

# Software and Robotics Workshop

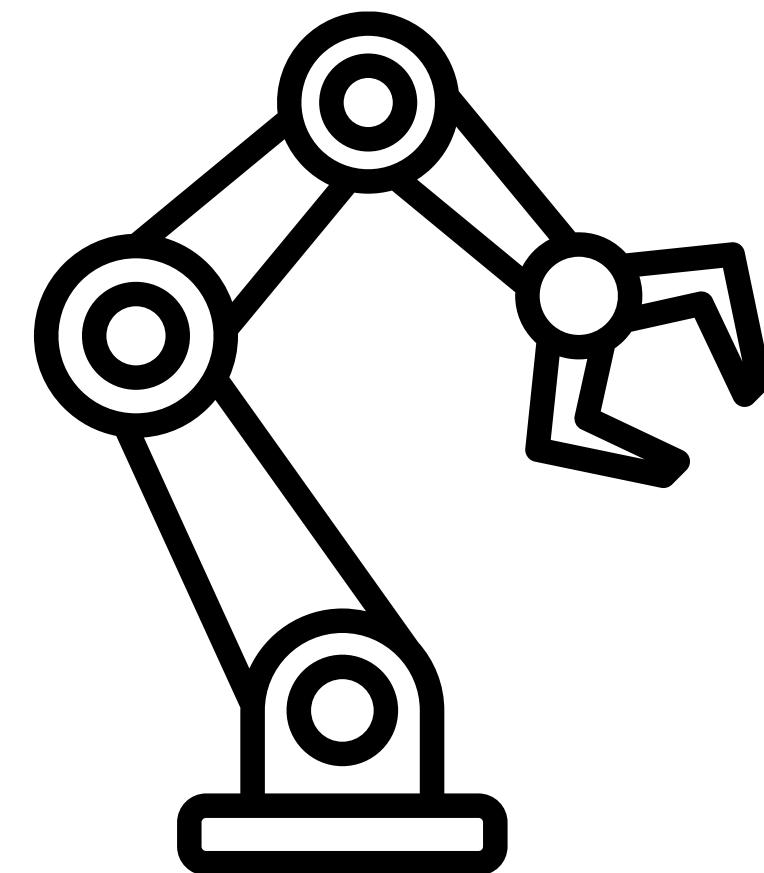
C++ and Robotics Experience

September 2024



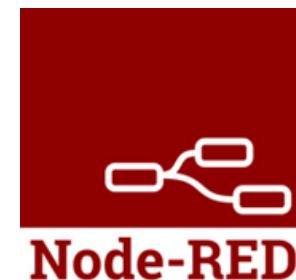
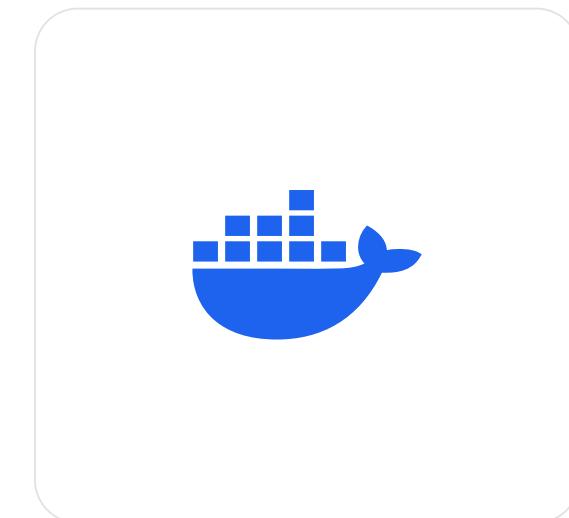
MAKERSPACE

AIRBUS



# Ziel des Kurses

- Grundlagen Softwareentwicklung für autonome Systeme
- Robot Operating System
- Grundlagen C++, Docker, Ubuntu, Git

 ROS

# About me

Daniel Brunner

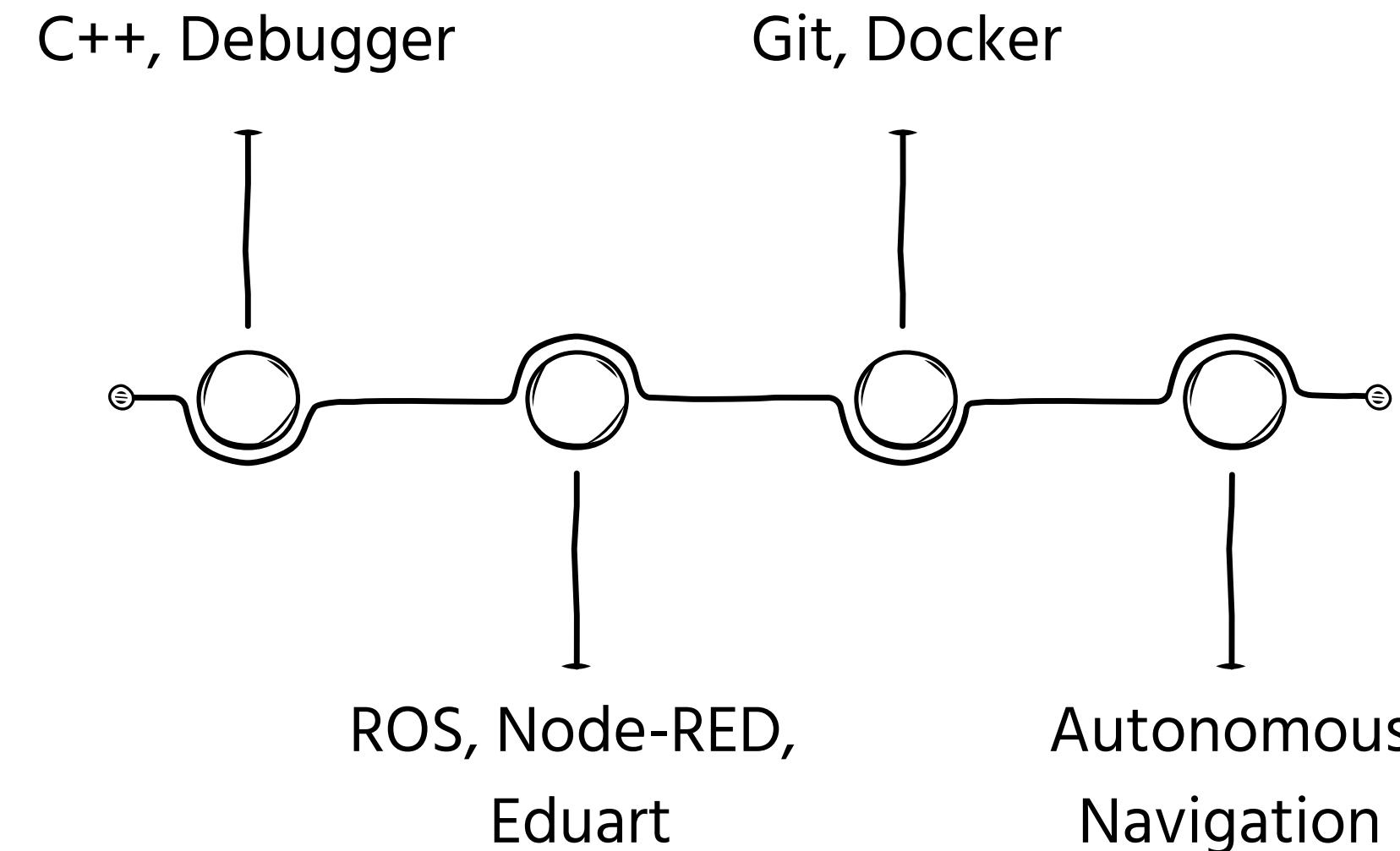
Master für Maschinenbau und Mechatronik, TUM

2 Jahre @ TUfast, Team Lead für Regelungstechnik, Autonomes Fahren

Masterarbeit für TUM Autonomous Motorsports, Softwareentwicklung, Mercedes AMG

Qualifying Lap

# Überblick



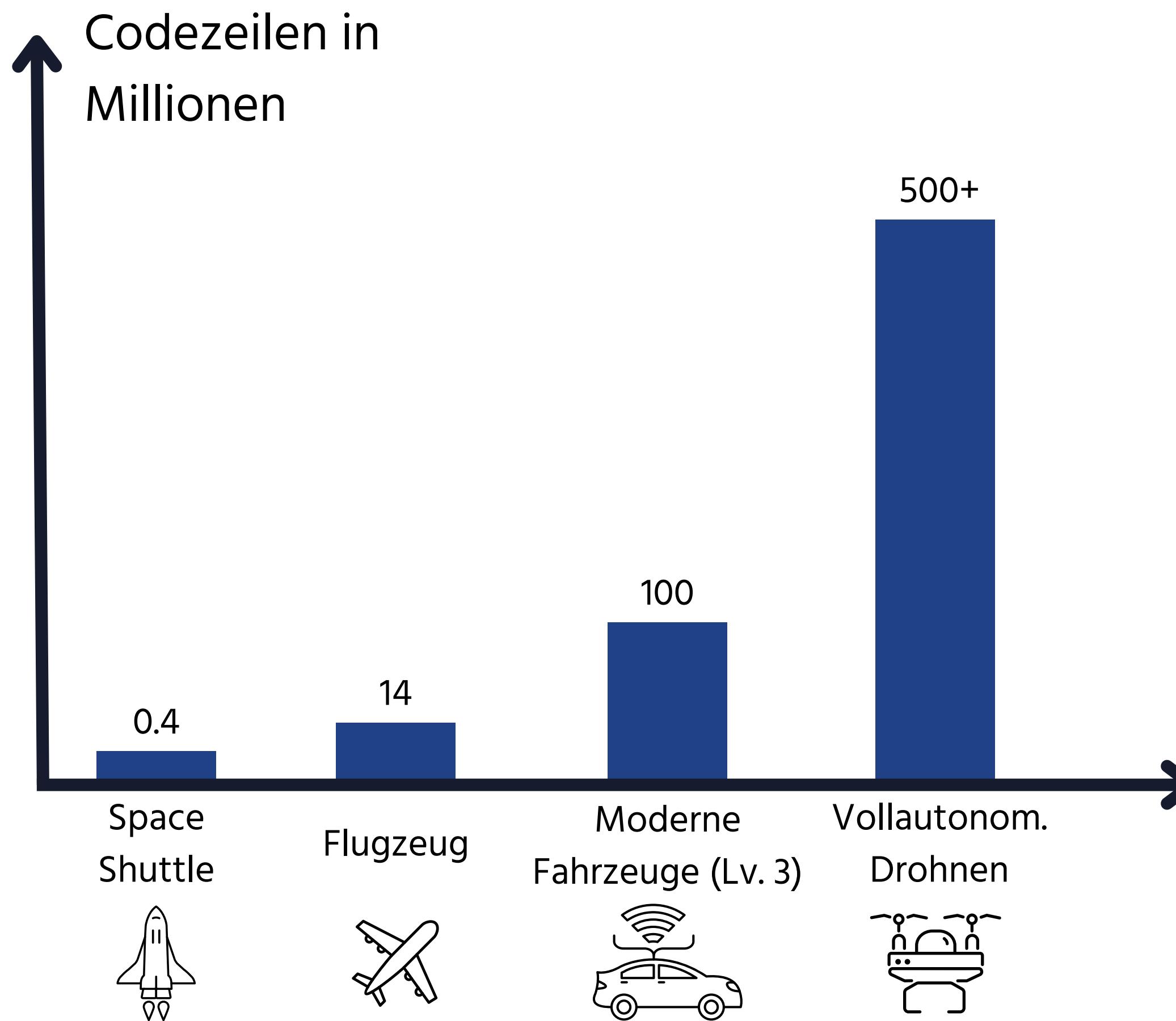
# Tag 1: Autonome Systeme, Ubuntu, Programmierung, C++

Vormittag:

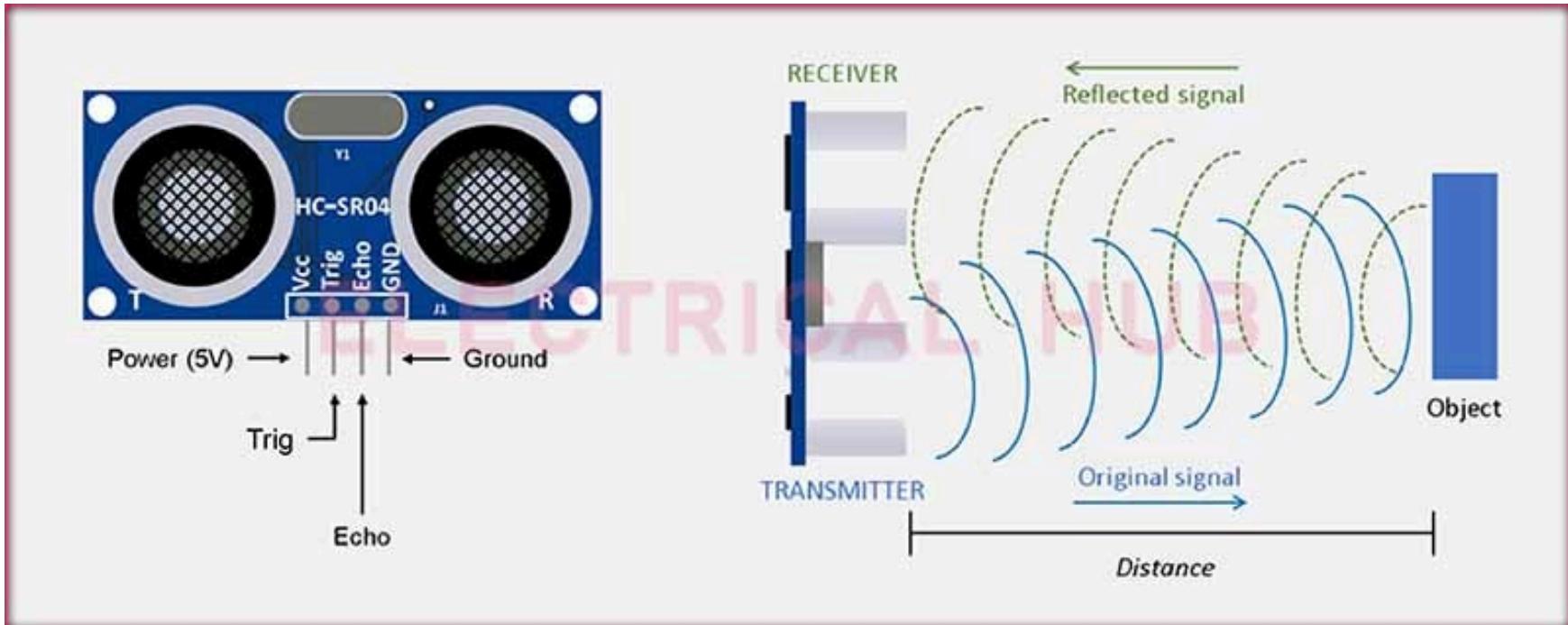
- Übersicht über Autonome Systeme
- Einstieg in Ubuntu
- Hangman in C++

Nachmittag

- ROS
- Erste Talker und Listener Node

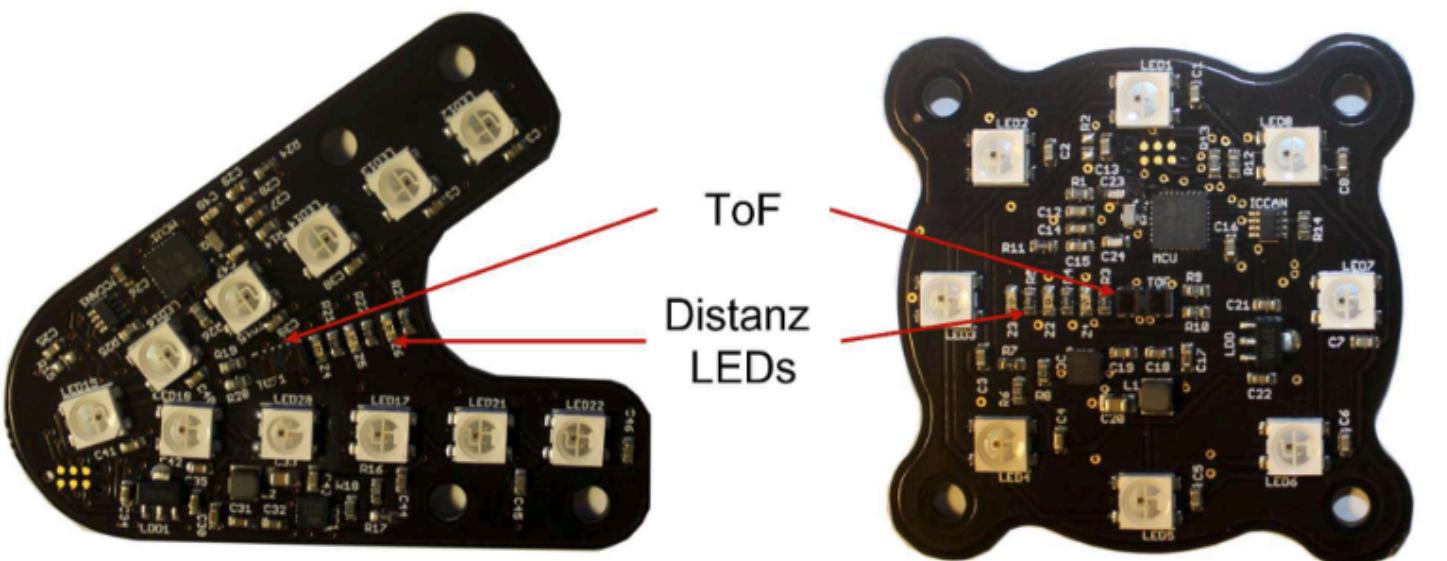
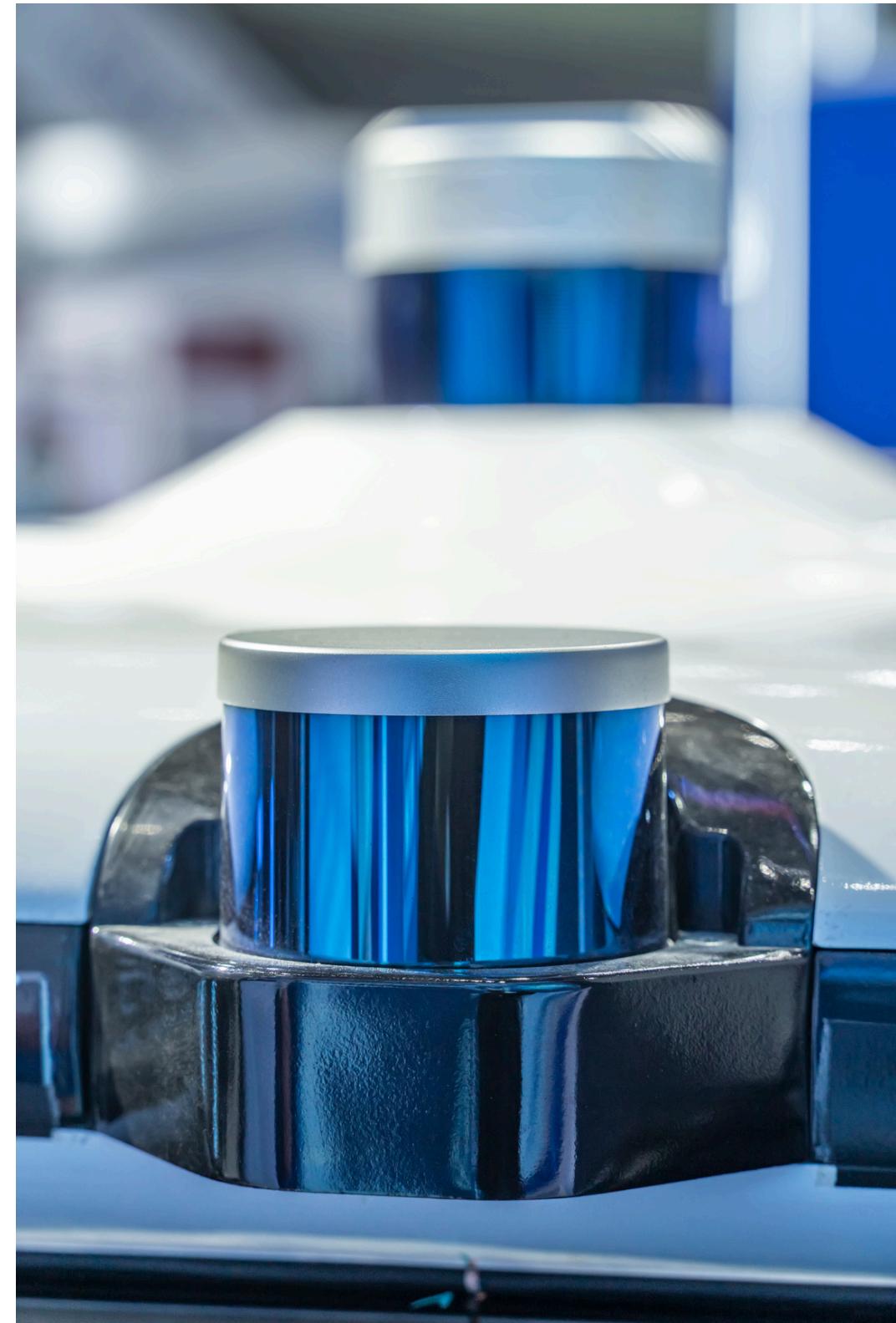


## Radar (Radio Detection And Ranging)



Quelle: <https://azadtechhub.com/ultrasonic-sensors-a-comprehensive-guide/>

## Lidar (Light Detection And Ranging)

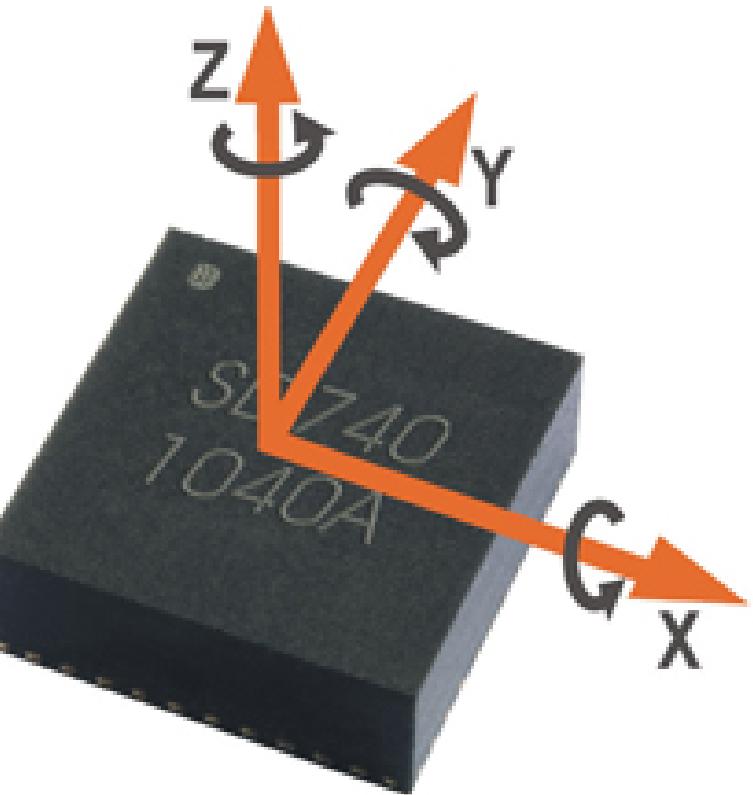


## Kamera



Quelle: <https://www.mensjournal.com/gear/the-guide/lg-developing-car-camera-with-integrated-heater-for-autonomous-cars>

## IMU

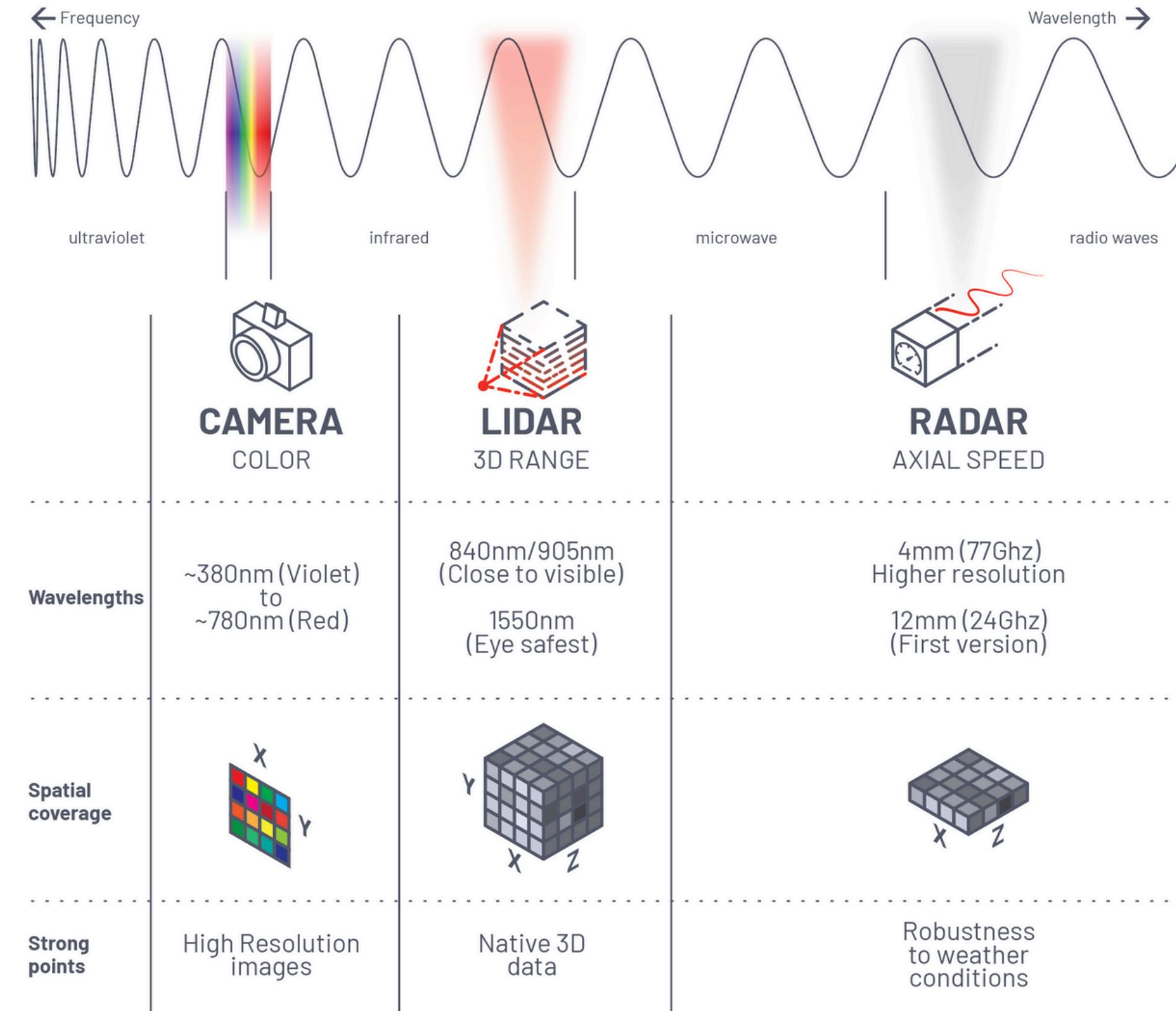


Quelle: <https://www.designworldonline.com/6dof-sensors-improve-motion-sensing-applications/>

