hardware interface. The communication mechanisms between nodes are ports, events and services. It is worth mentioning that even with this kind of robot hardware interface, OROCOS can achieve real-time capabilities when the user creates his own tasks (tasks) and drivers (drivers such as EtherCAT and CANopen) are used. Using the naming service of CORBA and also provides its own lookup system. It may be the most industrially-oriented RSI studied in this survey and makes emphasis in real-time which is one of the most forgotten aspects in the existing RSIs.

This RSI also provides a good set of robotics algorithms and libraries which focusing on aspects such as kinematics, motion control, and Bayesian filtering. Many drivers are also provided for robots of KUKA and some perception sensors such as cameras and laser rangefinders. It is partially integrated with ROS and YARP.

OROCOS have been applied intensively in real-time multi-agent control systems for manipulator robots and has also been used as the software infrastructure in the Urban Challenge for the DARPA Urban Challenge.

In the context of MAS development, neither agent mobility nor agent communication standard is implemented.

### 5.8. ORCA (Open Robot Control Software)

This is a mature RSI developed by the Kungliga Tekniska Högskolan and currently mainly maintained by the Australian Centre for Field Robotics. It was originally a part of the OROCOS project,

although these two branches never managed to merge. It allows processes that communicate by means of services. These services implement the functionality defined in a standard interface well known for promoting reuse. ORCA uses the ICE technology and has no mechanism of lookup or discovery service.

The supported operating systems are Linux, Windows, OS other UNIX systems. It also supports QNX; nevertheless ORCA not offer its own mechanisms to confront problems with real time restrictions.

ORCA uses ICE technology in its transparent layer, which uses the binary serialization mechanism that this technology provides. It also allows it to make TCP, UDP and SSL communications. The main development language is C++, although it has experimental support for other languages such as Java, Python, C#.

ORCA provides visual and command-line development which allow a convenient definition of the architecture. Offers a great number of off-the-shelf components for the extended devices such as laser range-finders, cameras, etc. It supports the real-time events and uses its own agent communication standard.

### 5.9. Comparative study

The comparative study is at this point summarized in three tables. For each general aspect related to the networked agents, a table of results is shown, where rows represent the analyzed RSI and columns the analyzed aspects. Taking into account all these aspects and the major significant differences between the frameworks analyzed, an interpretation of this comparison in the fields of RSI and MAS is carried out in the following section.

**Table 2** reviews the general aspects discussed in Section 4.1 the proposed frameworks.

In accordance with the individual analysis, a comparative summary of those aspects related to the communications used RSI is presented in Tables 3 and 4. Note that implementation of the various RSI and the MAS have led to significant differences and similarities between these aspects.

The last group of significant framework aspects discussed previously is summarized in Table 5 for the selected RSI and MAS.

As verified in this survey, there is a variety of technologies suitable for MAS development. For each framework studied certain aspects can be highlighted. MobileC allows the mobility of agents programmed in C/C++. JADE is noted for its compliance with the FIPA standard and its acceptance by the research and development community, in addition to having the most powerful lookup service. YARP offers very flexible and optimized transport methods and shows the best distributed aspects, such as availability of a pure P2P Architecture and a very powerful message-passing mechanism (topic-port). ROS stands out for its transverse and integrating approach to the various RSI's, provides the wider set of robotics software packages for drivers and robotics algorithms. It offers great tools, possesses widespread popularity, documentation and, hence, easier to learn and use. OpenRMF is a real-time system for managing the complex process. Agent-based architecture is the common characteristic implemented in other RSI. OpenRMF displays a platform with a large number of integrated development tools environment, and, as Eclipse and Loftus, a visual interface greatly facilitates the development tasks.

## 6. Conclusions

For many of the complex robotics systems application contexts, a solution based on Multi-Agent Systems is, in our opinion,