## GPS

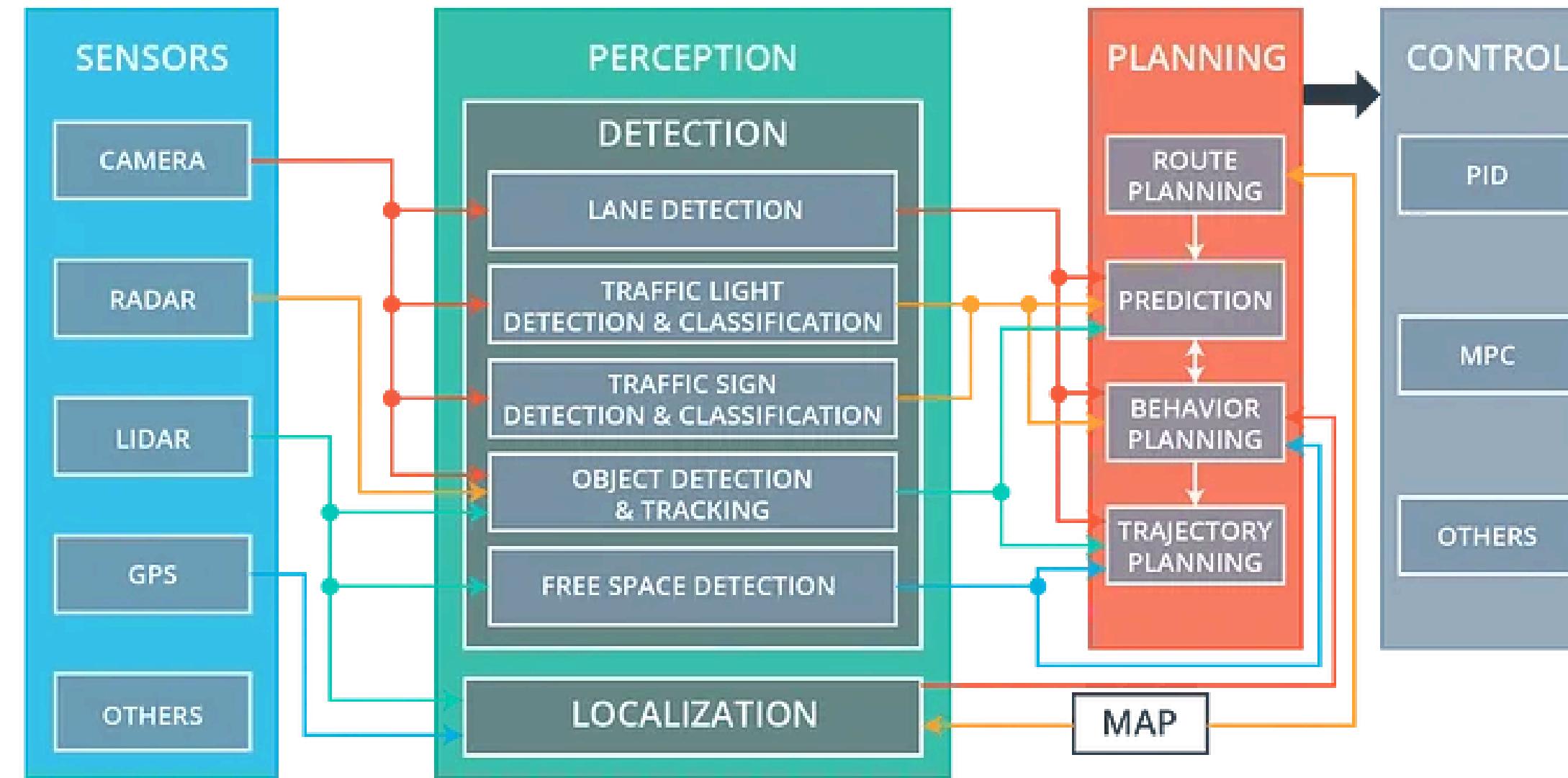


# Unterschied zwischen Kamera, Lidar und Radar



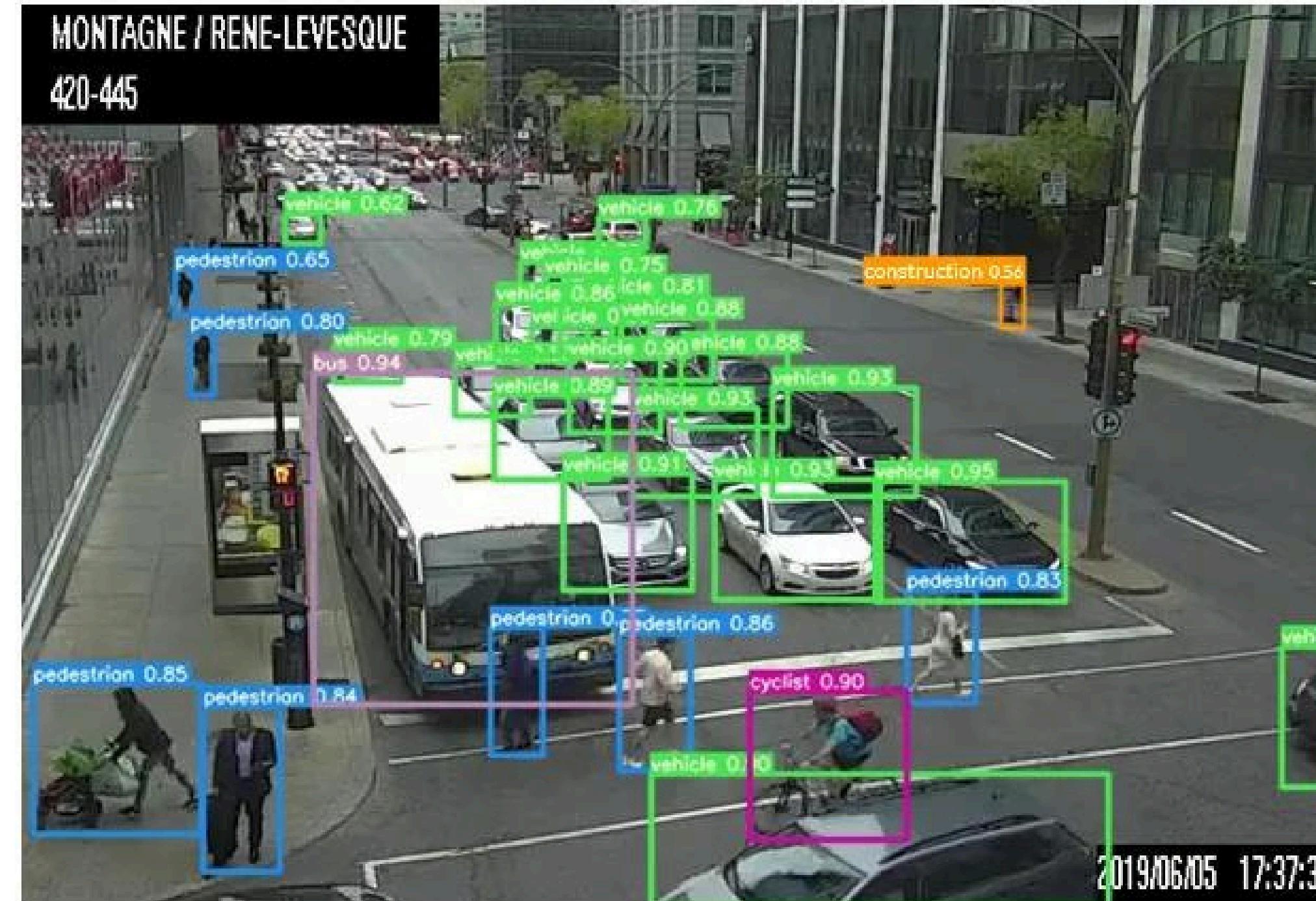
	Radar	Lidar	Camera
Reichweite	++	o	+
Auflösung	-	+	++
Sichtfeld	-	++	+
Geschwindigkeit	++	o	-
3D-Wahrnehmung	-	++	o
Objekterkennung	-	o	++
Ausfallsicherheit	++	o	-
Kosten	+	--	++
Platzverbrauch	+	--	+

# Software Stack eines UAVs



Quelle: Udacity Self Driving Car Nano Degree Program — Autonomous System overview

## Objektklassifikation und -detektion

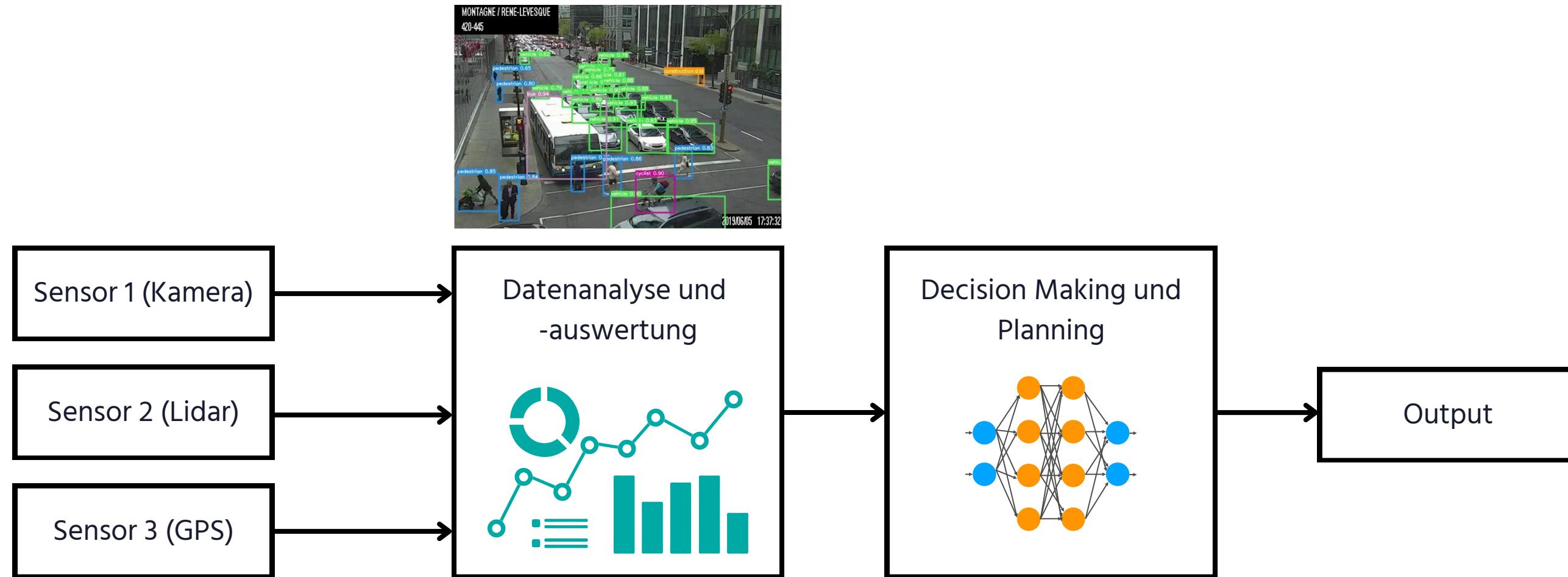


Quelle: <https://towardsdatascience.com/detecting-objects-in-urban-scenes-using-yolov5-568bd0a63c7>

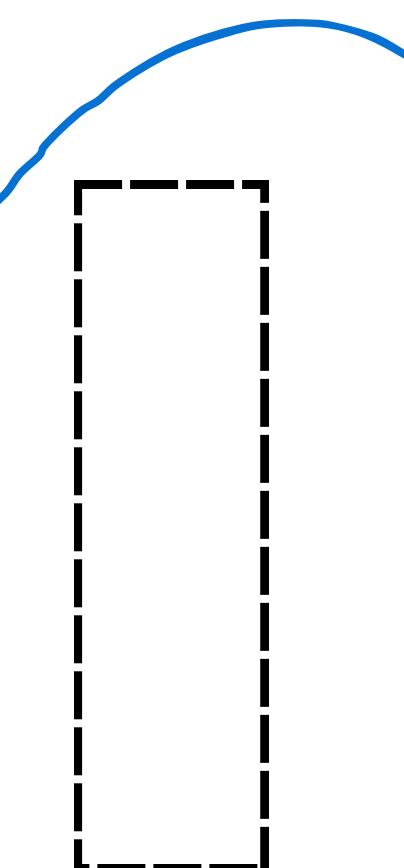
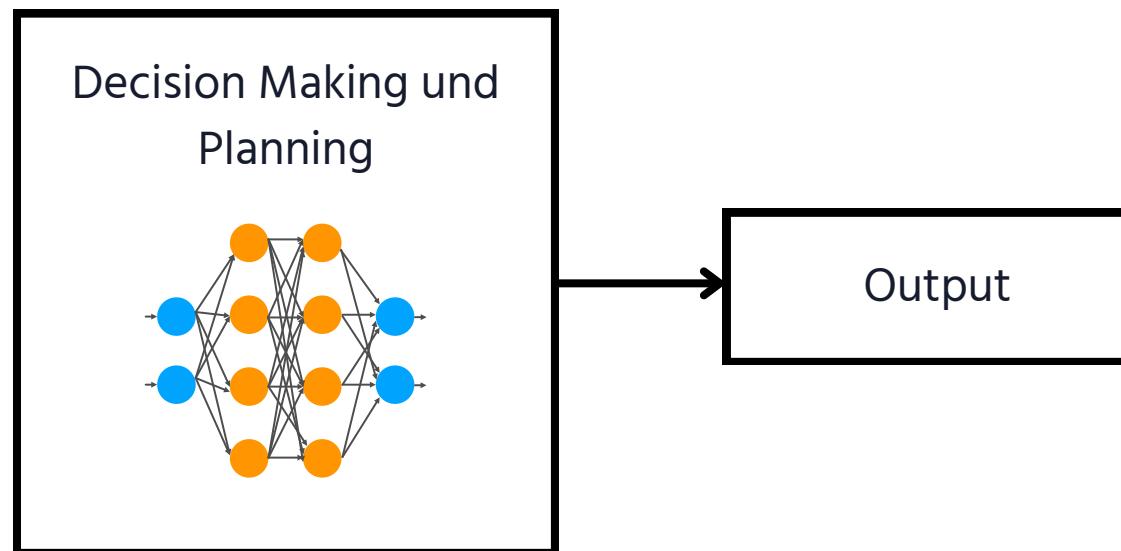
# Objektklassifikation und -detektion



# Sensor-Fusion



# Planning and Control

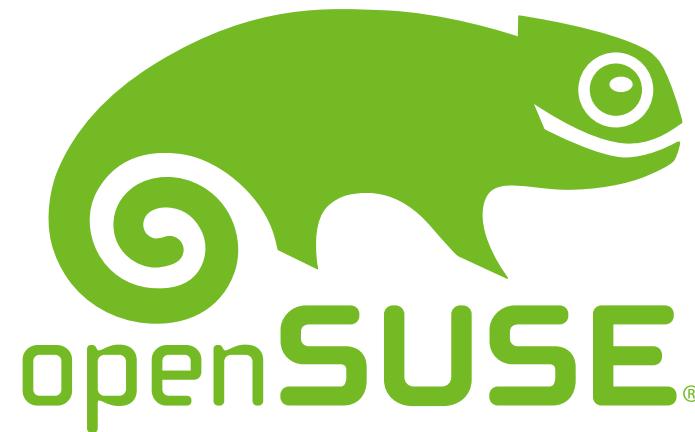


# Coffee Break!



# Linux Intro

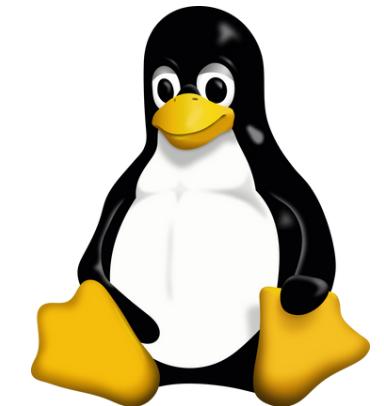




# Was ist Linux?



debian



# Why Linux?

free

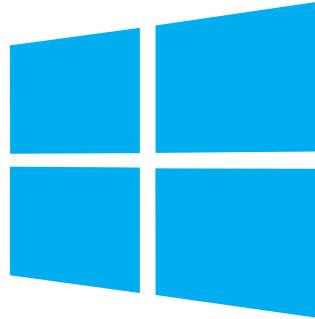
Bessere  
Performance

Admin-Rechte

Bessere Appstores -  
und management

Bash

# Unterschiede zu Windows



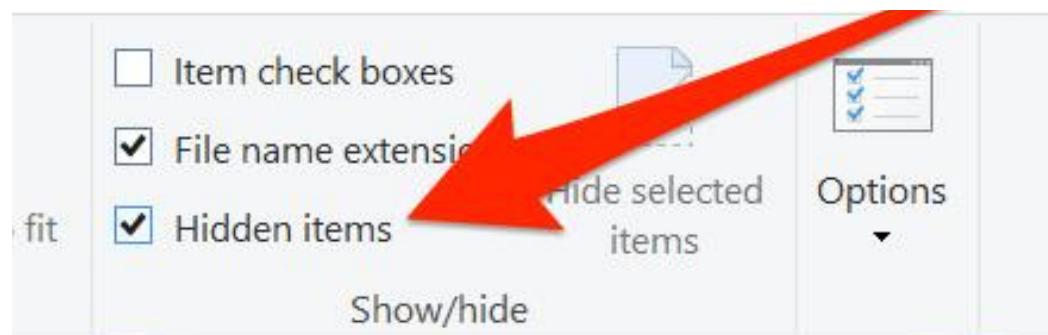
ROOT

C:\longrightarrow /

HOME C:\Benutzer\&lt;username&gt;\longrightarrow /home/&lt;benutzer&gt; oder ~/

CASE

“a” = “A”\longrightarrow “a” ≠ “A”



HIDDEN

Name startet mit “.”

EXECUTABLE

.exe format\longrightarrow Keine “extension”

# Linux - Terminal

Terminal öffnen	<b>CTRL + ALT + T</b>
Aktuelles Verzeichnis anzeigen	<b>pwd</b>
Verzeichnis wechseln	<b>cd, cd ., cd &lt;path&gt;</b>
Verzeichnis erstellen	<b>mkdir &lt;dir_name&gt;</b>
Datei erstellen	<b>touch</b>
Datei verschieben/umbenennen	<b>mv &lt;file_name&gt;</b>
Textdatei bearbeiten	<b>gedit &lt;file_name&gt;</b>
Inhalt des Verzeichnisses auflisten	<b>ls, ls -a (for hidden files)</b>
Datei/Verzeichnis löschen	<b>rm &lt;file_name&gt;, rm -r &lt;dir_name&gt;</b>
Autovervollständigung	TAB key
Vom Paketmanager installieren	<b>sudo apt install &lt;package_name&gt;</b>
Schnellreferenz / Handbuchseiten	<b>whatis &lt;cmd&gt;, man &lt;cmd&gt;</b>
Ausführbare Datei starten	<b>./&lt;executable_name&gt;</b>

# C++ Einstieg

Ein Programm besteht aus den folgenden Grundbausteinen:

- **Sequenzen:** Befehle werden in definierter Reihenfolge ausgeführt.
- **Auswahl:** Anhand von Bedingungen können Sequenzen verändert werden.
- **Wiederholung:** Sequenzen können mehrfach ausgeführt werden.

# C++ Einstieg

Computer verstehen nur “Maschinensprache”.

Programmiersprachen existieren, um die Maschinensprache für Menschen verständlich zu machen.

Programme werden insbesondere dadurch verständlich, dass eine einzelne Zeile Code viele Maschinenbefehle darstellt.

Compiler übersetzen Code in Maschinensprache.  
Interpreter führen den zu jeder Zeile Code gehörenden Code individuell aus.

```

mov  [ (SRB.CDB) + 0],01Bh
mov  [ (SRB.CDB) + 4],0
mov  [ (SRB.TargetID)], 2
push SEG SRB
push OFFSET SRB
call [ASPI_Entry]
call [ASPI_Entry]
add  sp,4
mov  dx,OFFSET OKMsg
xor  ah,ah
mov  al,[ (SRB.Status)]

```

Eject SCSI/CDROM Drive  
NASM x86 Dialekt

# C++ Elemente

## **Header einfügen:**

- `#include <iostream>` für Ein- und Ausgabe
- Weitere Header je nach Bedarf (z. B. `<string>`)

## **main() Funktion:**

- Startpunkt jedes C++-Programms
- Muss einen Rückgabewert vom Typ `int` haben (normalerweise `return 0`)

## **Variablen-deklaration:**

- Angabe des Datentyps vor der Variable (`int`, `float`, `string`, `bool`)

## **Eingabe und Ausgabe:**

- `std::cout` für die Ausgabe
- `std::cin` für die Eingabe

## **Kontrollstrukturen:**

- `if-else` für Bedingungen
- `for, while` für Schleifen

# C++ Elemente

## Funktionen:

- Ermöglichen Code-Wiederverwendung
- Haben Rückgabewert, Namen und Parameter

## Arrays:

- Speichern mehrere Werte desselben Typs
- Zugriff auf Werte durch Indizes

## Objektorientierung:

- Klassen definieren, die Attribute und Methoden enthalten
- Objekte basieren auf Klassen

## Kommentare:

- Einzeilige Kommentare: // Kommentar
- Mehrzeilige Kommentare: /\* Kommentar \*/



Projektstruktur



Textsuche



git



Ausführung



Extensions

...



X 0 △ 0 ⌂



# Aufgabe: Hangman

Schreibt ein Programm, dass das berühmte Spiel “Hangman” begleitet und anzeigt.

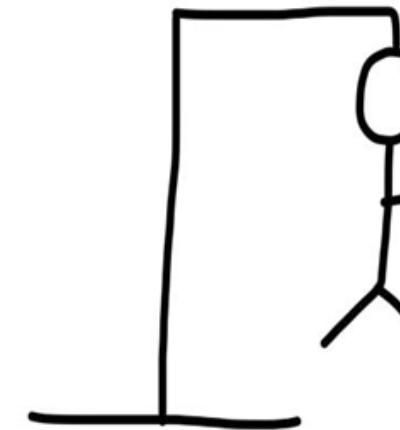
- Es wird ein zufälliges Wort “ausgedacht” - zum Beispiel “AirbusAtMakerSpace” oder “programming”
- mittels `std::cin >>` werden einzelne Buchstaben eingelesen.
- Falls der eingelesene Buchstabe im Wort vorhanden ist, wird er das um diesen Buchstaben ergänzte Wort ausgegeben
- Falls der eingelesene Buchstabe nicht im Wort vorhanden ist, wird ein weiteres Körperteil dem Hangman hinzugefügt (siehe Tipp)

Sobald die Figur komplett ist, hat der Spieler Verloren.

Falls das Wort vorhererraten wurde, hat der Spieler gewonnen.

Tipp: <https://gist.github.com/chrishorton/8510732aa9a80a03c829b09f12e20d9c> (Hangman)

Tipp: Hangman Art als array



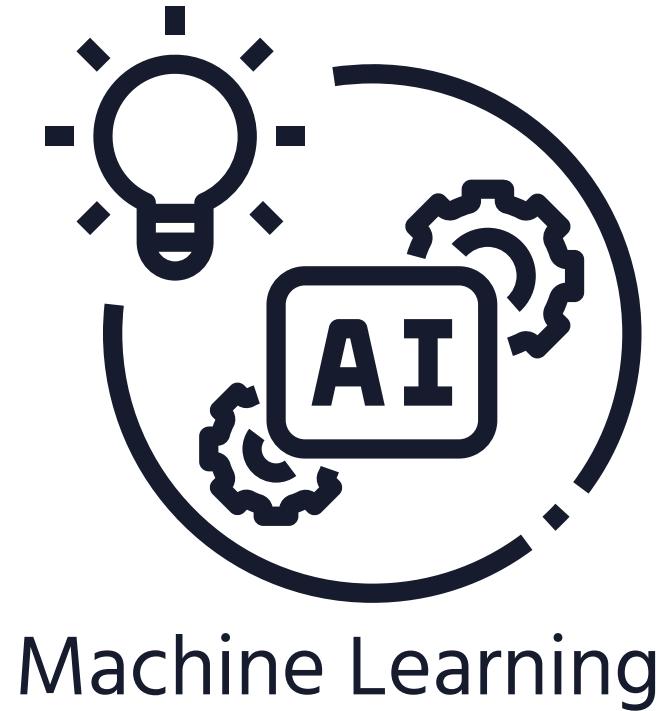
H\_A\_N\_ \_M\_A\_N



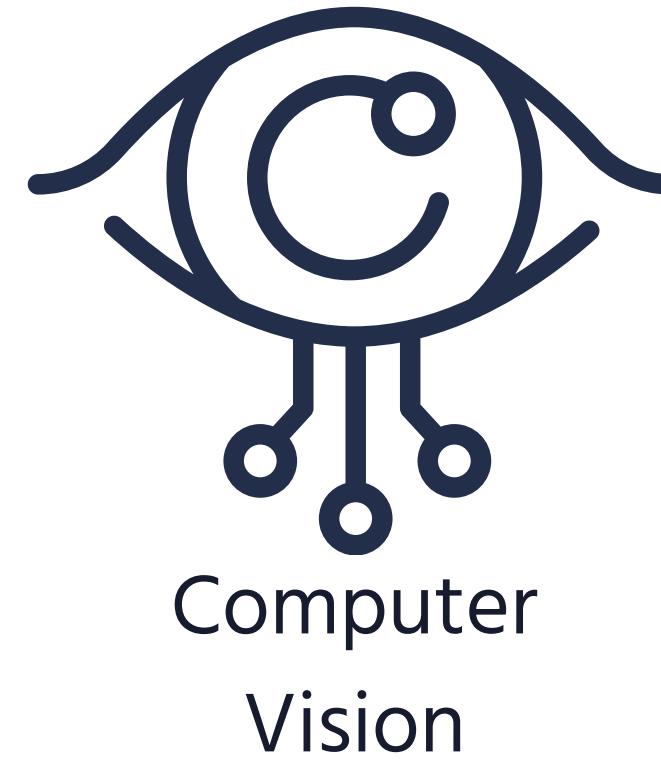
# ROS Theorie



Robot Operating System



Machine Learning



Computer  
Vision



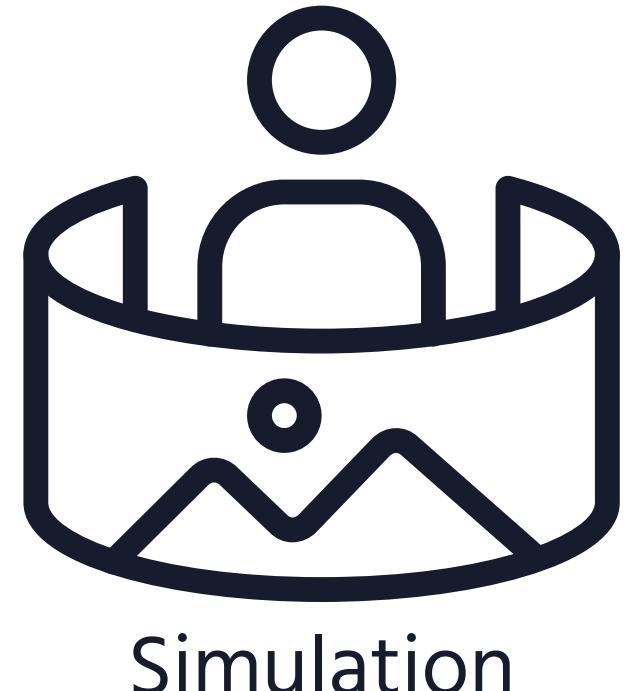
Visualisierung



Data Logging



Objekterkennung



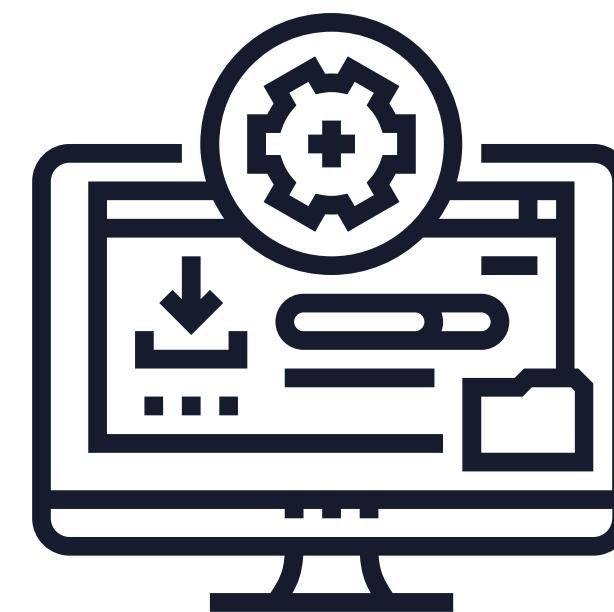
Simulation



Kommunikation



Trajektorienplanung



Hardwaretreiber

# ROS

Robot Operating System



Robot Control

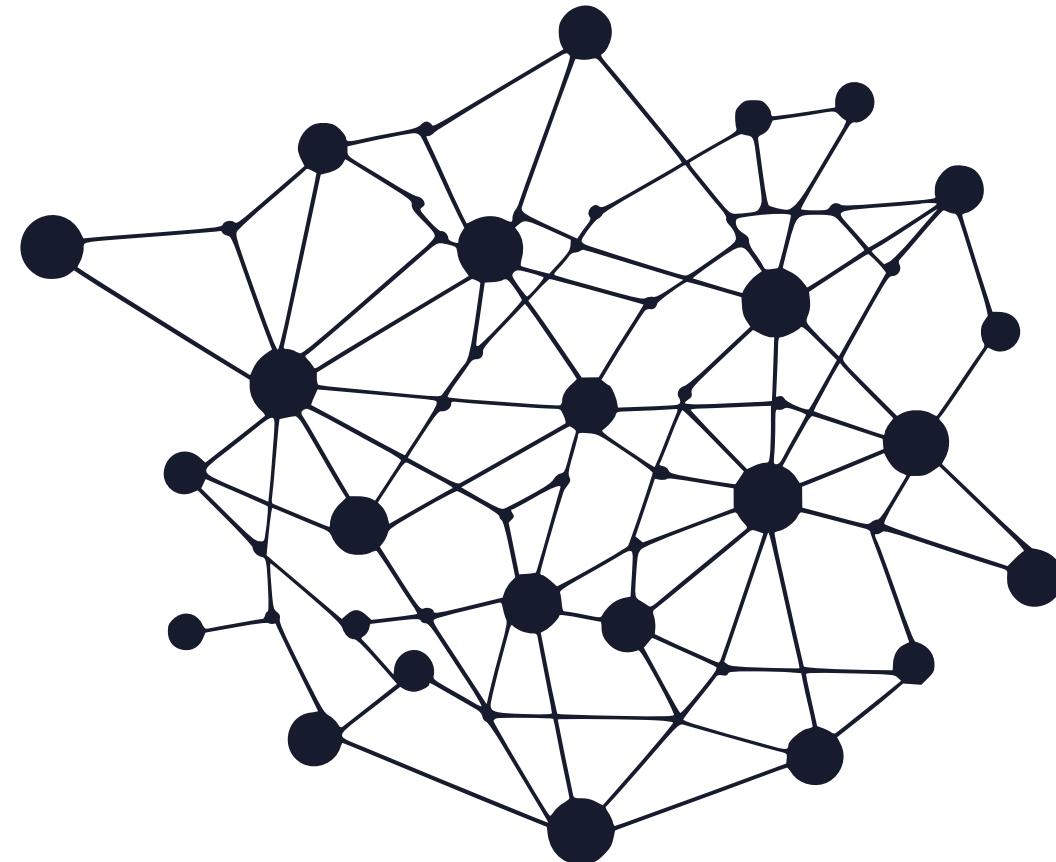


# Wieso ROS?

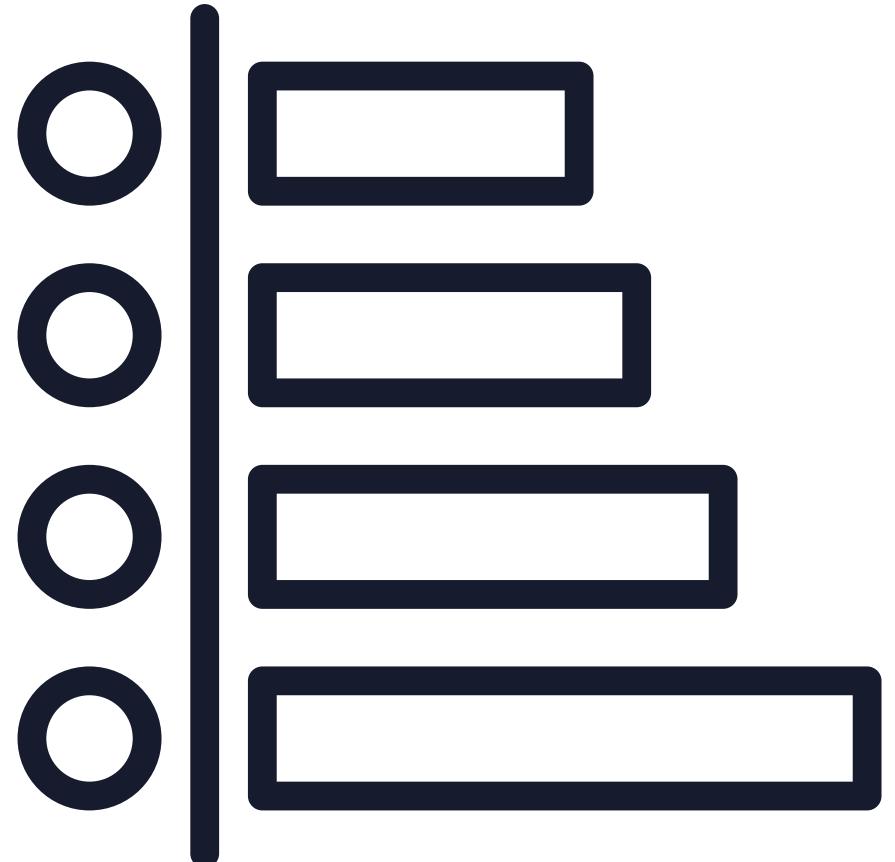
- Modularität
- Wiederverwendbarkeit
- Community-Unterstützung
- Interoperabilität
- Werkzeuge und Simulationen
- Standardisierung
- Skalierbarkeit
- Echtzeitfähigkeiten
- Bildverarbeitung und KI
- Quelloffenheit

# Daten werden in

Nodes



Topics



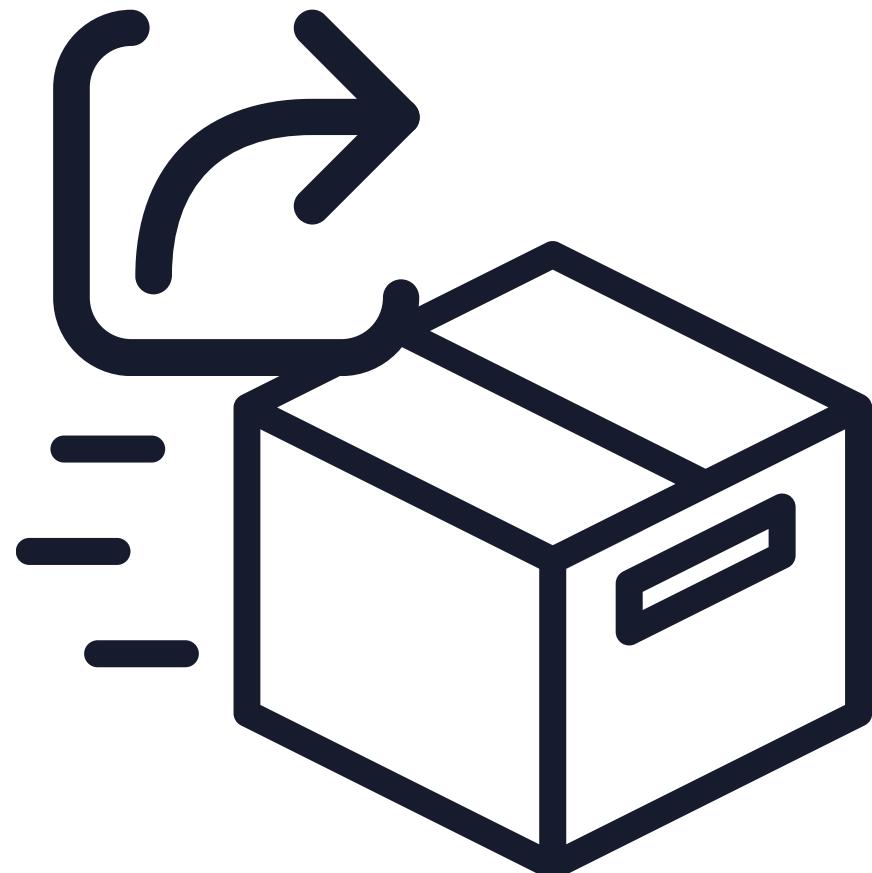
Messages



gespeichert

# Daten werden durch

Publisher



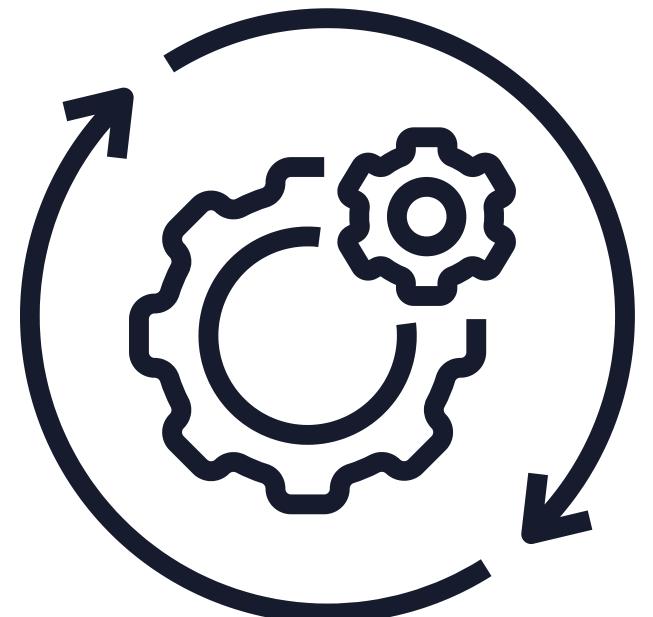
Subscriber



Services



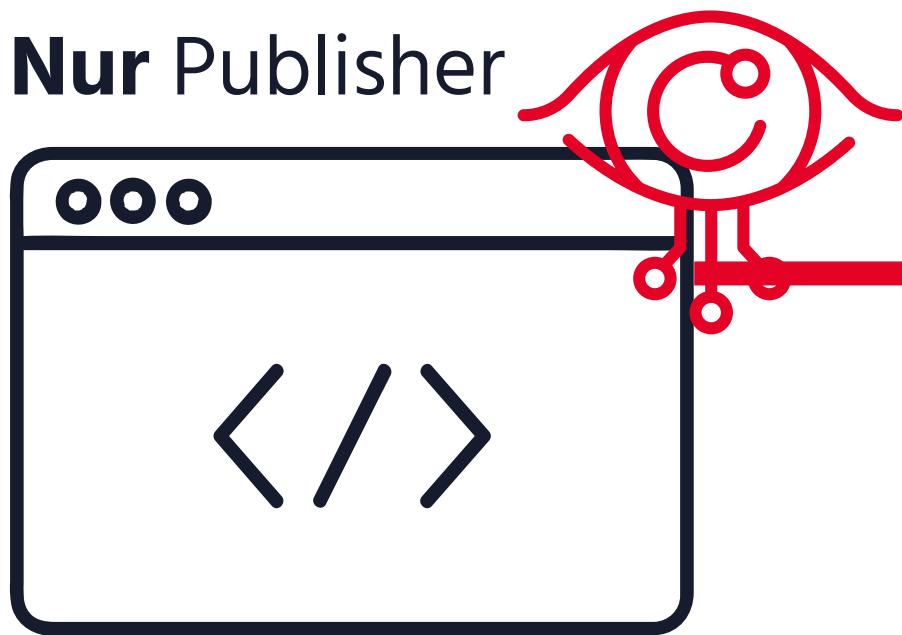
Actions



# herumgeschickt

Node "Perception"

**Nur Publisher**

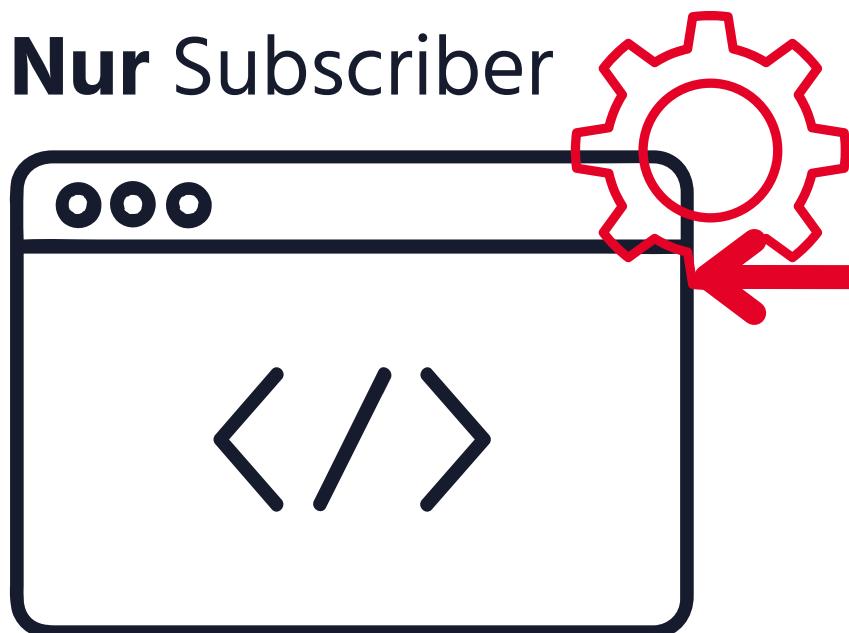


Position

Topic 1

Node "Motor Control"

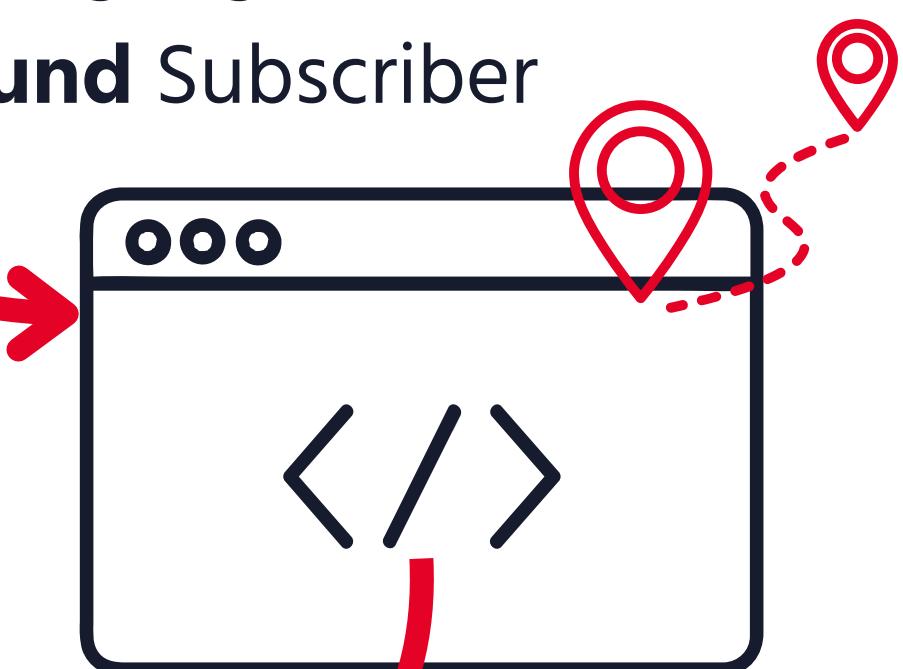
**Nur Subscriber**



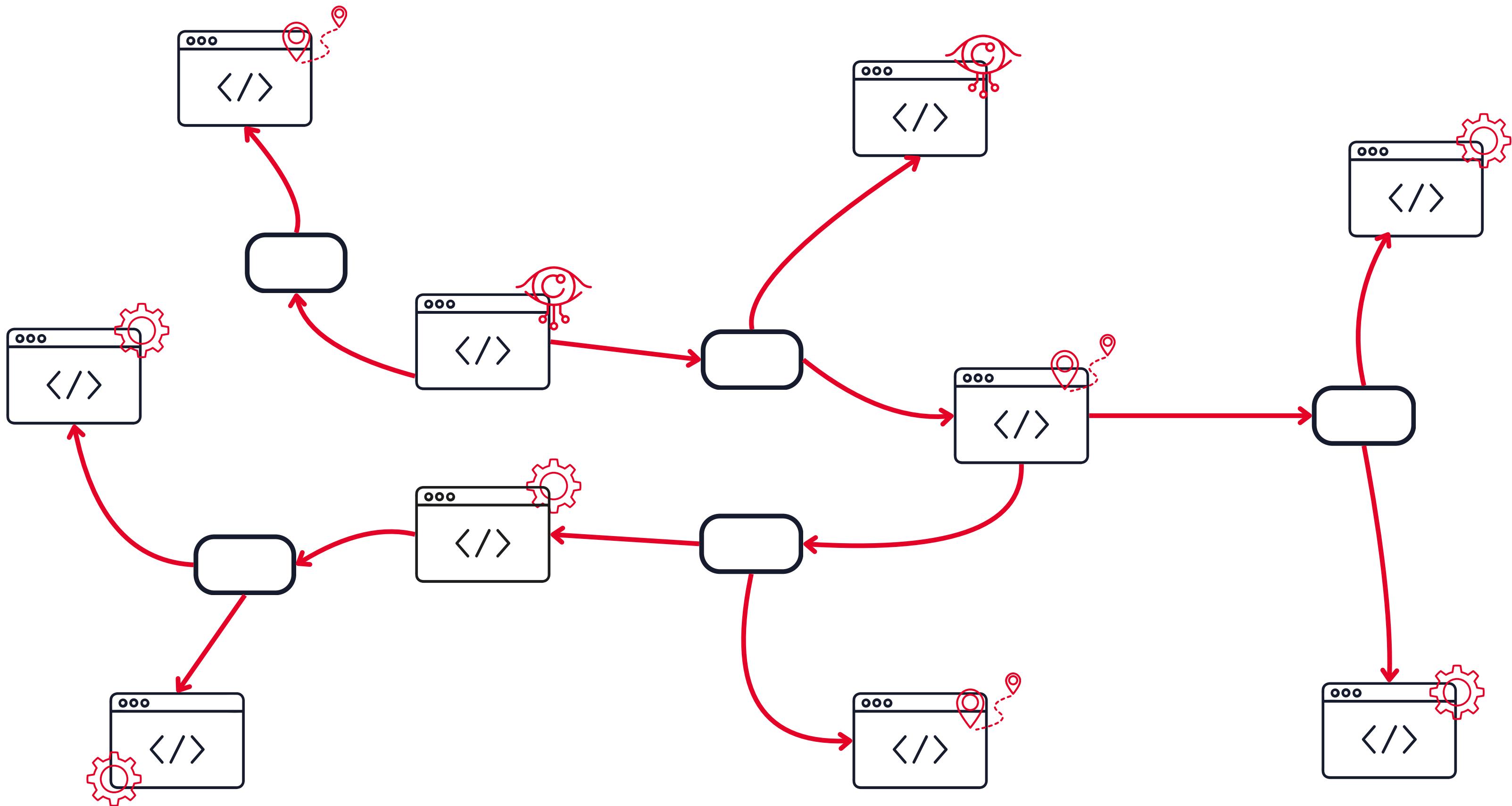
Beschleunigung  
und Lenkwinkel

Topic 2

Node "Bewegungs-Planner"  
Publisher **und** Subscriber

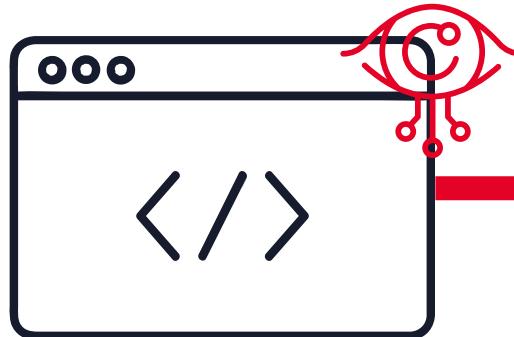


# ROS Graphen

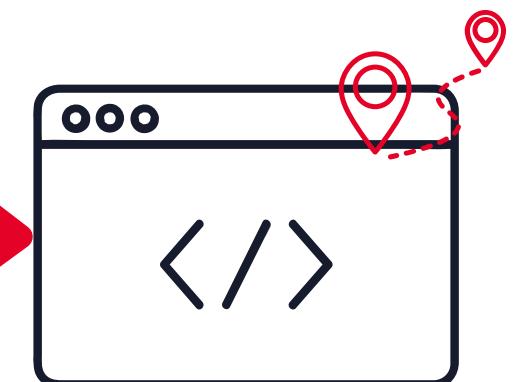


# Services

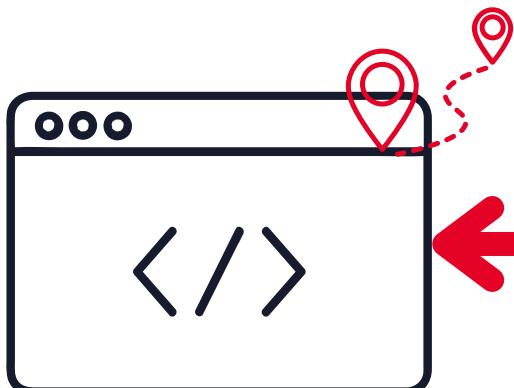
Topics:



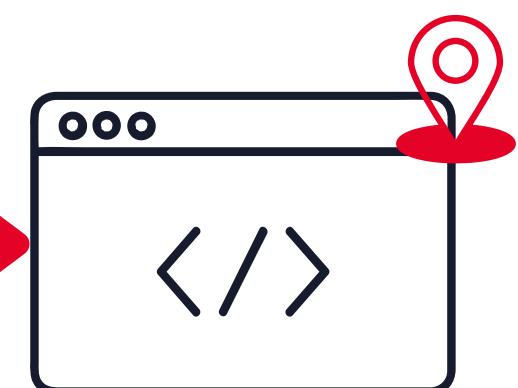
Nur in eine Richtung!



Services:



Beide Richtungen!



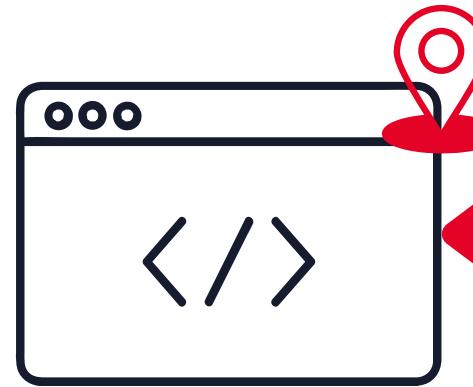
Trajectory Planner

Position Calculation

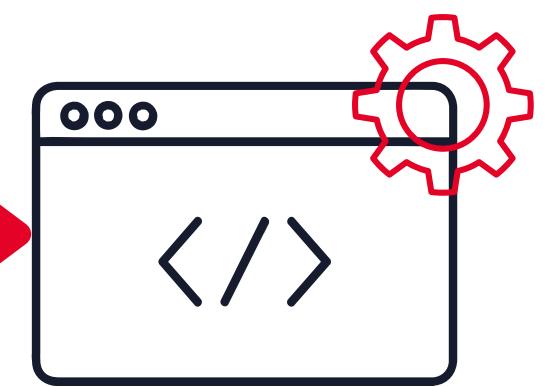
Kurze Berechnungen

# Actions

Action client



Action server



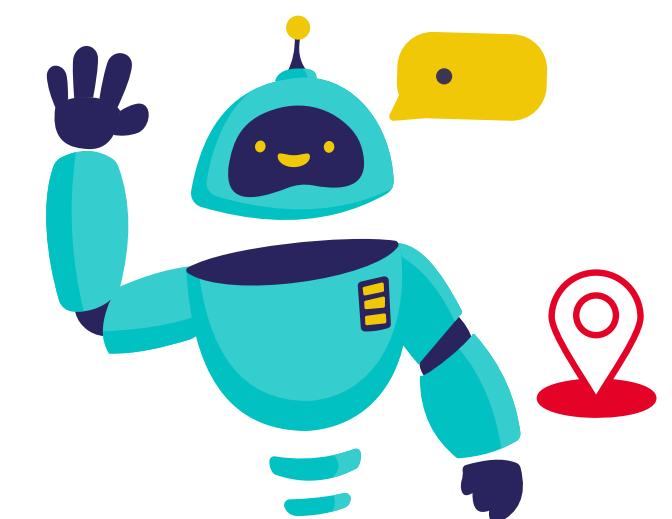
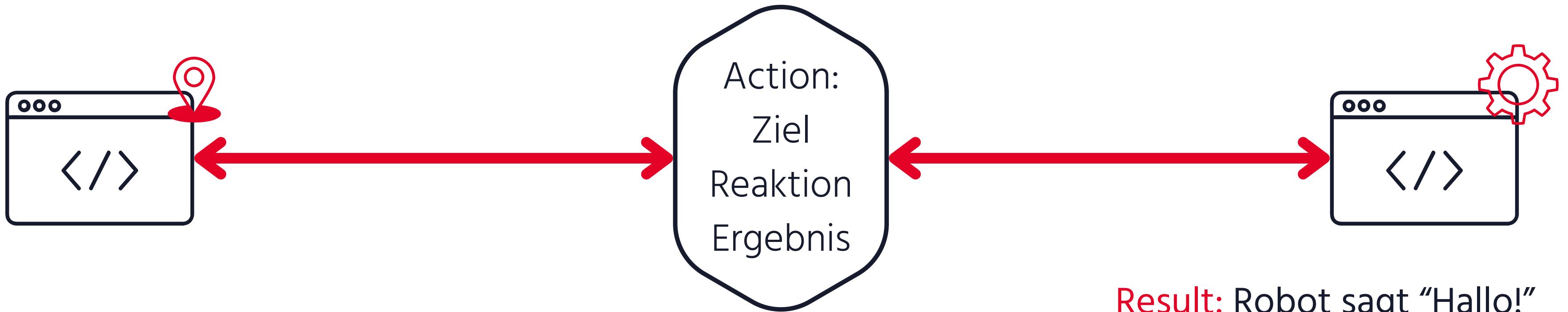
Roboter bewegt sich = Reaktion

Distanz zum Ziel



Kommunikation über eine längere Zeit

# Actions



Kommunikation über eine längere Zeit

# Zusammenfassung

	Regelmäßige Kommunikation	Kurz dauernde Kommunikation	Lang dauernde Kommunikation
Topics	eine Richtung		
Services		kurze Berechnung	
Actions			Ziel, Reaktion und Ergebnis

# ROS Workspace

```
workspace_folder/  
src/  
  cpp_package_1/  
    CMakeLists.txt  
    include/cpp_package_1/  
    package.xml  
    src/  
  
  py_package_1/  
    package.xml  
    resource/py_package_1  
    setup.cfg  
    setup.py  
    py_package_1/  
...  
  cpp_package_n/  
    CMakeLists.txt  
    include/cpp_package_n/  
    package.xml  
    src/
```

Ein ROS Workspace beinhaltet den ganzen Code, der für die Trajektorienplanung, Objekterkennung oder Regelung eines autonomen Roboters nötig ist.  
Die Packages kann man entweder in **C++** oder **Python** schreiben. Eine wichtige Datei ist die **package.xml**, in dieser Datei steht eine Liste von Abhängigkeiten anderer Packages.  
Die CMakeLists.txt für Packages in C++ ist notwendig für das Kompilieren.

# Unser erstes eigenes Package - Talker & Listener

Terminal:

```
mkdir ros2_ws
cd ros2_ws
mkdir src
cd src
ros2 pkg create --build-type ament_cmake --license Apache-2.0 cpp_pubsub
cd cpp_pubsub
cd src
touch publisher_member_function.cpp
touch subscriber_member_function.cpp
```

Kopiere nun jeweils den Code von

<https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html>

Wir werden uns den Code in den nächsten Schritten genauer anschauen.

# publisher\_member\_function.cpp

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;
```

Am Anfang des Codes befinden sich die standardmäßigen C++-Headerdateien, die wir verwenden werden. Nach den standardmäßigen C++-Headern folgt das Einbinden von `rclcpp/rclcpp.hpp`, das es uns ermöglicht, die gebräuchlichsten Teile des ROS2-Systems zu nutzen. Zuletzt kommt `std_msgs/msg/string.hpp`, das den integrierten Nachrichtentyp enthält, den wir zum Veröffentlichen von Daten verwenden werden.

Mehr zu den std\_msgs:

[https://docs.ros2.org/latest/api/std\\_msgs/index-msg.html](https://docs.ros2.org/latest/api/std_msgs/index-msg.html)

# publisher\_member\_function.cpp

```
class MinimalPublisher : public rclcpp::Node
```

Die nächste Zeile erstellt die Node-Klasse `MinimalPublisher`, indem sie von `rclcpp::Node` erbt. Alles in diesem Code bezieht sich auf den Node.

# publisher\_member\_function.cpp

```
public:  
    MinimalPublisher()  
    : Node("minimal_publisher"), count_(0)  
    {  
        publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);  
        timer_ = this->create_wall_timer(  
            500ms, std::bind(&MinimalPublisher::timer_callback, this));  
    }
```

Der public Konstruktor benennt die Node `minimal_publisher` und initialisiert `count_` auf 0. Innerhalb des Konstruktors wird der `Publisher` mit dem Message-Typ String, dem Topic-Namen `topic` und der erforderlichen Warteschlangengröße initialisiert, um Nachrichten im Falle eines Rückstaus zu begrenzen. Anschließend wird `timer_ initialisiert`, was dazu führt, dass die Funktion `timer_callback` zweimal pro Sekunde ausgeführt wird.

# publisher\_member\_function.cpp

```
private:  
    void timer_callback()  
{  
    auto message = std_msgs::msg::String();  
    message.data = "Hello, world! " + std::to_string(count_++);  
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());  
    publisher_->publish(message);  
}
```

Die Funktion `timer_callback` ist der Ort, an dem die Message-Daten festgelegt werden und die Messages tatsächlich veröffentlicht werden. Das Makro `RCLCPP_INFO` stellt sicher, dass jede veröffentlichte Message auf der Konsole ausgegeben wird.

# publisher\_member\_function.cpp

```
rclcpp::TimerBase::SharedPtr timer_;  
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;  
size_t count_;
```

Zuletzt erfolgt die Deklaration der Felder für den Timer, den Publisher und den Zähler.

```
int main(int argc, char * argv[ ])  
{  
    rclcpp::init(argc, argv);  
    rclcpp::spin(std::make_shared<MinimalPublisher>());  
    rclcpp::shutdown();  
    return 0;  
}
```

Nach der Node-Klasse **MinimalPublisher** folgt **main**, wo der Node tatsächlich ausgeführt wird. `rclcpp::init` initialisiert ROS 2, und `rclcpp::spin` beginnt mit der Verarbeitung von Daten aus dem Node, einschließlich der Callbacks vom Timer.

# CMakeLists.txt

Wir müssen unter `find_package(ament_cmake REQUIRED)` zunächst weitere Abhängigkeiten hinzufügen:

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

Dann müssen wir jeweils für Talker und Listener die Executables hinzufügen, damit wir diese mittels `ros2 run` aufrufen können:

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
add_executable(listener src/listener_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)
```

Zum Schluss müssen wir noch angeben, wo die installierten Executables liegen sollen:

```
install(TARGETS
    talker
    listener
    DESTINATION lib/${PROJECT_NAME})
```

# package.xml

Hier müssen wir unter ament\_cmake noch unsere anderen Abhängigkeiten angeben, die in unseren Header vorkommen:

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

## Speichern nicht vergessen!

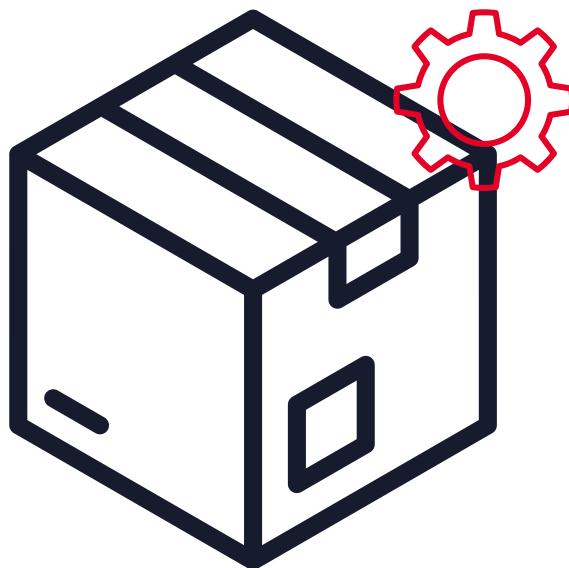
Terminal 1:

```
cd ~/ros2_ws
colcon build
. install/setup.bash
ros2 run cpp_pubsub talker
```

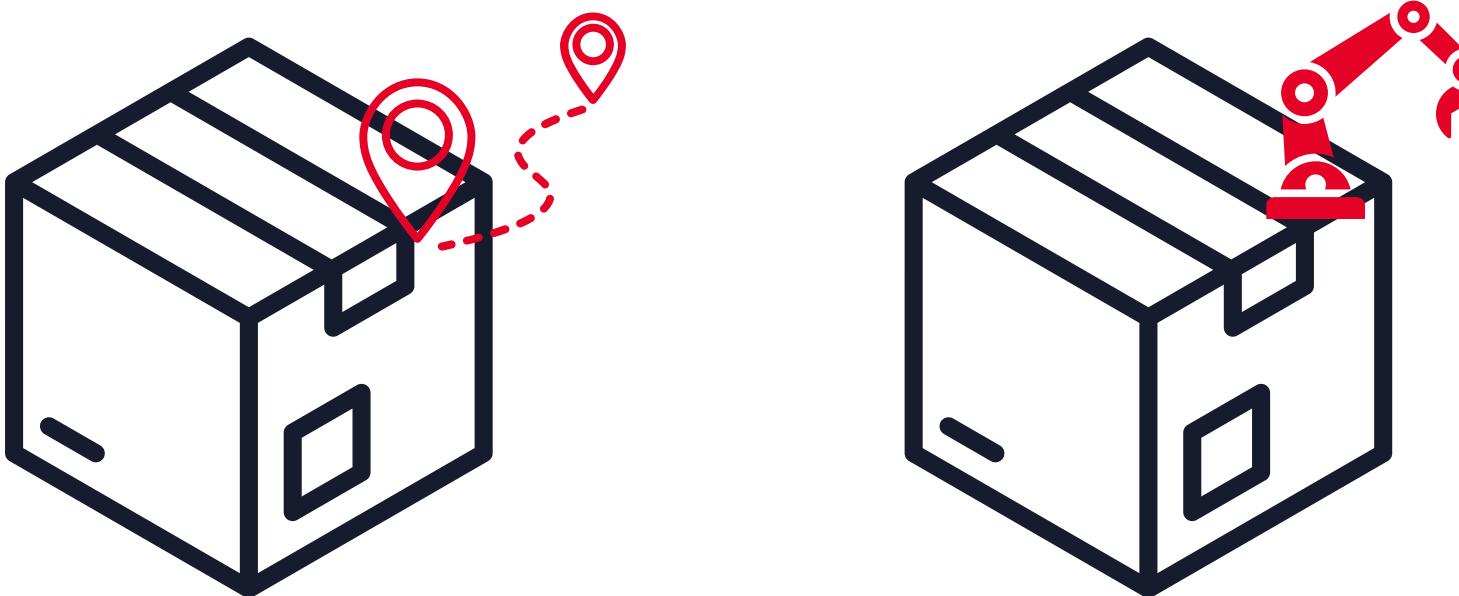
Terminal 2:

```
cd ~/ros2_ws
. install/setup.bash
ros2 run cpp_pubsub listener
```

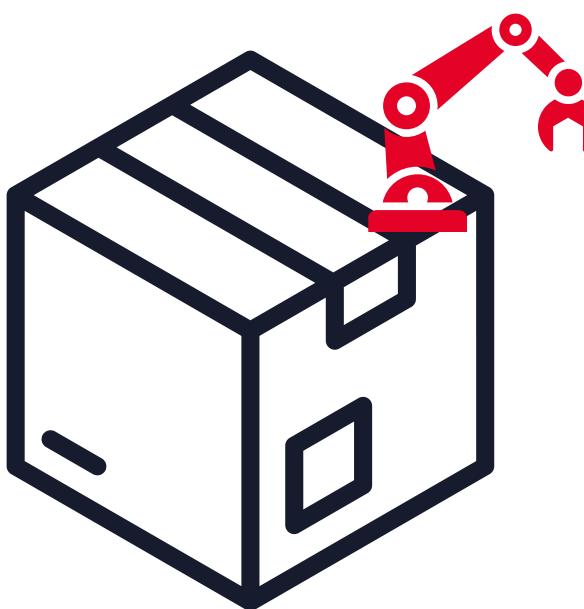
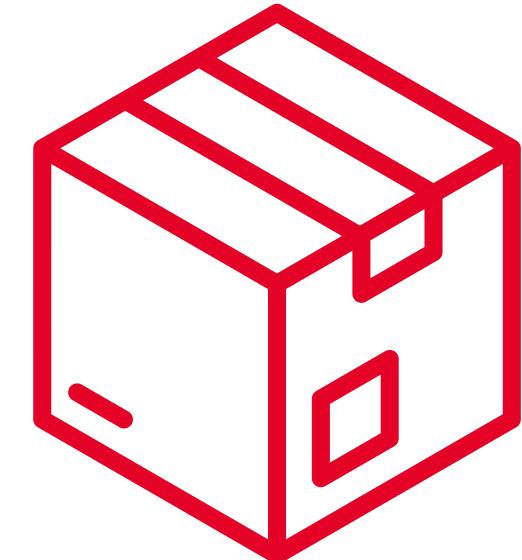
# Open source packages



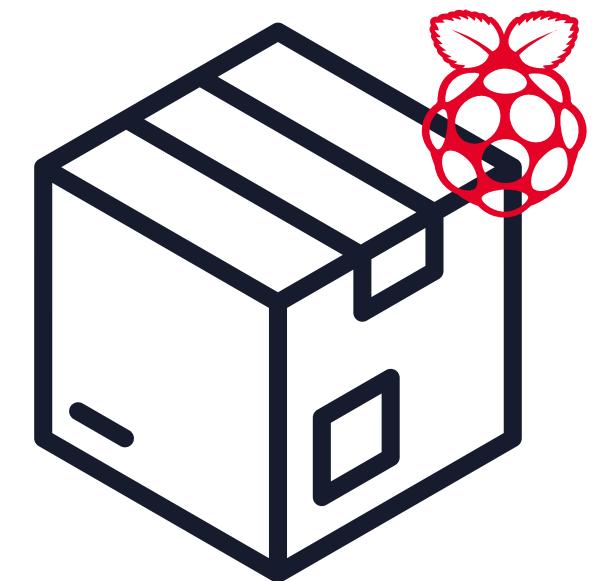
ros2\_control



Navigation2

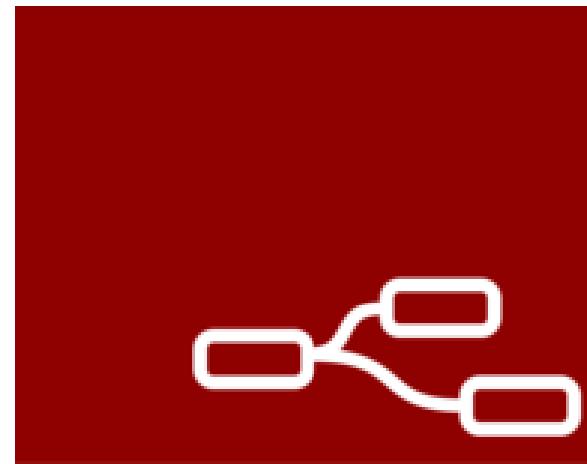


Movelt



micro-ROS

# Open source packages



Node-RED bietet eine Möglichkeit, ROS visuell zu programmieren.  
Vorteil: Man muss keinen Code schreiben!

# Weitere Tools

- `rqt`
  - Monitor
  - Plots
  - Rosbag
  - Dynamic reconfigure: Online Modifikation von Parameter
  - ...
- `rviz2`
  - Visualisierungstool für tf-frames, Marker, sensoren, Maps, etc...
  - Keine Physik!
- Simulationsumgebungen
  - Gazebo
  - Unity

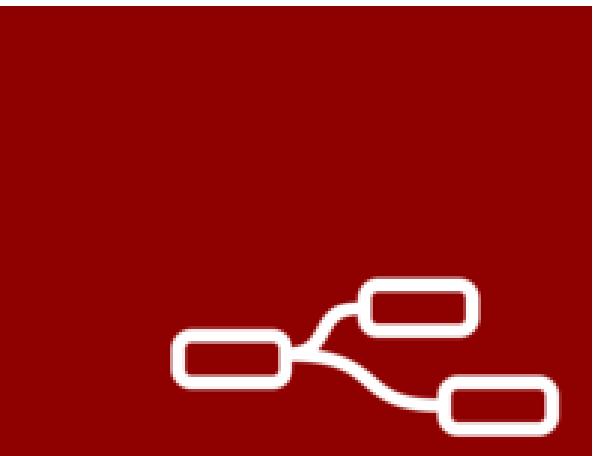
# ROS Bags

- Ein Rosbag speichert alle Daten von Messages
- Dateiformat: .db3 bzw. mcap
- Für späteres Visualisieren und Analysieren
- Aufnehmen
  - ros2 bag record --all
- Nur bestimmte Topics aufnehmen
  - ros2 bag record topic\_1 topic\_2 topic\_3
- Rosbag abspielen
  - ros2 bag play bag\_name.db3
- Rosbag langsamer (mit halber Geschwindigkeit) abspielen, also loop
  - ros2 bag play --rate=0.5 bag\_name.bag --loop

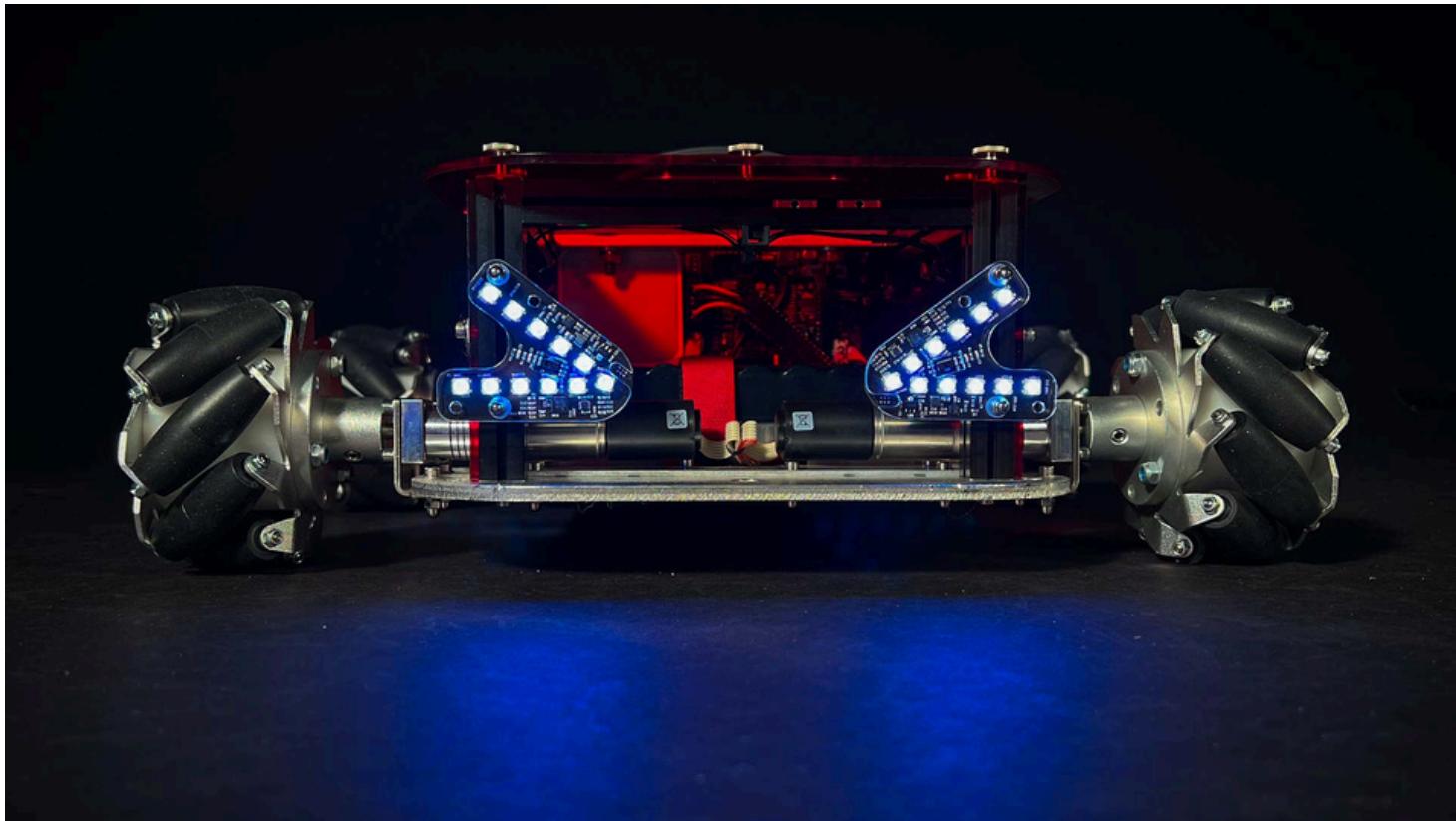
# Happy Feierabend!

Bis morgen um 9:00!

Dann geht's weiter mit:



**Node-RED**



# **Software and Robotics Workshop**

C++ and Robotics Experience

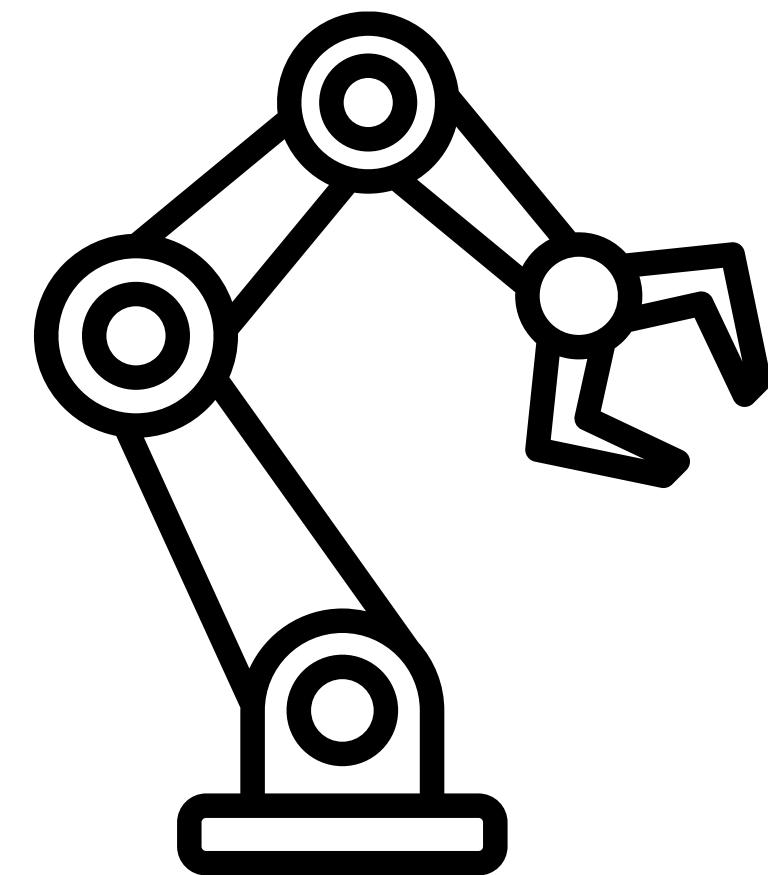
September 2024

Tag 2



**MAKERSPACE**

**AIRBUS**



# Tag 2: EduArt, Node-RED

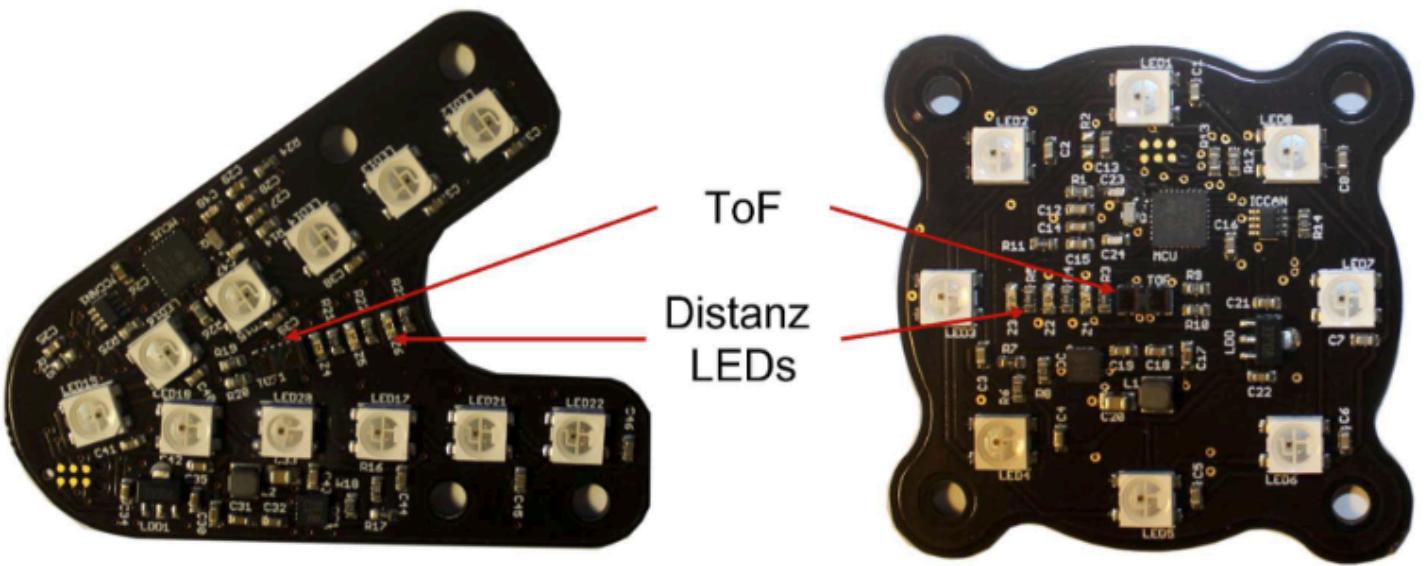
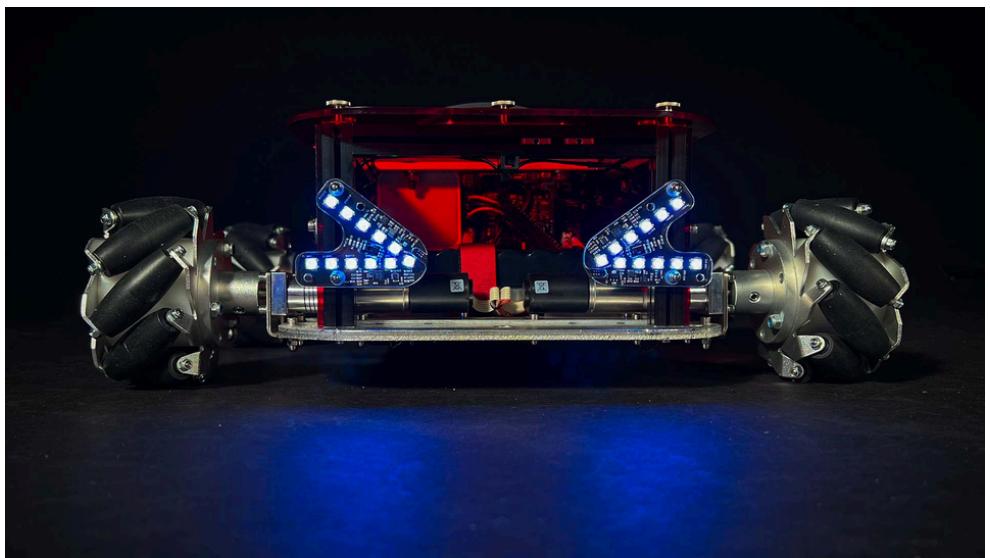
Vormittag:

- Übersicht über EduArt-Roboter
- Node-RED mit ROS Integration

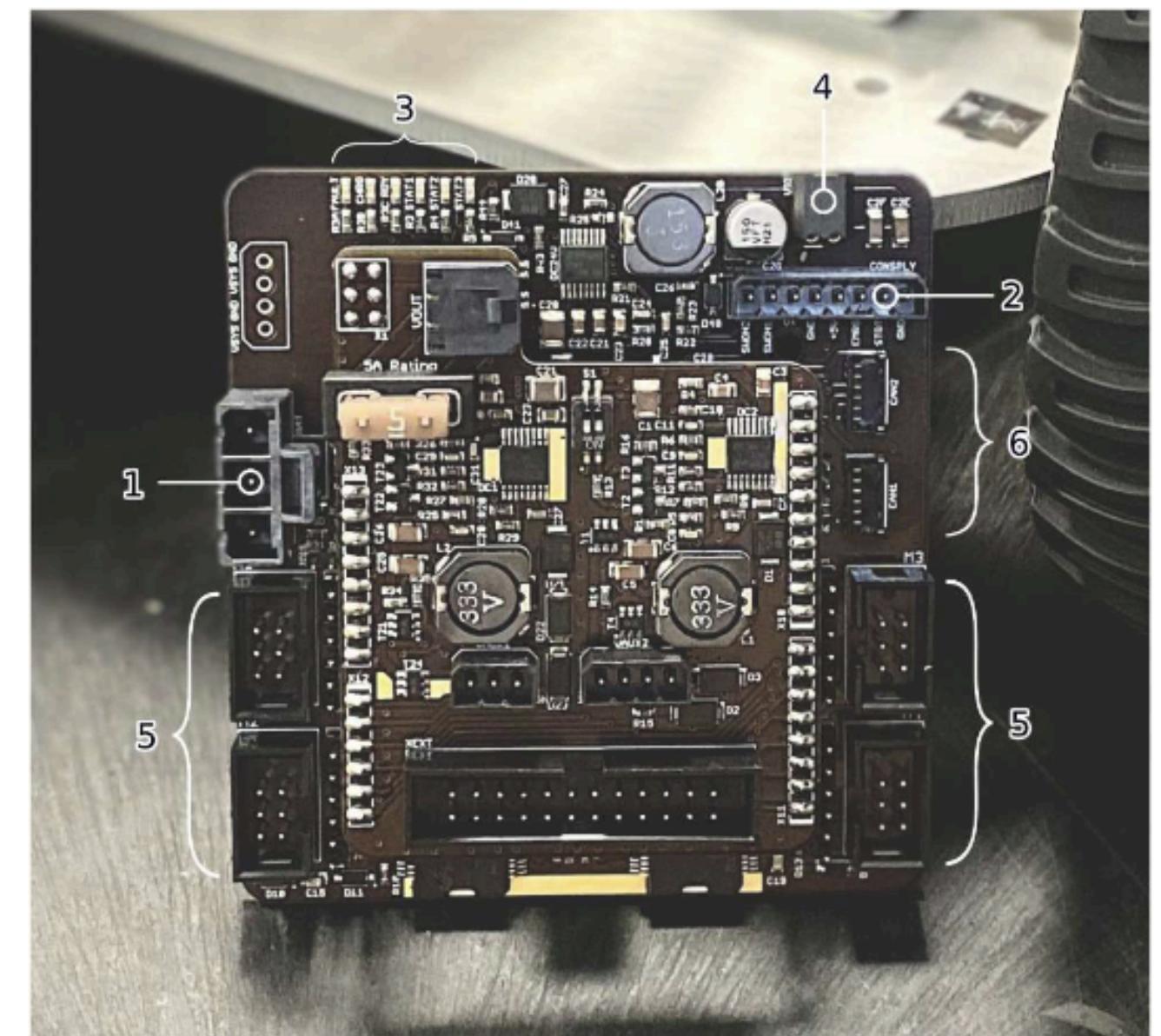
Nachmittag

- Verschiedene Aufgaben

# EduArt Roboter

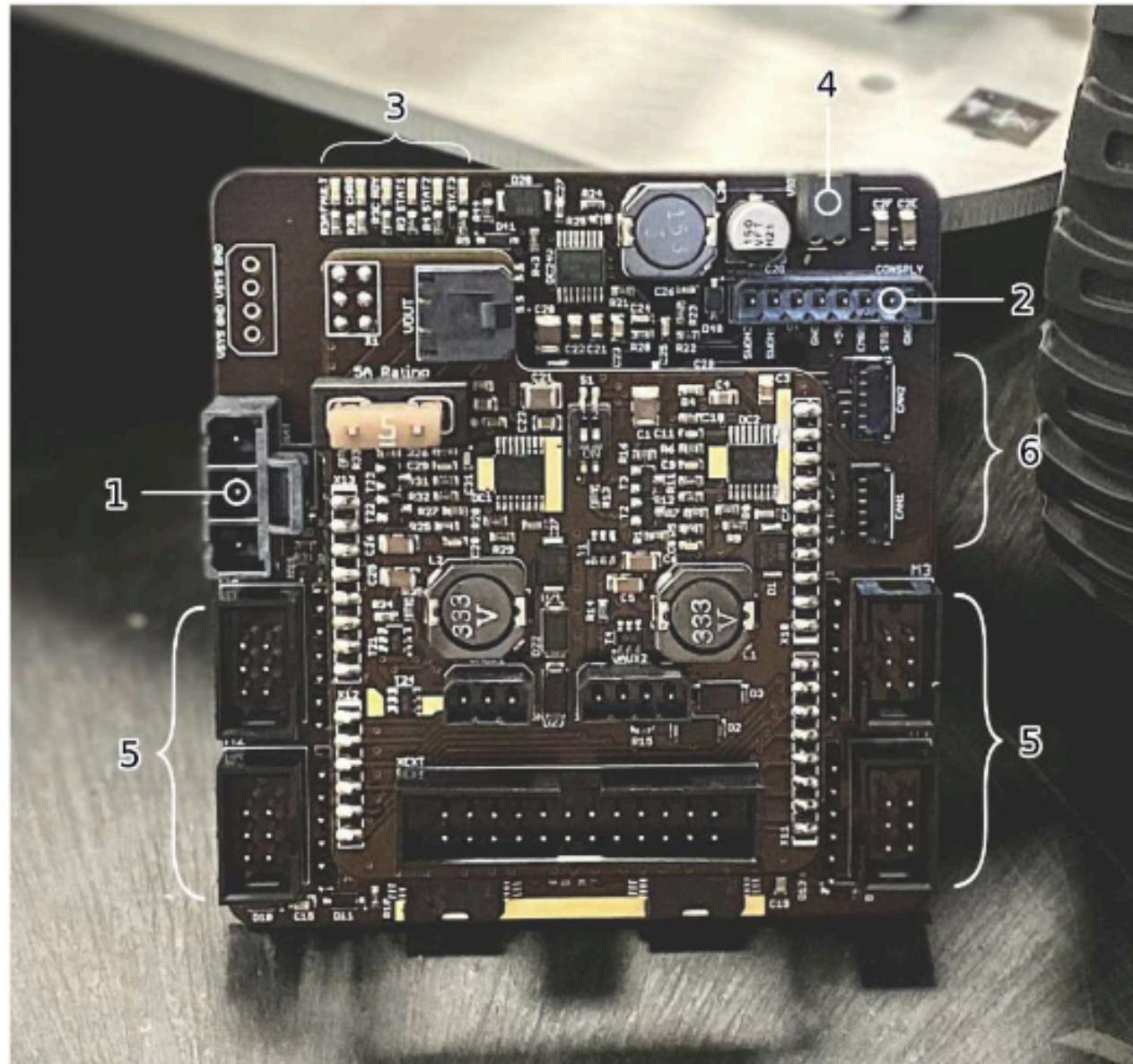


Robotic Extension Shield



# EduArt Roboter

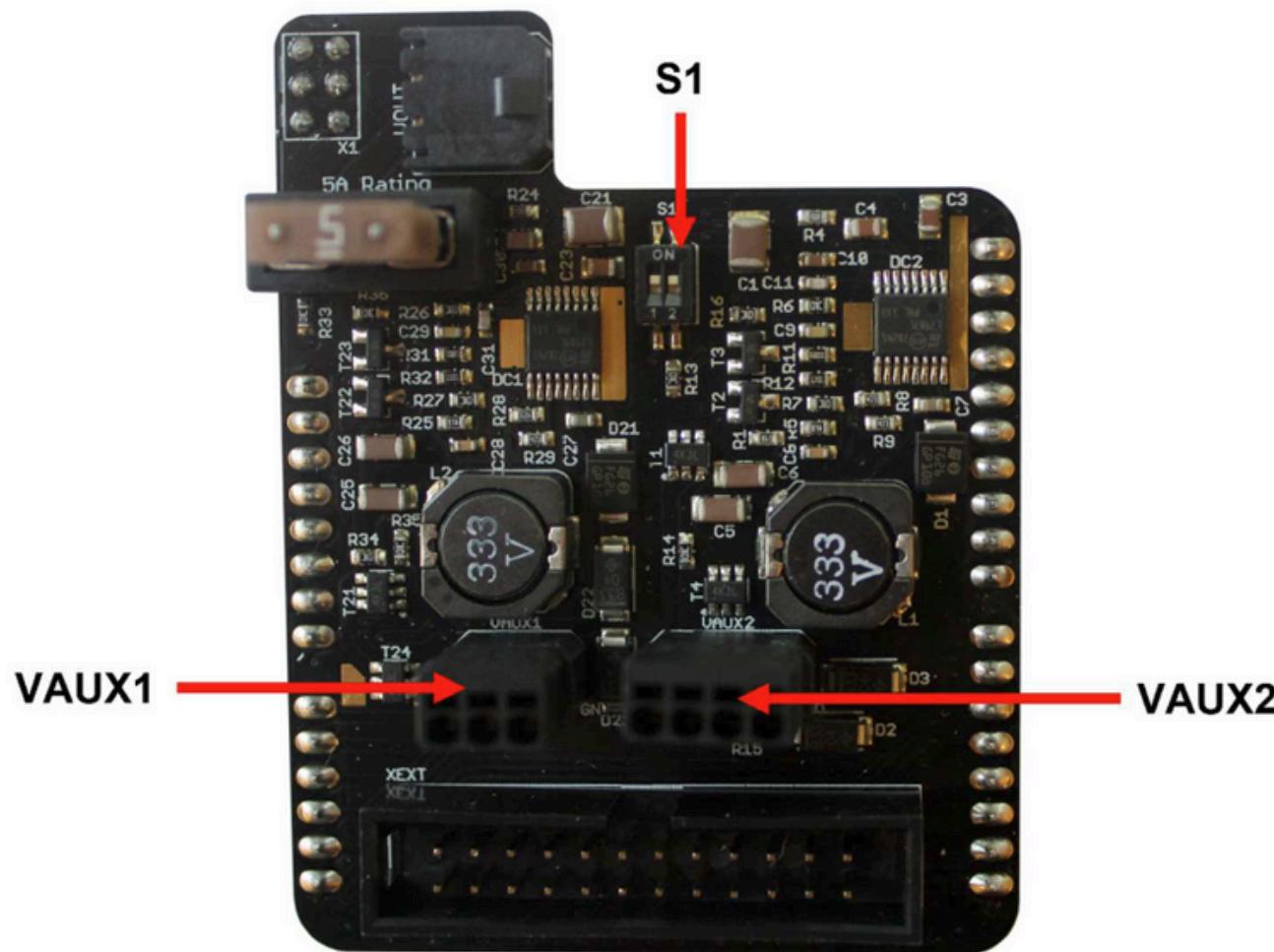
## Robotic Extension Shield



- Akkumanagementsystem (Laderegler, Unterspannungsschutz, Temperaturüberwachung während des Ladevorgangs)
- Systemspannungsmessung
- Inertiale Messeinheit mit 6 Freiheitsgraden
- Hauptsicherung
- Anschluss und Steuerung von vier bürstenkommutierten Gleichstrommotoren
- Arduino Uno Schnittstelle
- Anschluss von Bedienelementen (Ein-Taster, Ein-/Aus-Schalter, Haltebuchse)
- Versorgung des IOT2050

# EduArt Roboter

## Spannungswandler



Wandelt Spannung auf 19V, 12V und 5V um, da Motoren und das Extension Shield verschiedene Spannungen brauchen. Dieser Spannungswandler ist mit 5A abgesichert.

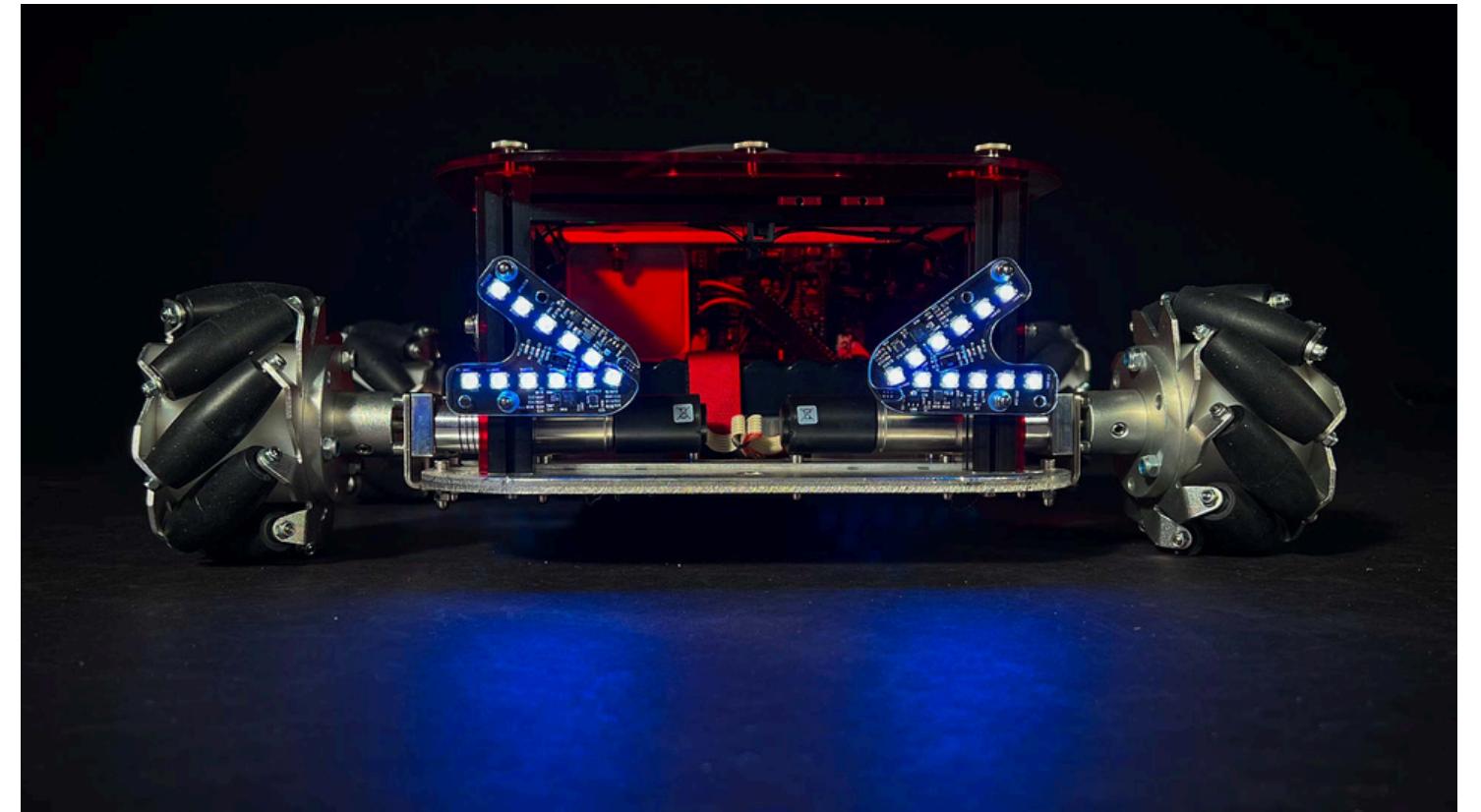
Abbildung 7.3: 2-Kanal-DC/DC-Spannungswandler

# Wie man sich mit Eduard verbindet

In einem beliebigen Terminal (Windows oder Ubuntu) eingeben:

```
ssh root@192.168.0.100
```

Passwort ist "**root**" (Man sieht nichts wenn man tippt)



# Wie kann ich mir Daten am Eduard anschauen?

1) Verbinden via ssh

2) Der Code läuft in sogennanten “Container”:

**docker ps**

3) Wir müssen in diesen Docker “springen”, in dem wir uns die Daten anschauen können, z.B.:

**docker exec -it eduard-iotbot-0.4.1 bash**

4) Dann müssen wir noch sourcen

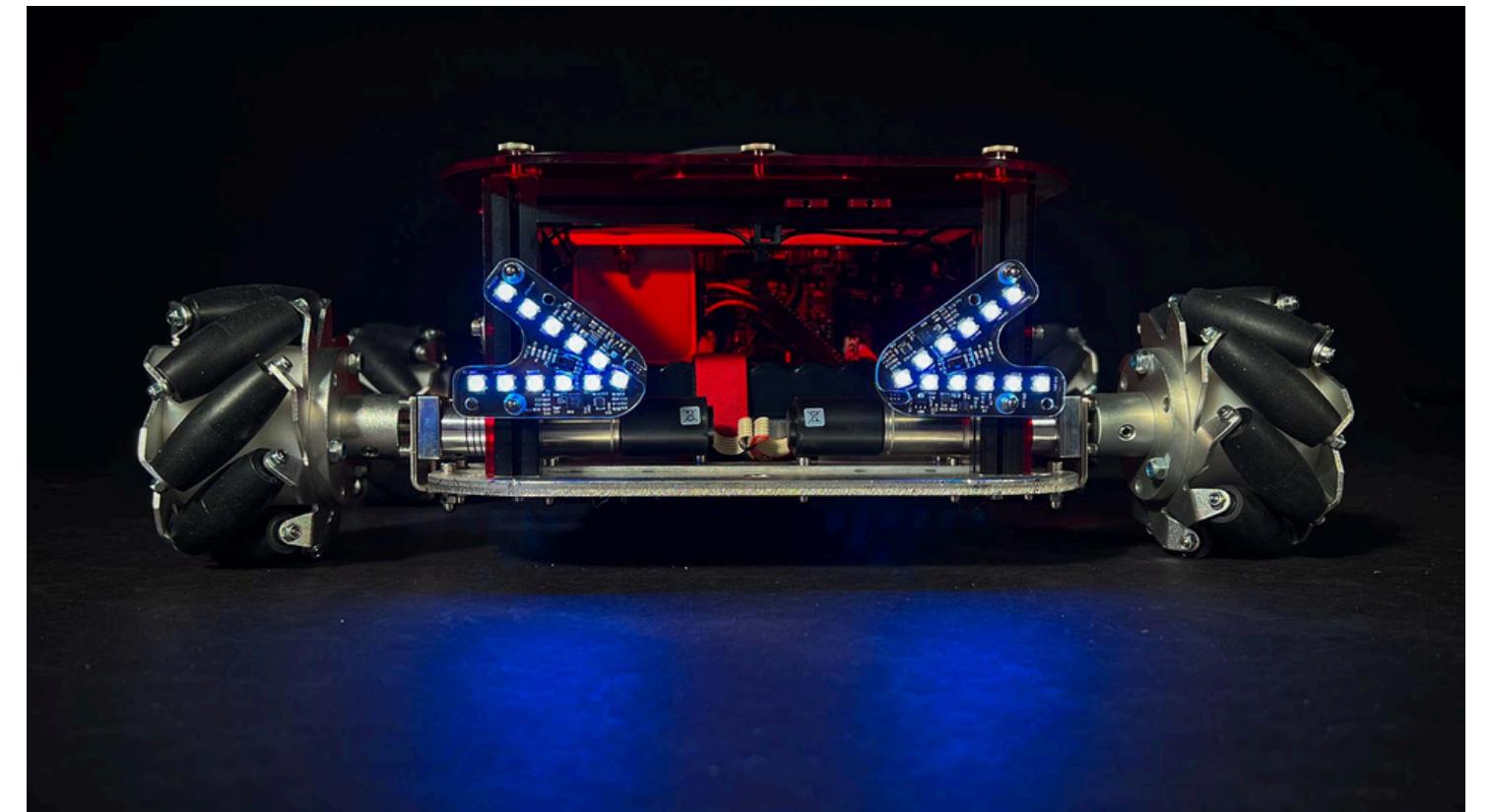
**source install/setup.bash**

5) Liste von verfügbaren topics

**ros2 topic list**

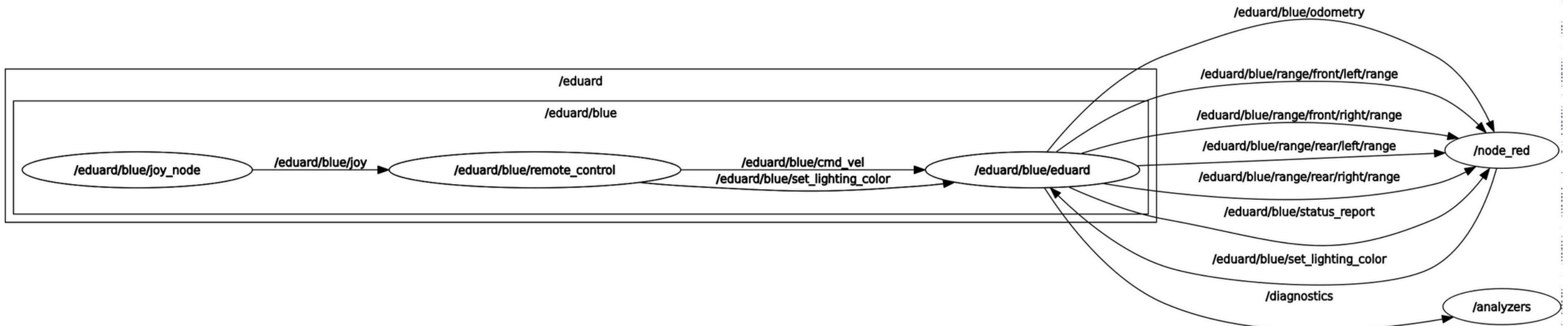
6) Ausgeben des Inhalts des Topics

**ros2 topic echo <topic\_name>**



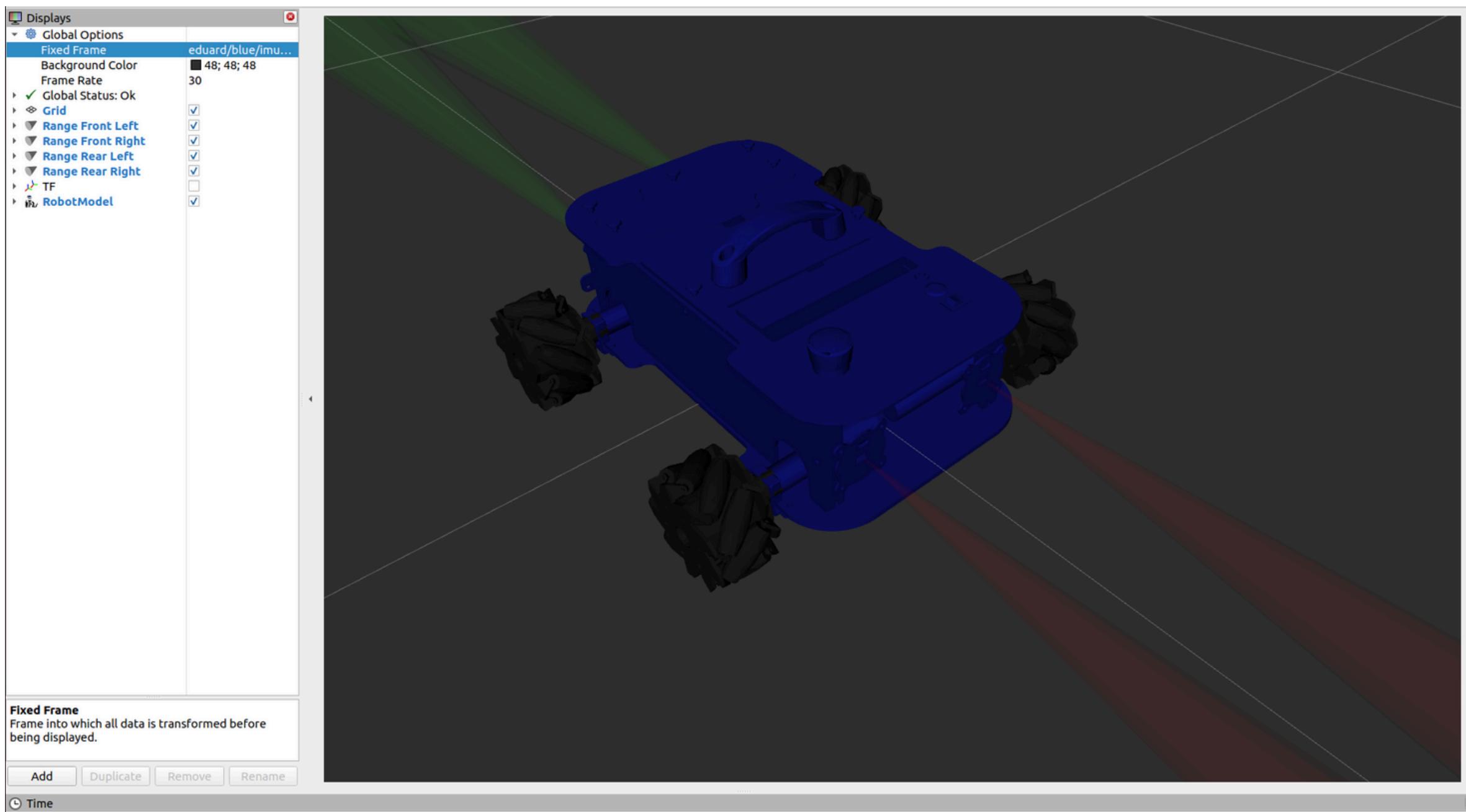
# Wie kann ich mir Daten am Eduard anschauen?

Mit nativen Ubuntu kann man auch das rqt-Tool verwenden:



# RVIZ

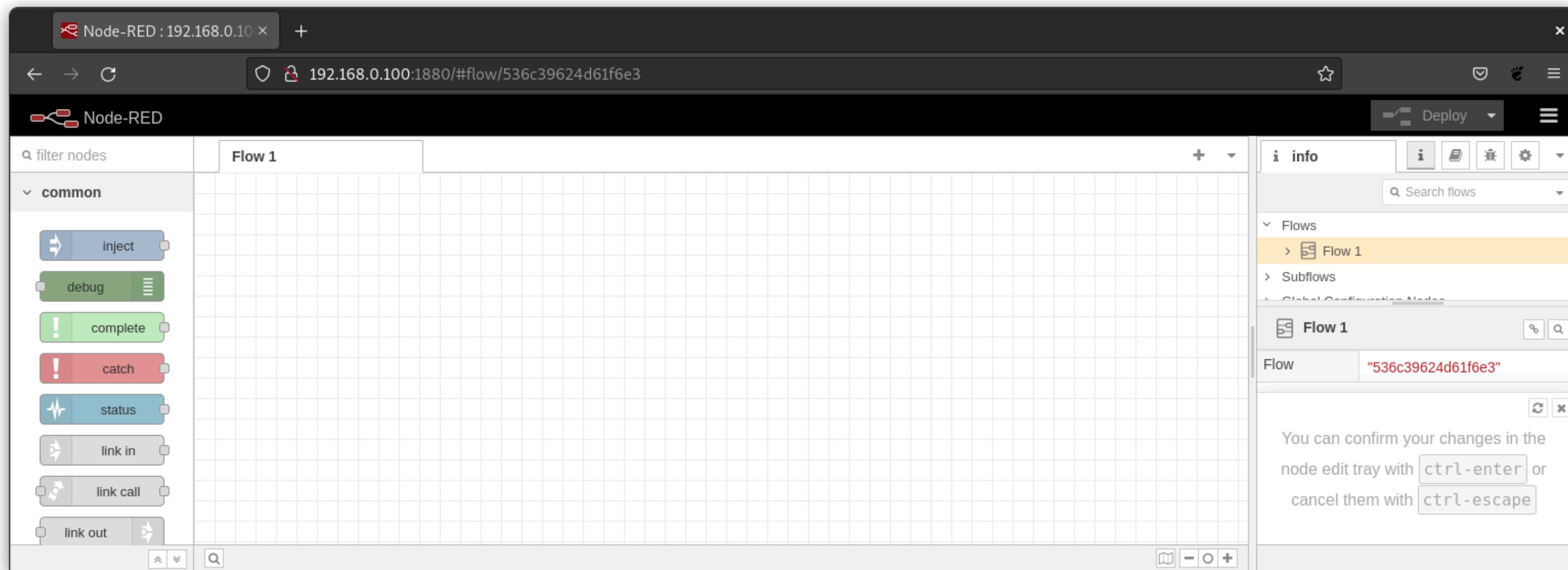
Mit nativen Ubuntu kann man auch das RVIZ-Tool verwenden:



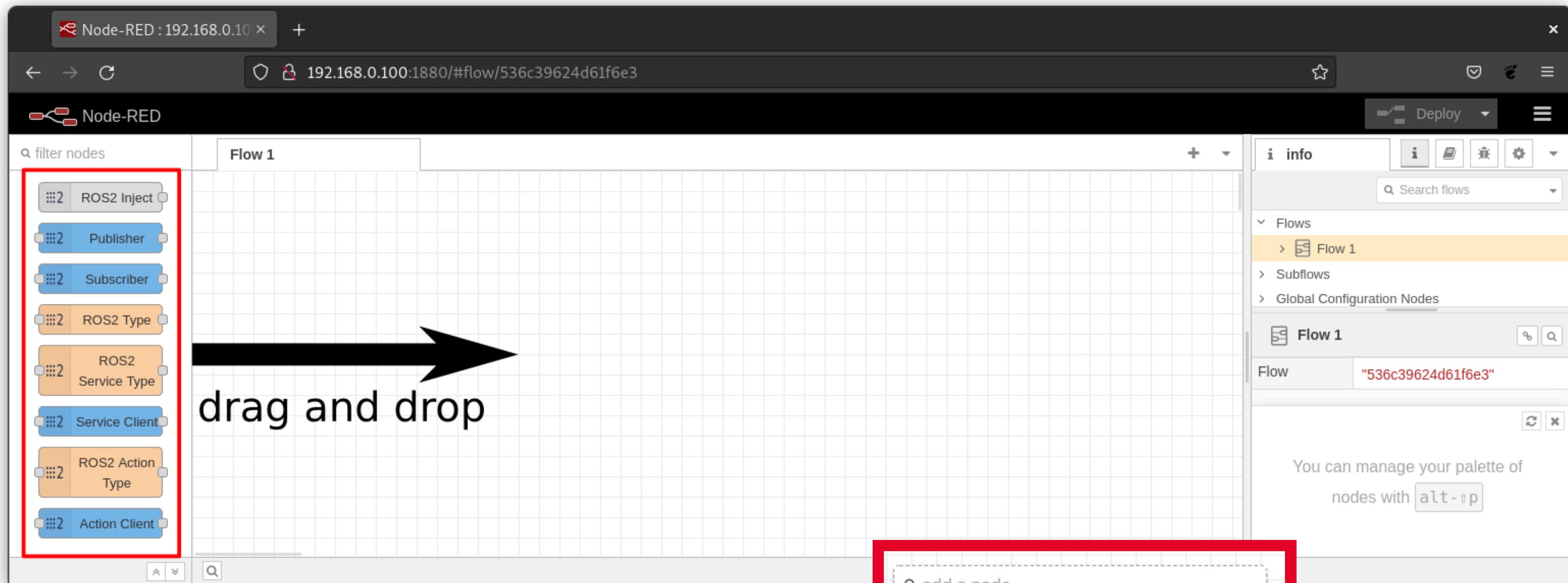
# Coffee Break!



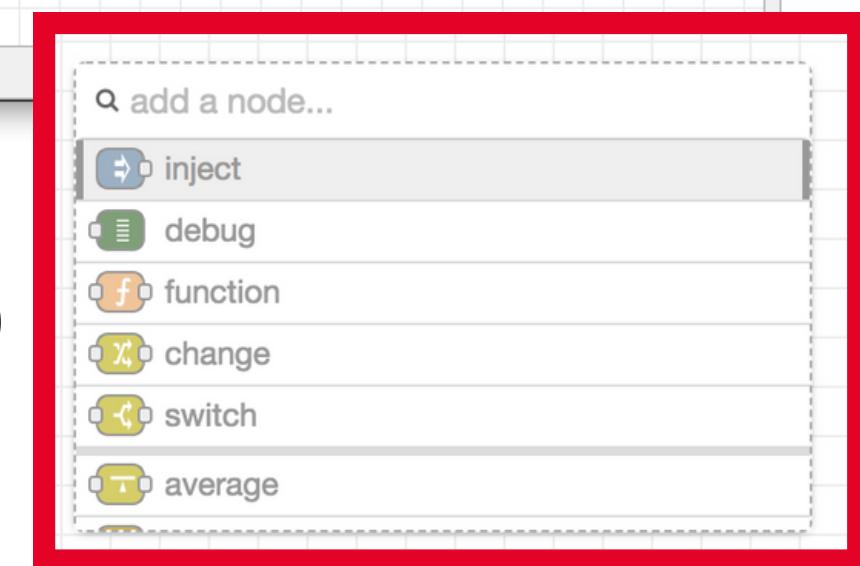
# Node-RED



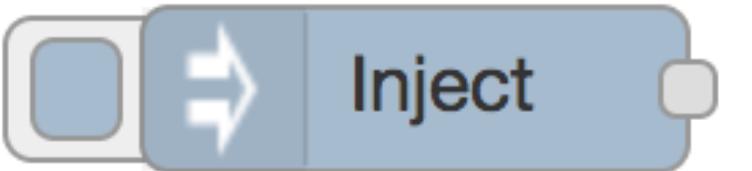
# Node-RED



Quick-Add Dialog (Ctrl + linke Maustaste)



# Node-RED



- ROS Topic
- “Payload”
  - flow
  - String, Boolean, Zahl, Buffer, Objekt, ...
  - Zeitstempel
- Man kann in Intervale Daten senden

## Java Script Types:

- Boolean - true, false
- Number - eg 0, 123.4
- String - "hello"
- Array - [1,2,3,4]
- Object - { "a": 1, "b": 2}
- Null

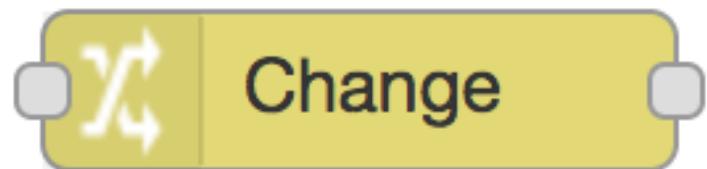
# Node-RED



- in Echtzeit Daten sehen
- Wenn nicht gebraucht wird, sollte man diese disablen.



- Funktionen werden hier in JavaScript geschrieben



- Daten umbenennen, Datentyp ändern, oder auch löschen

# Node-RED

Die Nodes sollten:

- eine klar definierte Aufgabe haben.
- einfach zu verwenden sein, unabhängig von der zugrunde liegenden Funktionalität.
- flexibel sein, welche Arten von Nachrichten-Eigenschaften sie akzeptieren.
- konsistent in dem sein, was sie senden.
- sich am Anfang, in der Mitte oder am Ende eines Flows befinden – aber nicht alles auf einmal.
- Fehler abfangen.

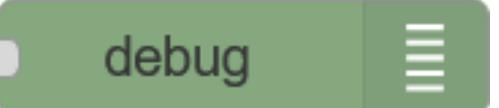
# Node-RED - Was ist ein Flow?

Ein Flow wird als ein Tab innerhalb des Editor-Arbeitsbereichs dargestellt und ist die Hauptmethode, um Nodes zu organisieren.

Der Begriff „Flow“ wird auch informell verwendet, um eine einzelne Gruppe von verbundenen Nodes zu beschreiben. Ein Flow (Tab) kann also mehrere Flows (Gruppen von verbundenen Nodes) enthalten.

# Aufgabe: Abstände messen mit Monitor

Bausteine die wir brauchen:

-  Damit legen wir den Typ des Topics fest.
-  Damit legen wir den Namen des Topics fest.
-  Damit lassen wir uns den Abstand anzeigen.
-  Optional: Debugger um Werte anzeigen zu lassen

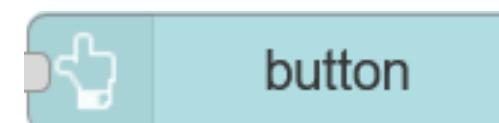
Aufgabe: Finde das Topic, in dem Spannung und Stromstärke gepublisiert werden und füge sie zu unserem Monitor hinzu!

# Aufgabe: Turn on the lights!

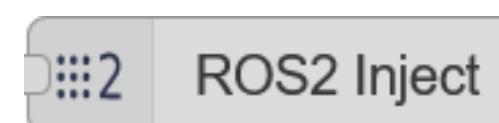
Wir wollen jetzt die LED-Lichter des Roboters mittels eigener Knöpfe kontrollieren. Bausteine die wir dafür brauchen:



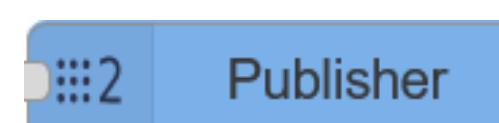
Damit legen wir den Typ des Topics fest.



Damit lösen wir eine bestimmte Aktion aus.



Damit senden wir einen Wert an das Topic.



Damit erzeugen wir einen ROS-Publisher



Optional: Man kann auch einen Sound ausgeben lassen

[https://github.com/EduArt-Robotik/edu\\_robot/blob/main/msg/SetLightingColor.msg](https://github.com/EduArt-Robotik/edu_robot/blob/main/msg/SetLightingColor.msg)



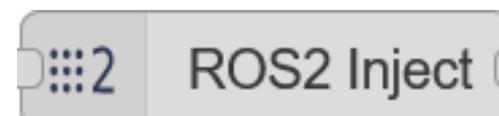
# Aufgabe: Implement your own Joystick!

Quiz: Was müssen wir jedes Mal machen, damit wir fahren können?

# Aufgabe: Implement your own Joystick!

Antwort: Wir müssen den richtigen Wert für den Modus setzen, sodass wir überhaupt erst losfahren können:

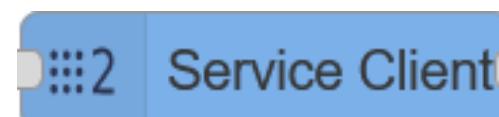
[https://github.com/EduArt-Robotik/edu\\_robot/blob/main/msg/Mode.msg](https://github.com/EduArt-Robotik/edu_robot/blob/main/msg/Mode.msg)



Damit senden wir einen Wert an das Topic.



Damit legen wir den Typ des Topics fest.



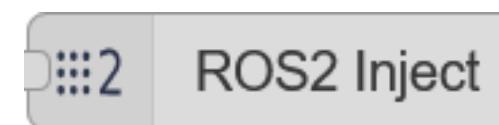
Diesmal brauche wir einen Service-Client (kurze Kommunikation)



Optional: Debugger um Werte anzeigen zu lassen

# Aufgabe: Implement your own Joystick!

Anschließend platzieren wir die weiteren Elemente, die wir für unseren Joystick brauchen:



Damit senden wir einen Wert an das Topic.



Custom Button (gegeben)



Damit legen wir den Typ des Topics fest.



Um ein eindeutiges Signal zu schicken (true = Fahren, false = nicht fahren)



Damit erzeugen wir einen ROS-Publisher

# Aufgabe: Don't hit the wall!

Für diese Aufgabe lassen wir den Roboter in eine beliebige Richtung fahren. Sobald ein Hinderniss erscheint, sollen wir rechtzeitig stoppen. Nutze die bisher verwendeten Bausteine. Um die Logik zu modellieren, mit der wir den Roboter zum Stoppen bringen, brauchen wir den "function"-Block:



```
// storing input values
if (msg.payload["range"] != undefined) {
    // if range is defined than input is a range measurement
    context.set("range", msg.payload["range"]);
}

else if (msg.payload["linear"] != undefined) {
    // if linear is defined than input is a twist message.
    context.set("twist", msg.payload);
}

// checking if distance is to close
if (context.get("range") < 0.2) {
    // override received twist command
    let twist = context.get("twist");

    twist.linear.x = 0.0;
    twist.linear.y = 0.0;
    twist.angular.z = 0.0;

    context.set("twist", twist);
}

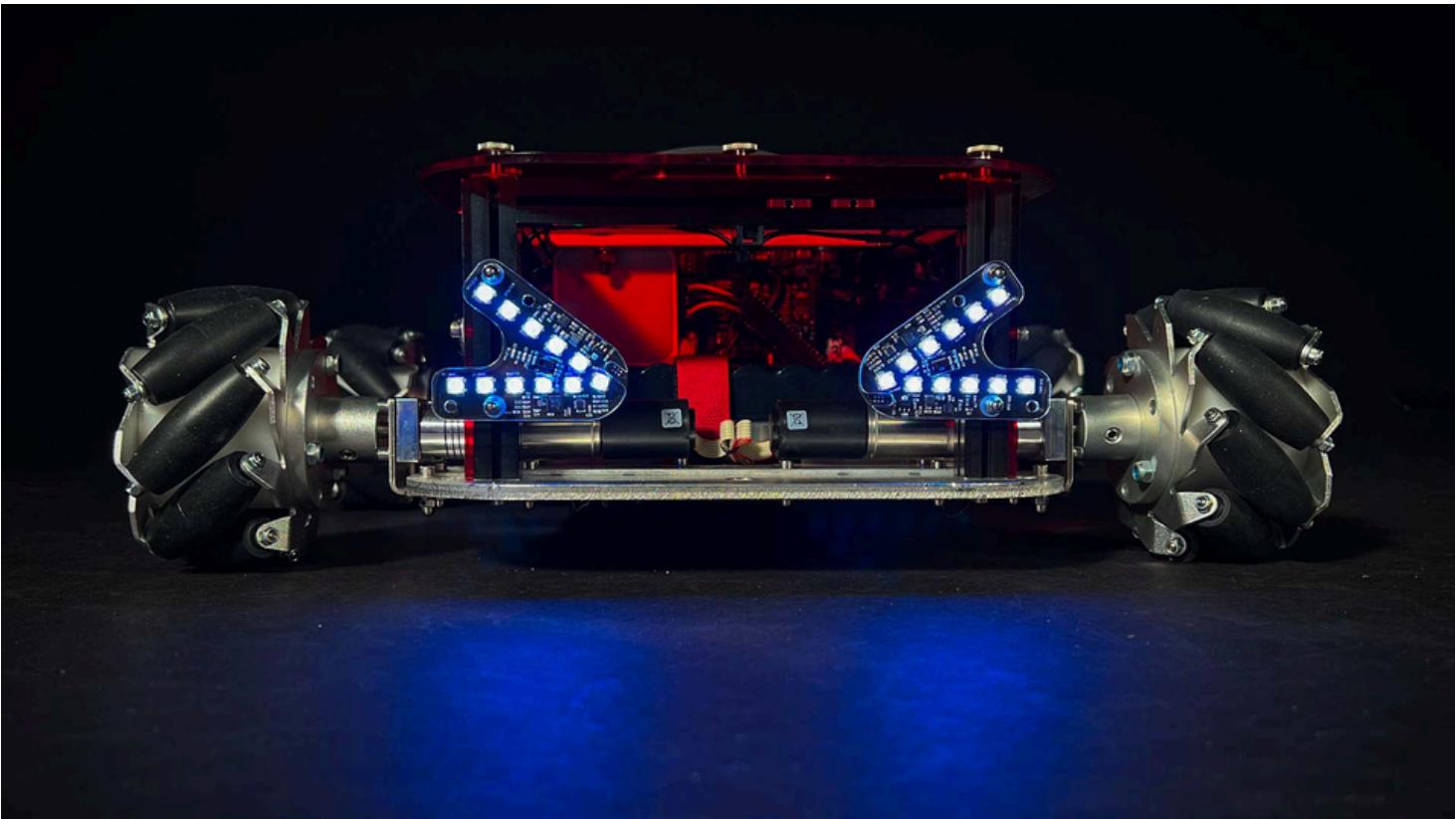
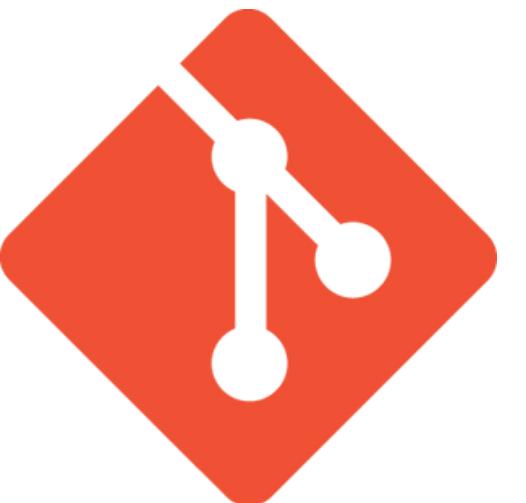
// publish twist command
msg = { payload: context.get("twist") };

return msg;
```

# Happy Feierabend!

Bis morgen um 9:00!

Dann geht's weiter mit:



# **Software and Robotics Workshop**

C++ and Robotics Experience

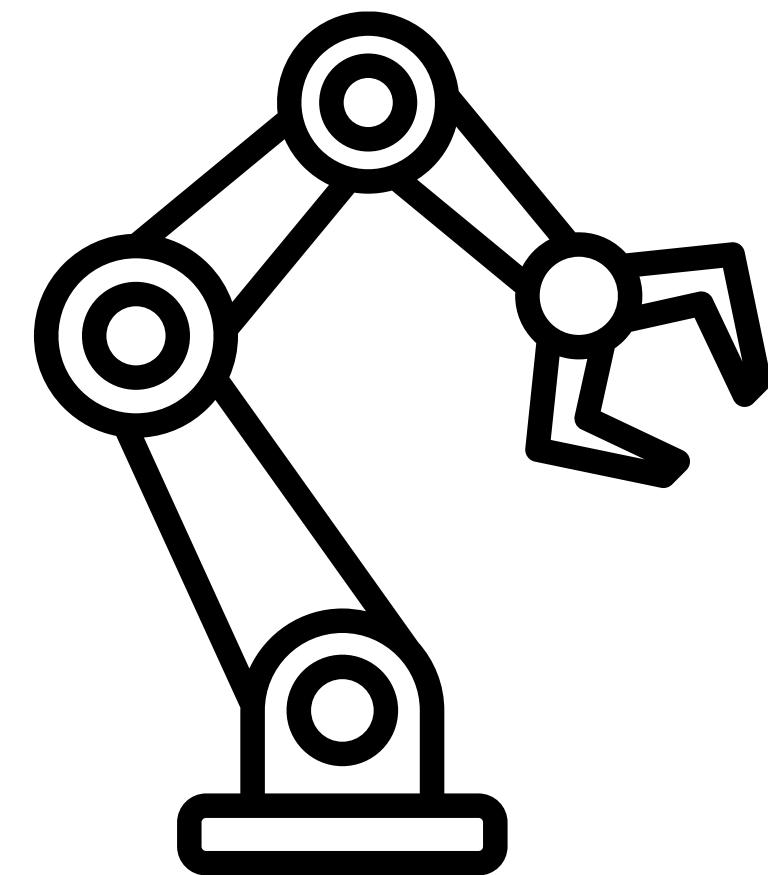
September 2024

Tag 3



**MAKERSPACE**

**AIRBUS**



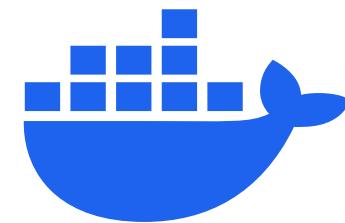
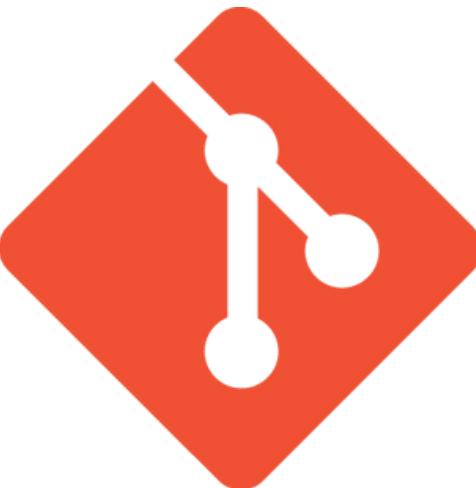
# Tag 3: Git, Navigation Challenge

Vormittag:

- Git
- Docker

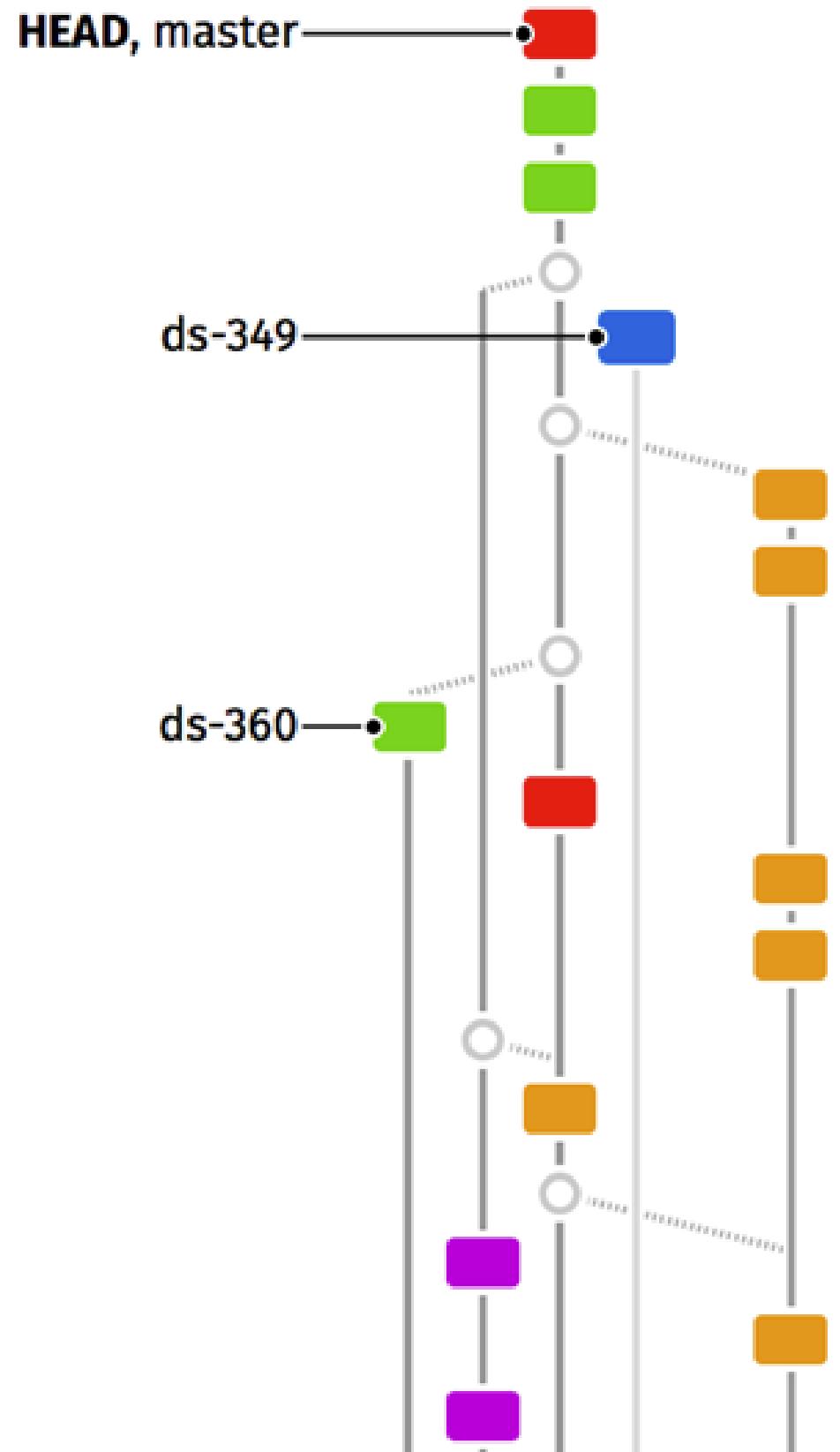
Nachmittag

- Navigation Challenge



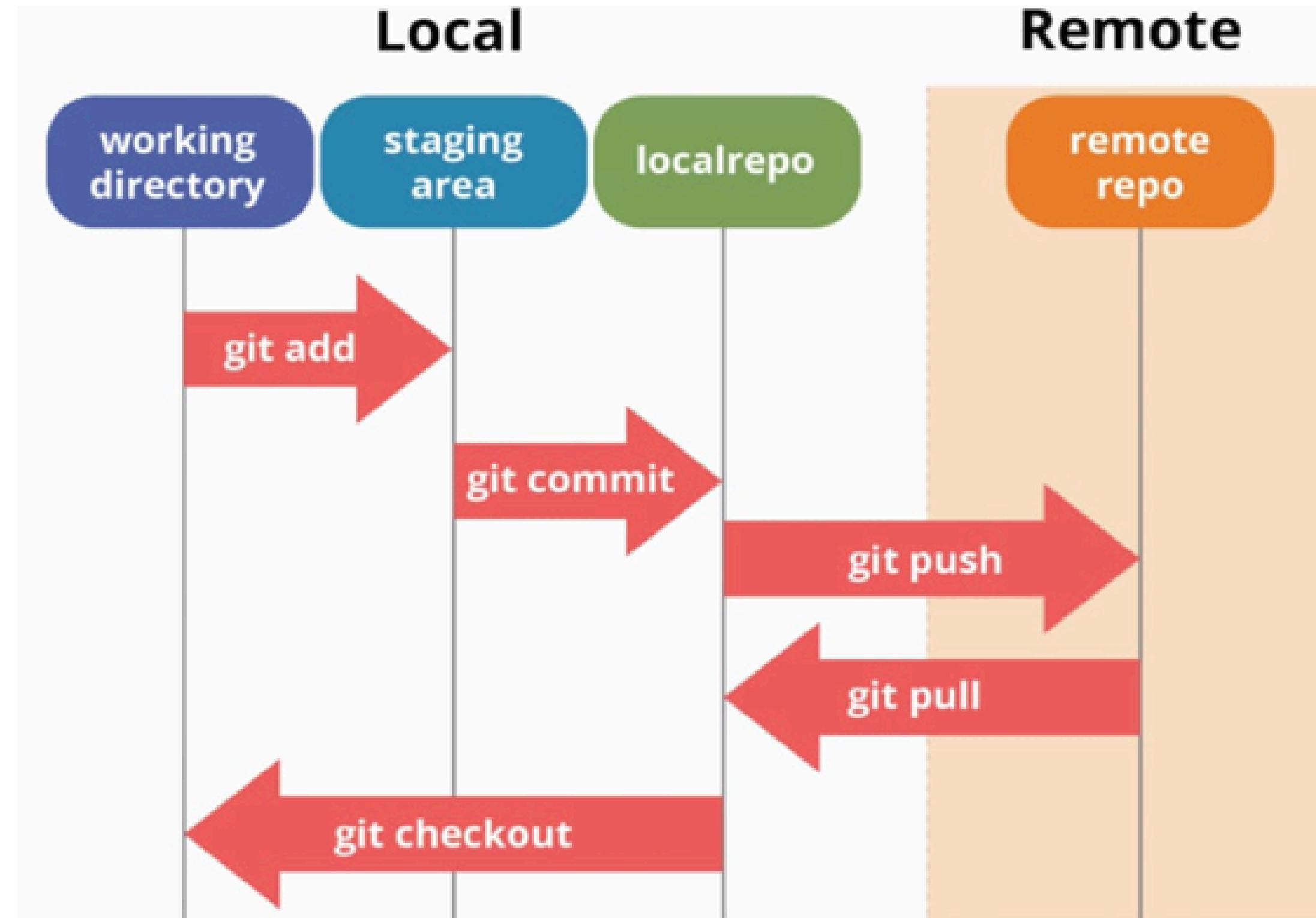
# Git: Why and how?

Source Code Management und Versionierung sind ein wichtiges Werkzeug bei der Software-Entwicklung.



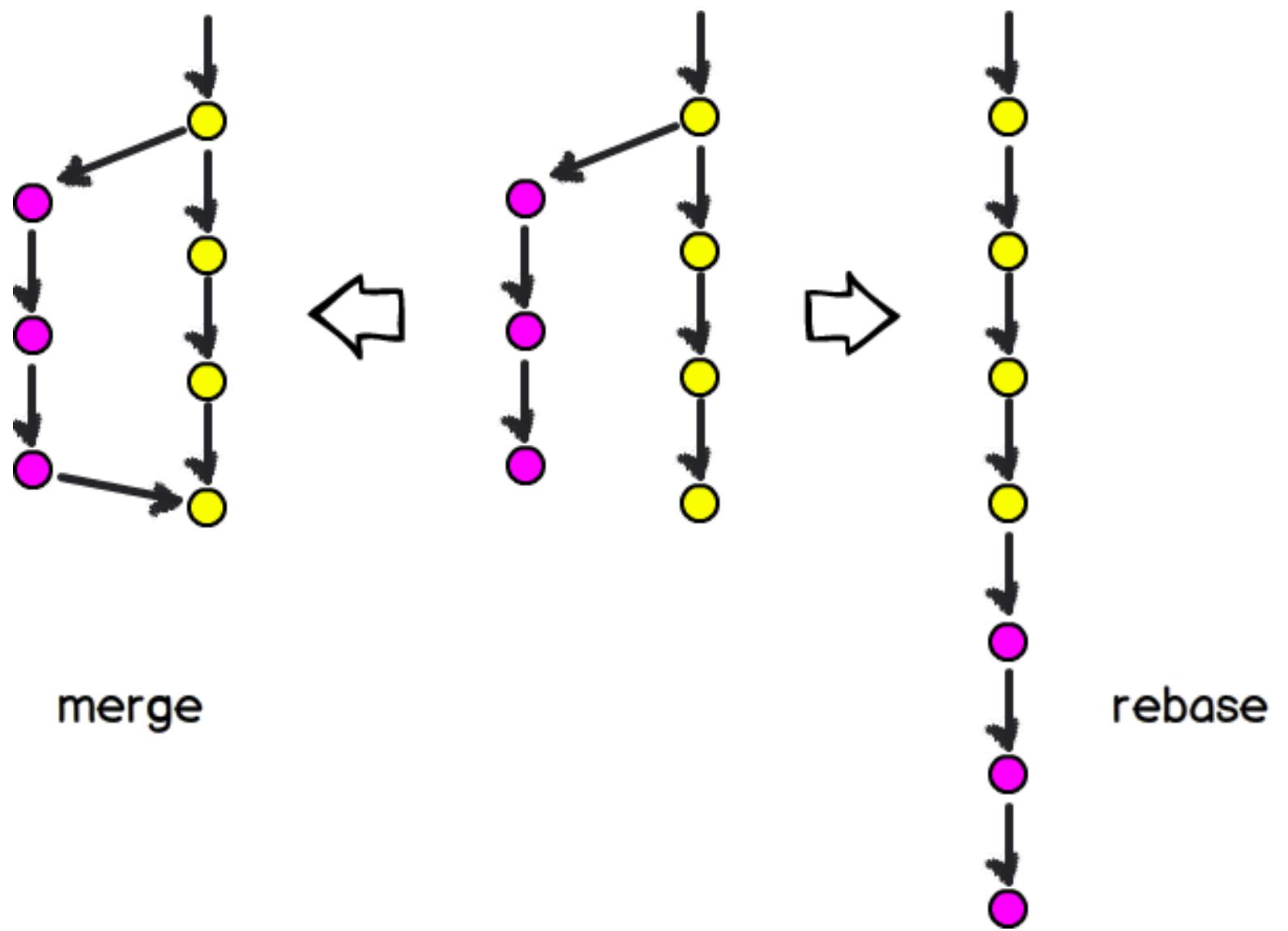
# Git: Grundbegriffe

Github/Gitlab



# Git: Merge & Rebase

- Bei einem **Merge** werden zwei unterschiedliche Historien verglichen und ein Commit erzeugt, der beide auf einen gemeinsamen Stand bringt.
- Ein **Merge** kommt vor, wenn man an unterschiedlichen Features in einem Projekt arbeitet, und diese voneinander Unabhängig auf einer Funktionierenden Version basieren sollen.
- Ein **Rebase** wendet alle Commit-Diffs auf basis des anderen commits an, und verändert dadurch die Identifier aller lila commits.



# Git: Merge Konflikte

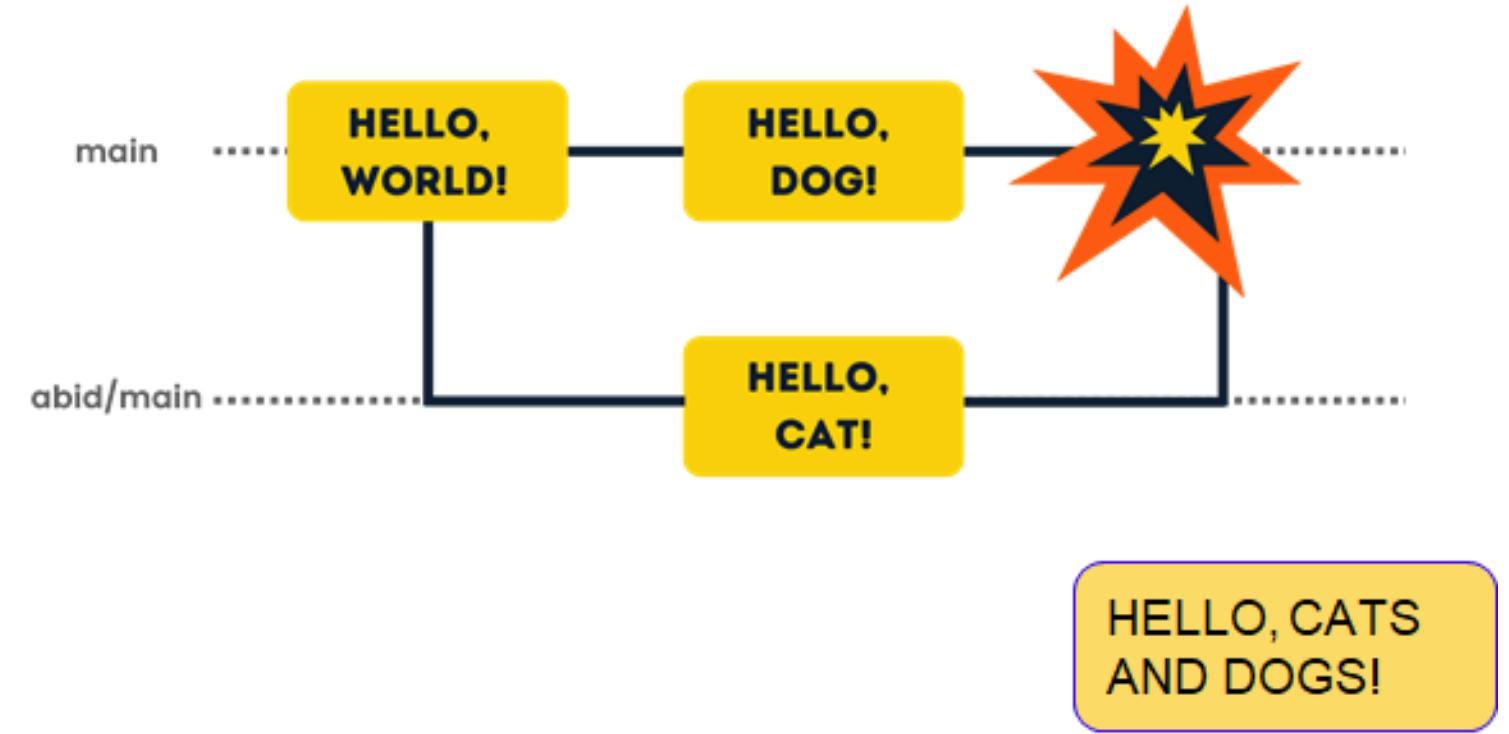
- Entstehen immer dann, wenn zwei Änderungen an der selben Datei passiert sind.
- Bei trivialen Konflikten gab es Änderungen in unterschiedlichen Teilen der Datei, +- ein paar Zeilen

für Kontext

- Bei echten Konflikten, wurde die gleiche Zeile geändert.

Dann muss in einem Three-Way-Merge entschieden werden, welche Version die Richtige ist oder eine ganz neue die beide Änderungen betrachtet.

- Die Empfehlung eines Grafischen Tools wird empfohlen (vimdiff, kdiff3, VSCode, ...)



# Git: Merge Konflikte

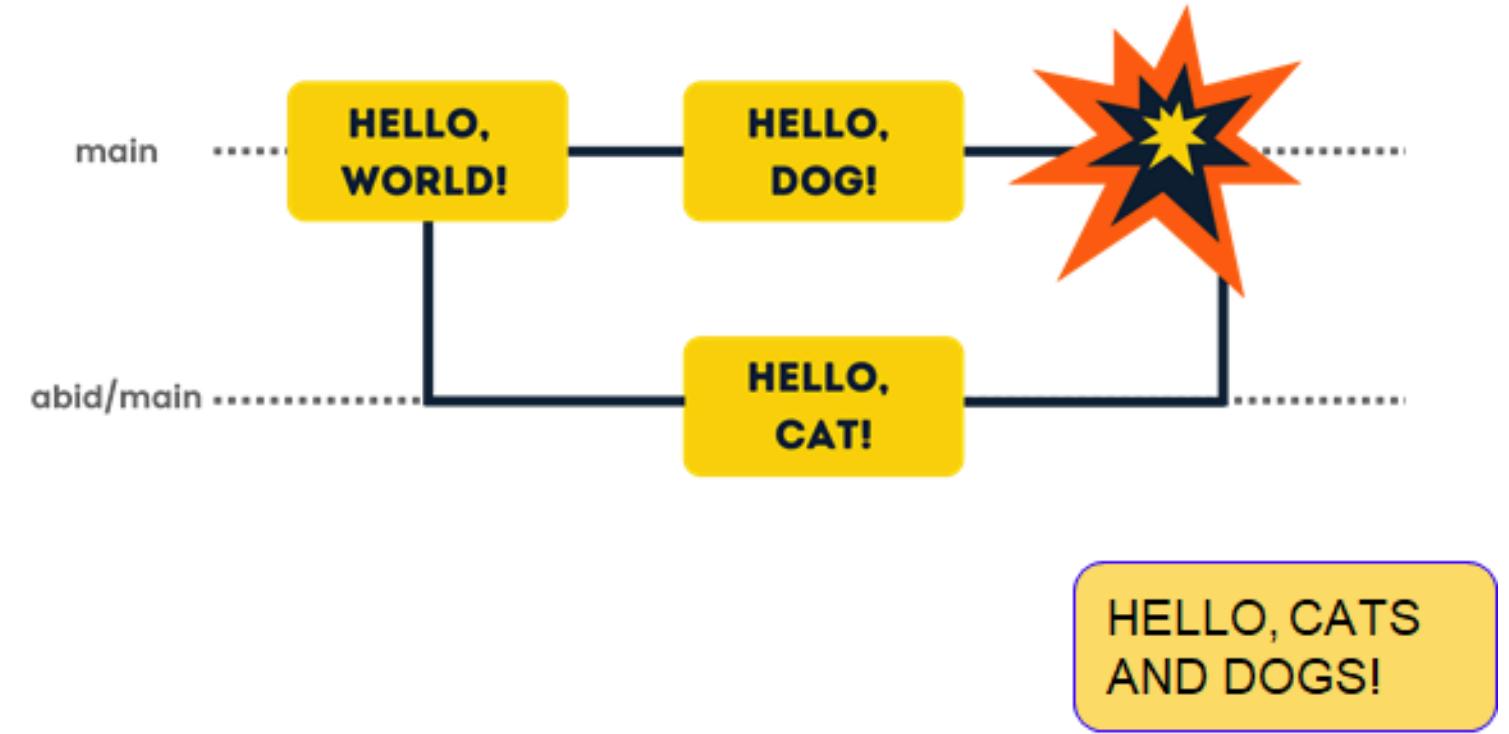
- Entstehen immer dann, wenn zwei Änderungen an der selben Datei passiert sind.
- Bei trivialen Konflikten gab es Änderungen in unterschiedlichen Teilen der Datei, +- ein paar Zeilen

für Kontext

- Bei echten Konflikten, wurde die gleiche Zeile geändert.

Dann muss in einem Three-Way-Merge entschieden werden,  
welche Version die Richtige ist oder eine ganz neue die beide Änderungen betrachtet.

- Die Empfehlung eines Grafischen Tools wird empfohlen (vimdiff, kdiff3, VSCode, ...)



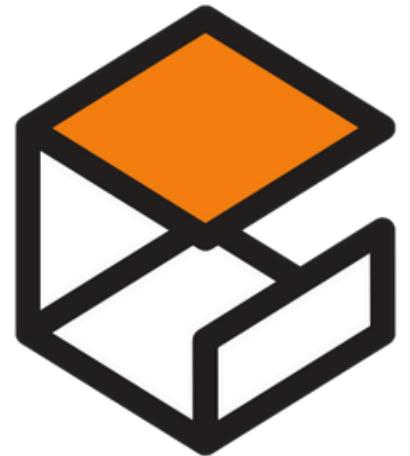
# Coffee Break!



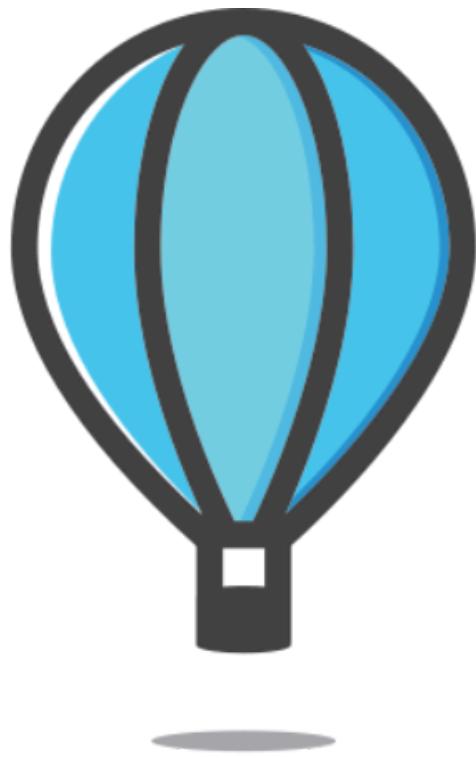
# Happy Feierabend!

Bis morgen um 9:00!

Dann geht's weiter mit:



GAZEBO

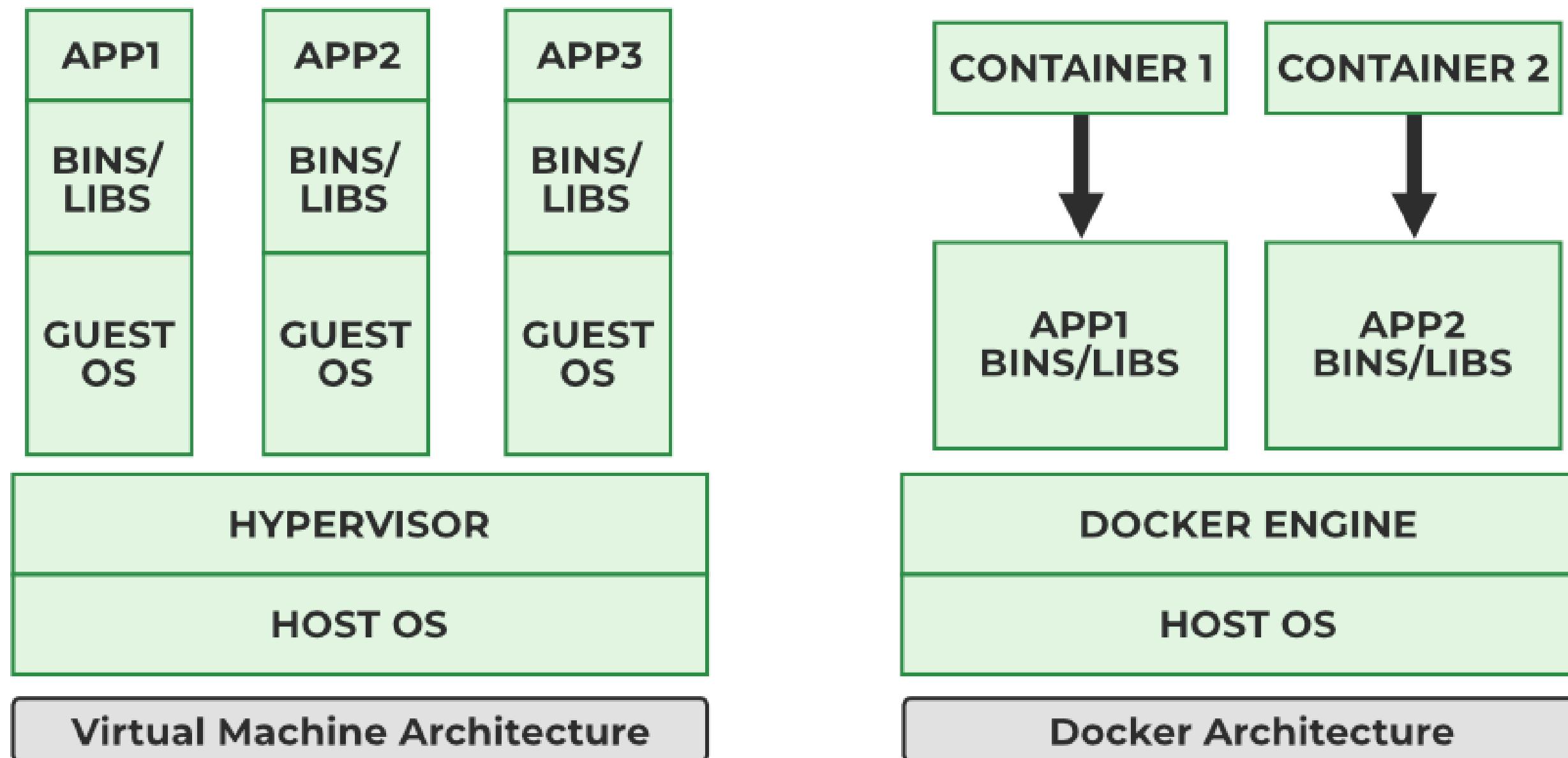


N A V 2



OPEN  
NAVIGATION

# Docker - What, Why and How

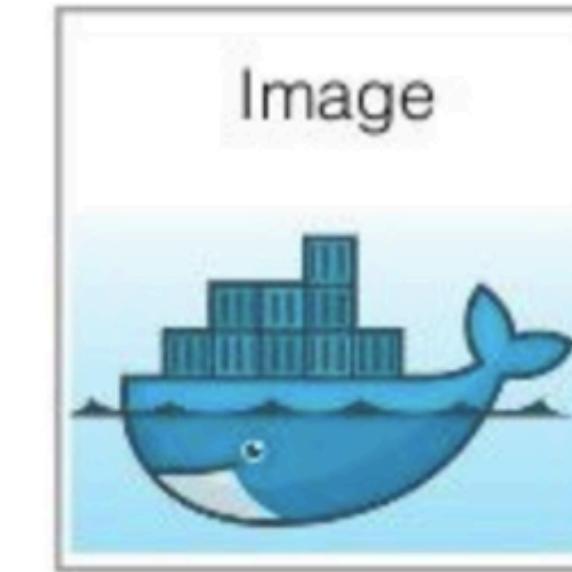


# Docker - Drei wichtige Begriffe



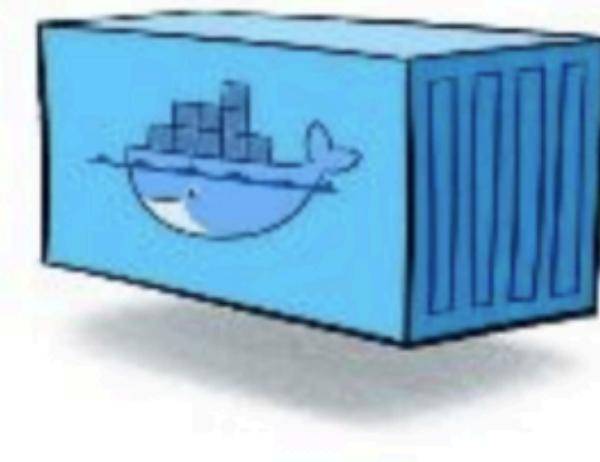
Dockerfile

build



Docker Image

run

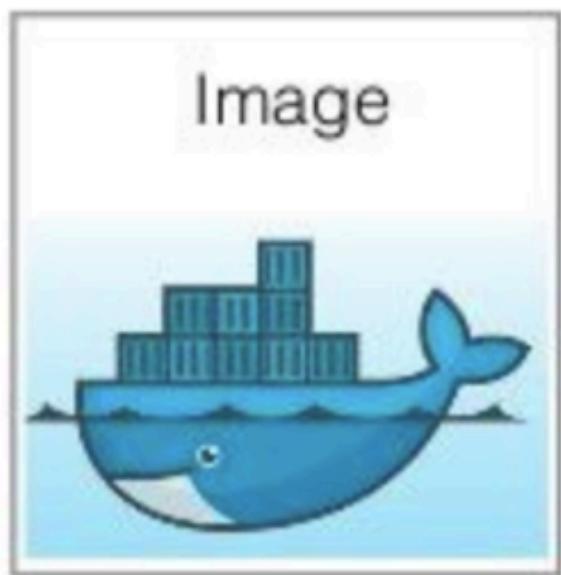


Docker Container



# Docker - Wie erstellen wir unser Dockerfile?

# Existierendes Docker Image Z.B. ROS Image



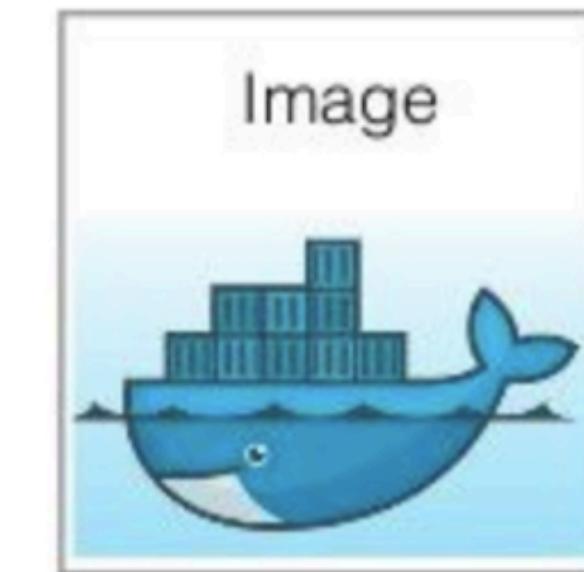
## Docker Image

# Unser eigenes Dockerfile



## Dockerfile

Neues, eigenes,  
spezifisches  
Docker Image



## Docker Image

# Docker - But why???

- inkompatible Software verbinden (z.B. ROS- Versionen brauchen immer bestimmte Ubuntu Versionen)
- Standardisiertes Testen und Anwenden von Software
- Standardisierte Development Umgebung
- Anwenden der Software ist viel simpler
- Code und Wissen schützen -> Nur das Image weitergeben

# Docker - Wichtige Befehle

docker ...

image ls

container run

container start

image pull

container ls

container stop

image rm

container rm

container exec

container prune

# Docker - Wichtige Befehle

docker ...

image ls

container run

container start

image pull

container ls

container stop

image rm

container rm

container exec

image build

container prune

# **Software and Robotics Workshop**

C++ and Robotics Experience

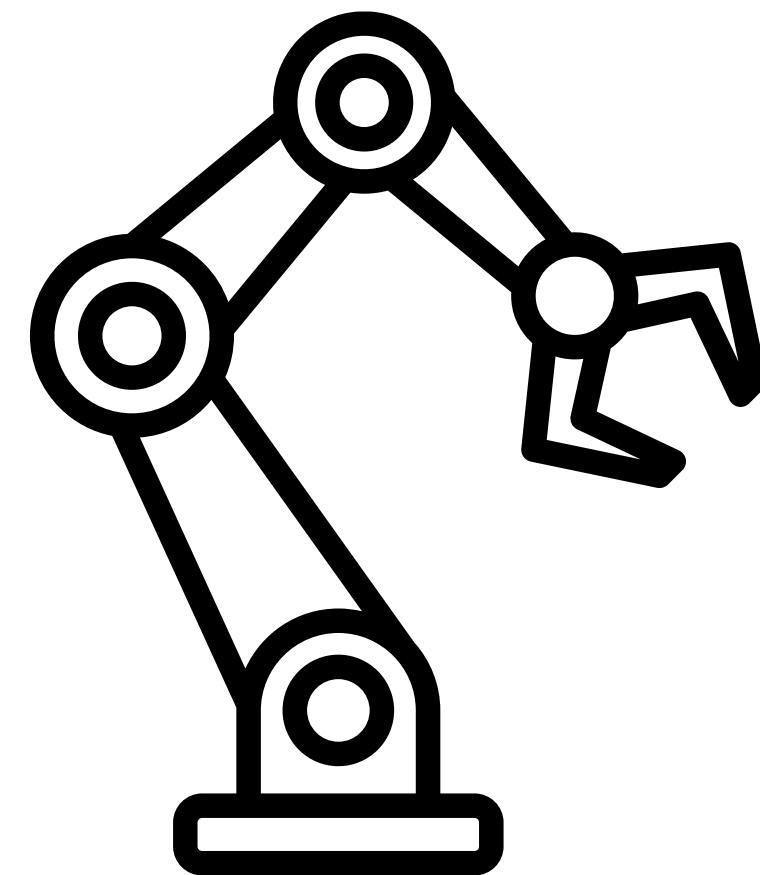
September 2024

**Tag 4**



**MAKERSPACE**

**AIRBUS**



# Tag 4: Nav2 und Goal Point Publisher

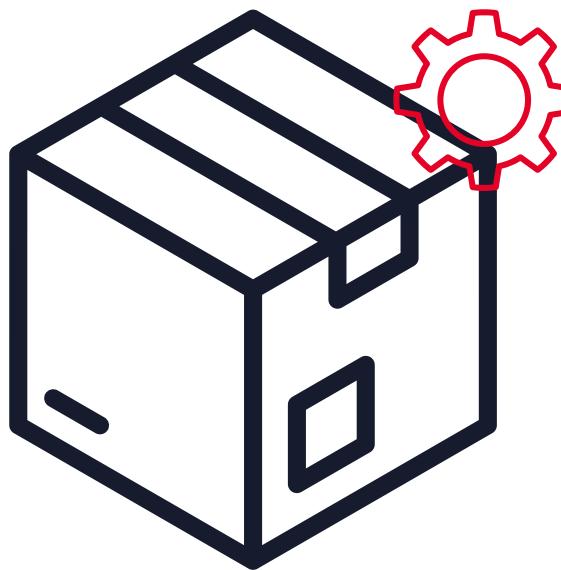
Vormittag:

- Nav2

Nachmittag

- Goal Point Publisher

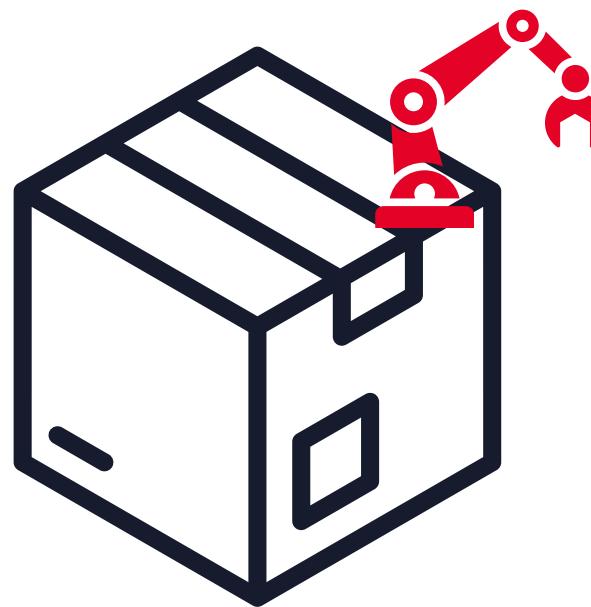
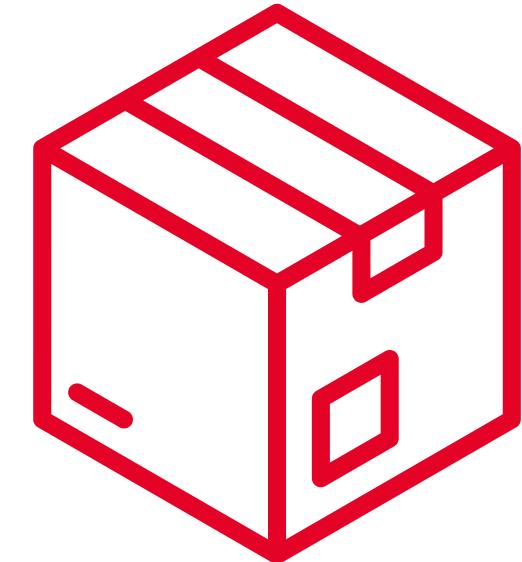
# Open source packages



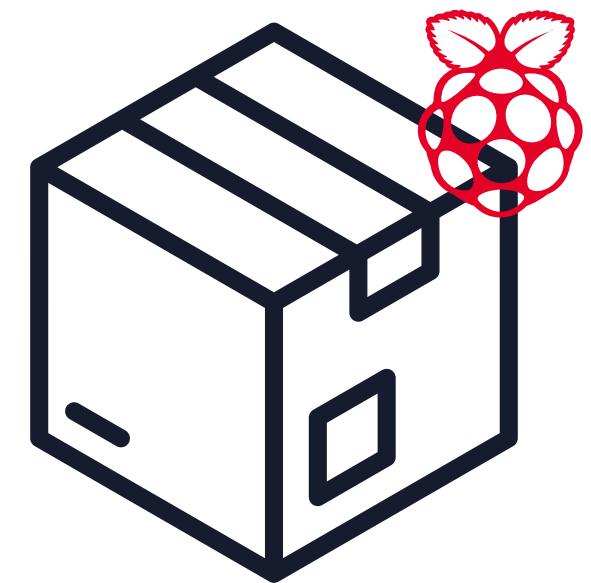
ros2\_control



Navigation2

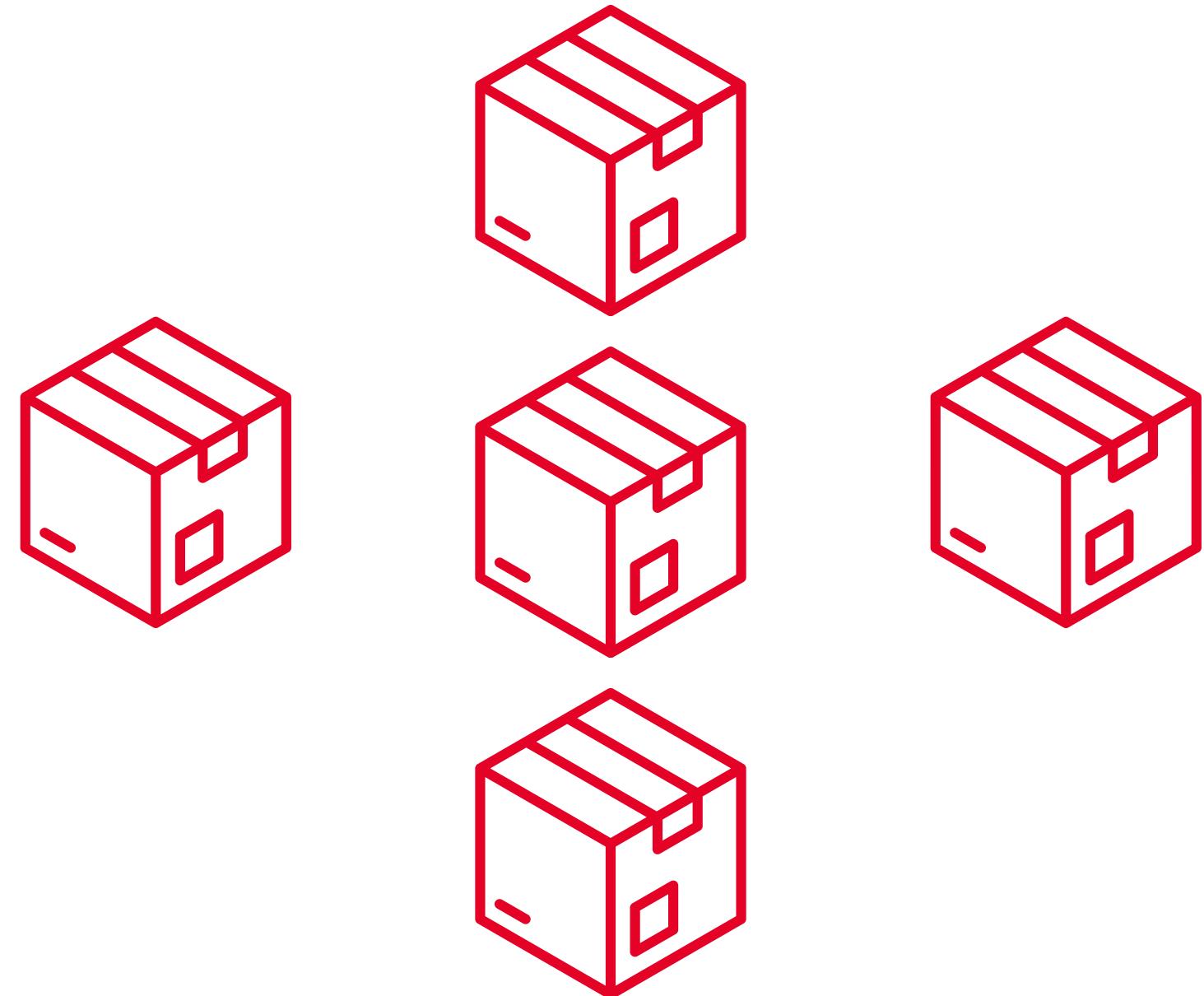


Movelt



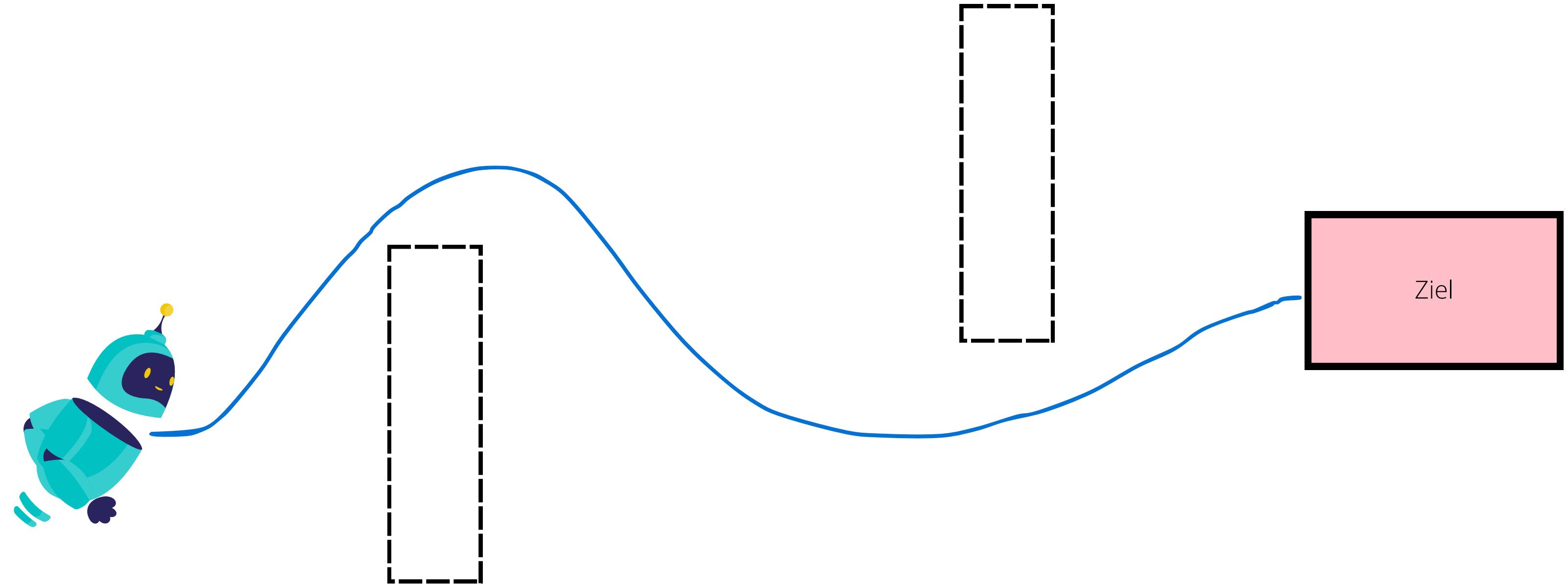
micro-ROS

# ROS2 Navigation2 Stack



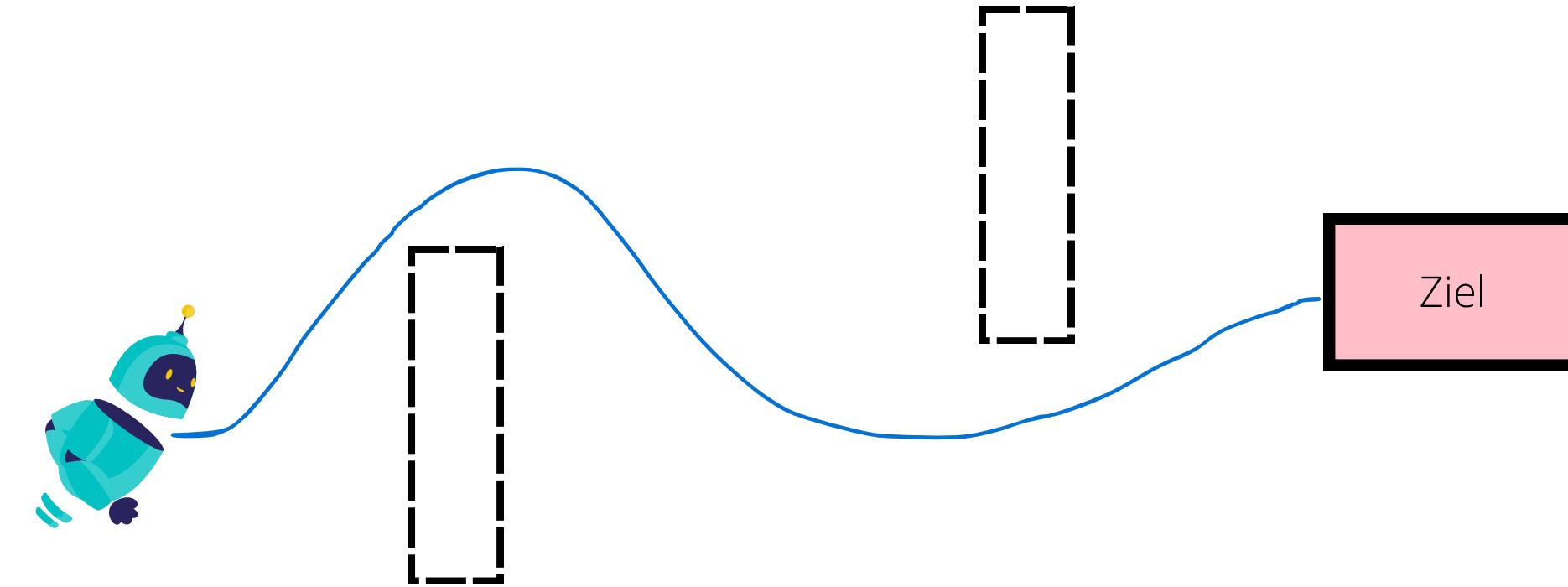
Ein Softwarestack ist eine Sammlung mehrerer Packages, die ein gemeinsames Ziel verfolgen.  
In diesem Fall ist das die Navigation eines Roboters.

# Was ist Navigation?



Wir wollen einen Roboter von A nach B bringen, auf **sicherer** Art und Weise und **autonom**

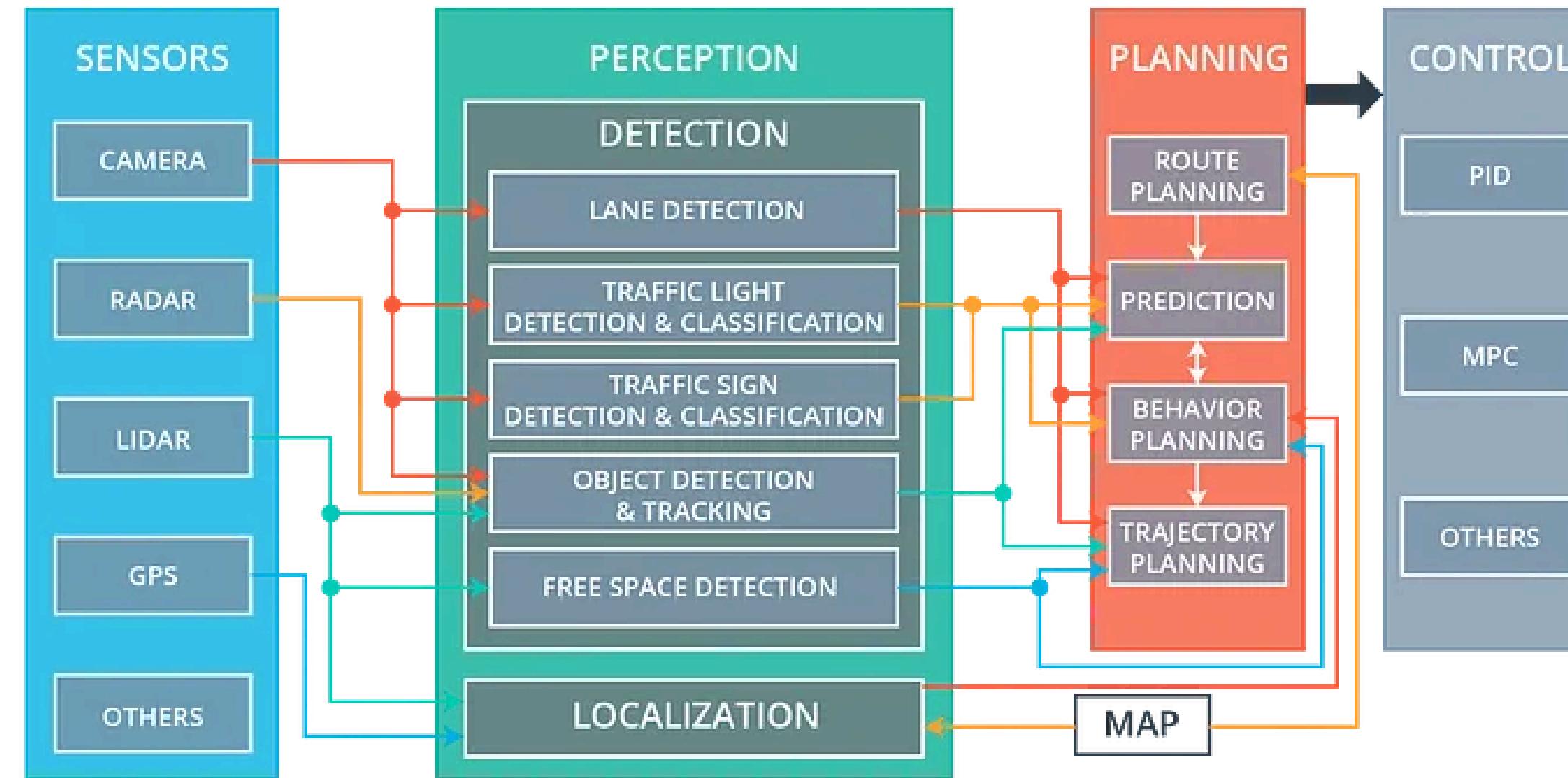
# Wie schaffen wir das?



Zwei Schritte:

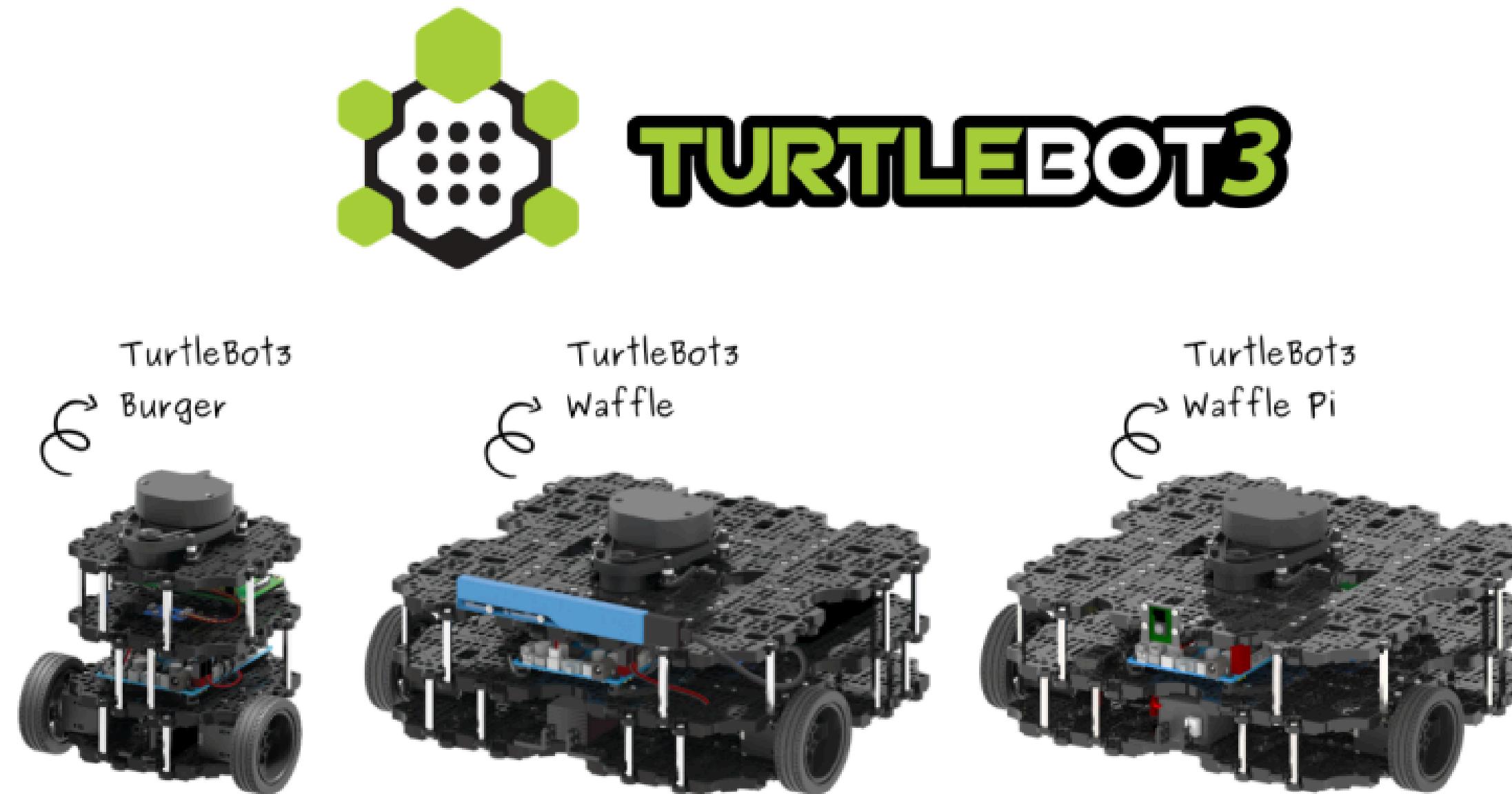
- Eine Karte erstellen mit **SLAM** (Simultaneous Localisation and Mapping)
- Den Roboter darin navigieren lassen

# Software Stack eines autonomen Fahrzeugs

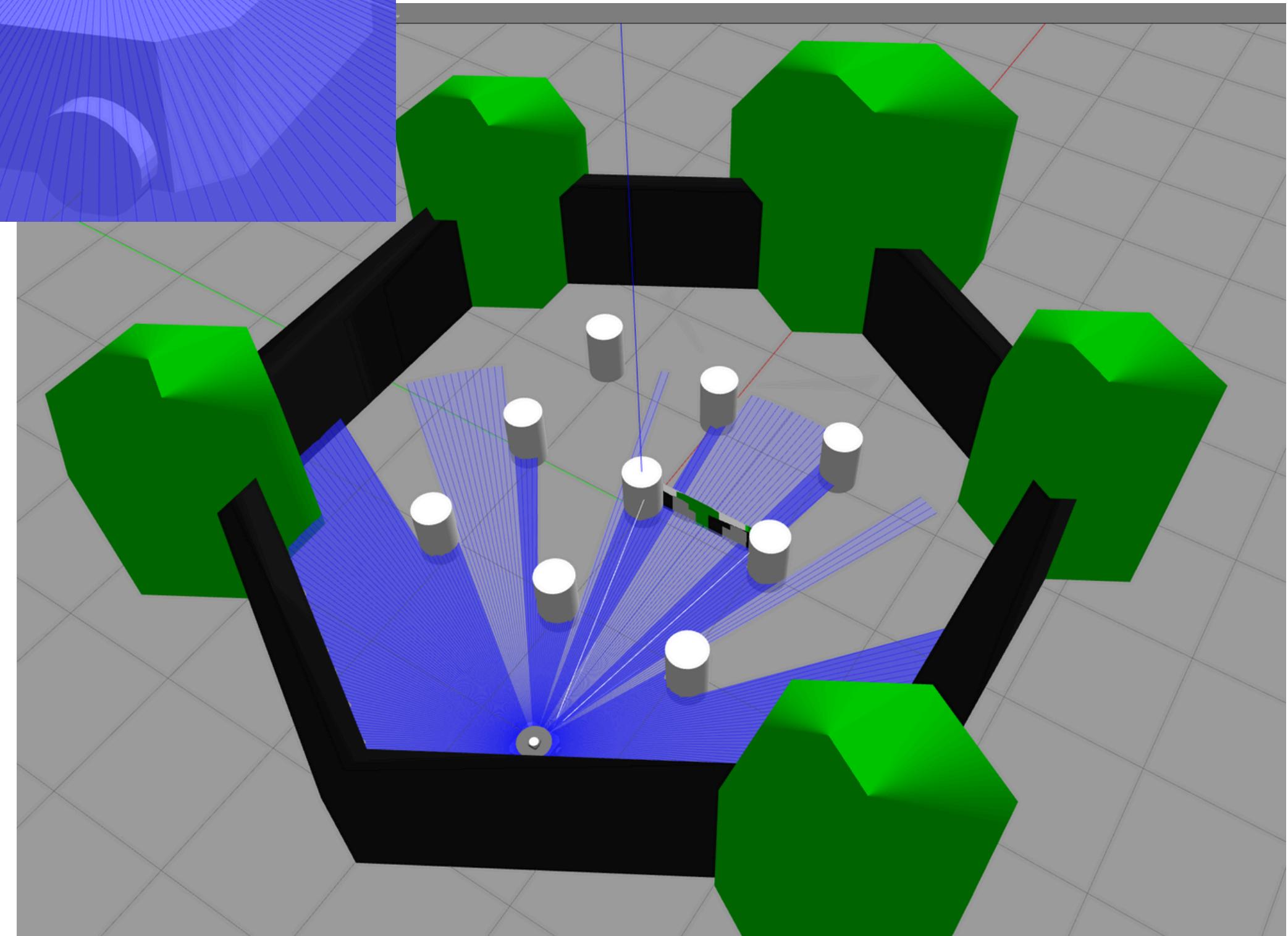
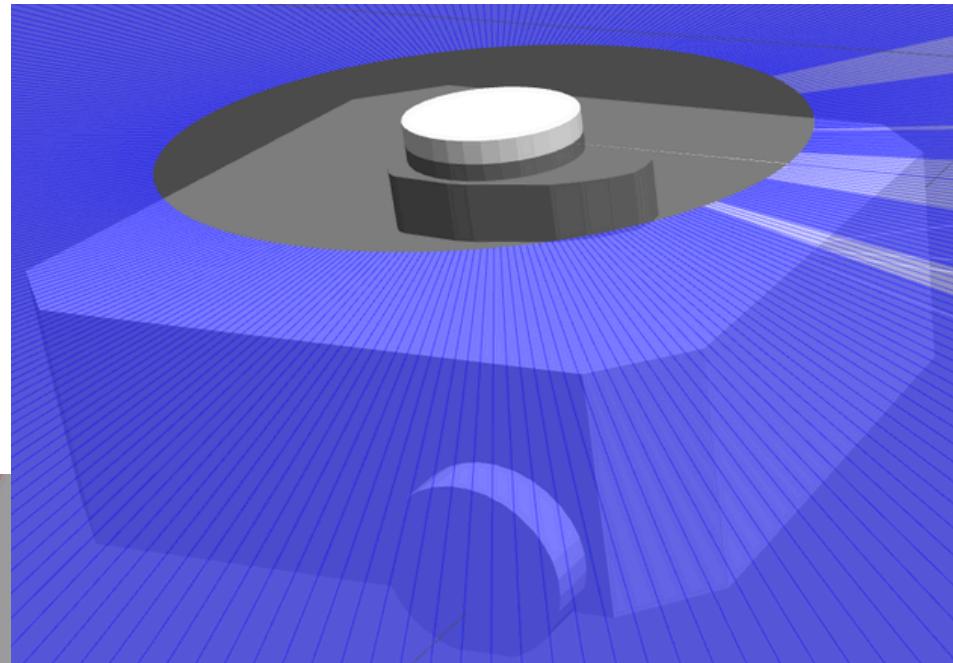
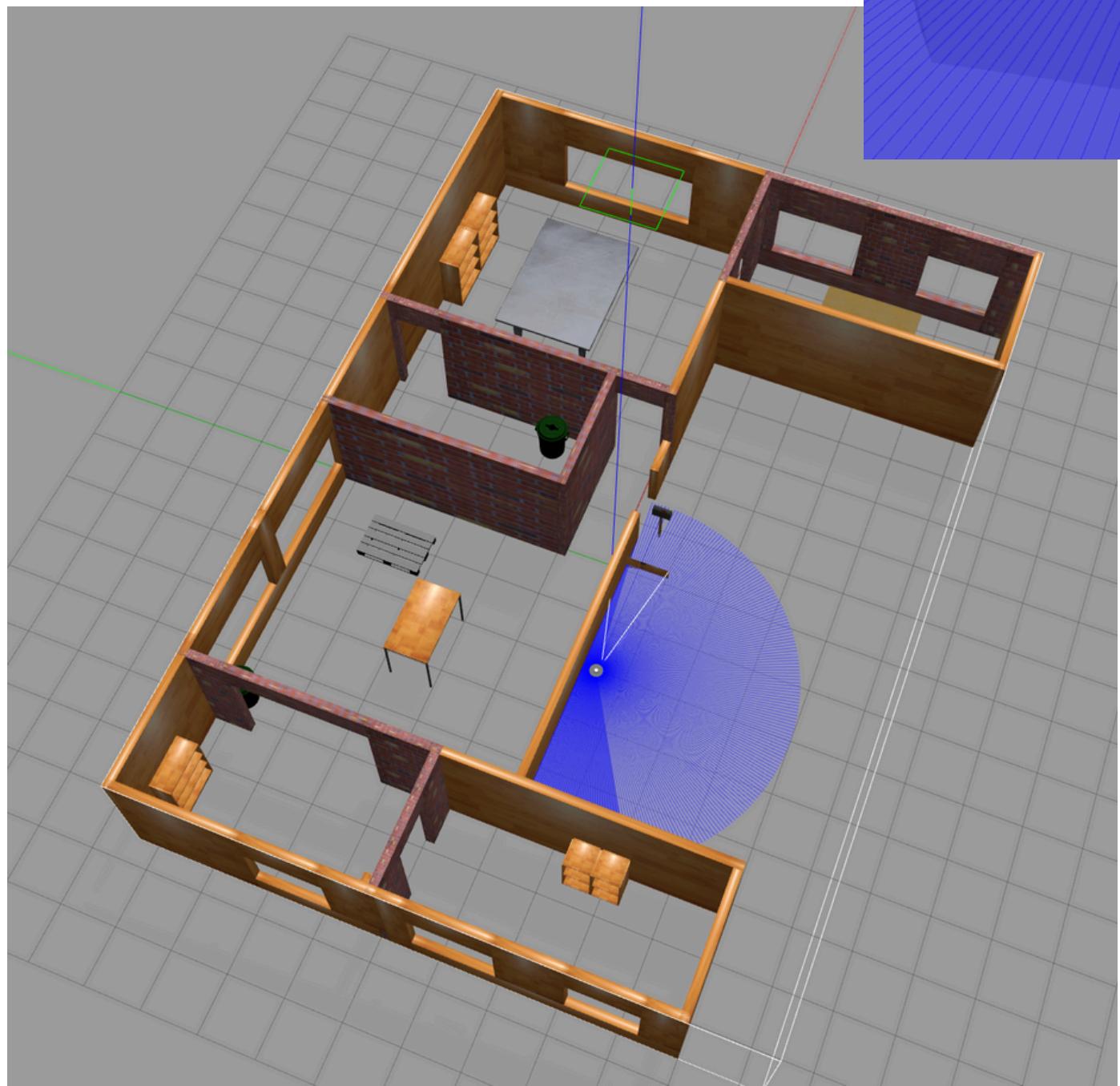


Quelle: Udacity Self Driving Car Nano Degree Program – Autonomous System overview

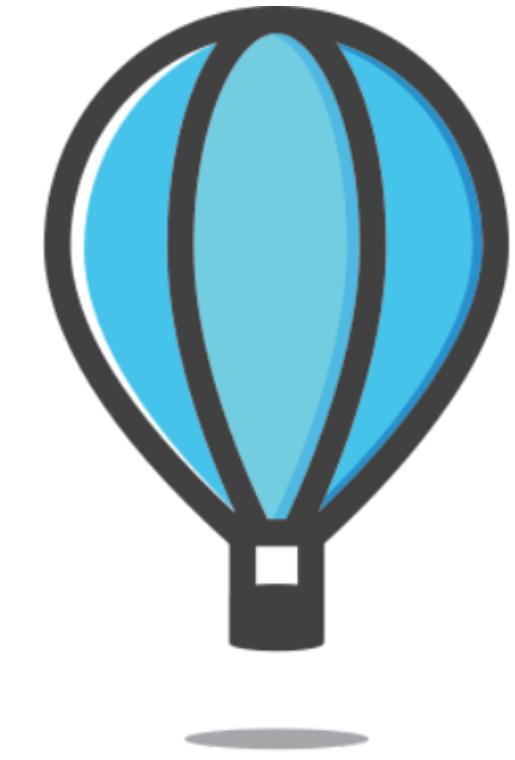
# Erste Einführung



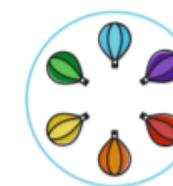
Quelle: <https://www.turtlebot.com/turtlebot3/>



# Nav2 Navigation Stack



N A V 2



**OPEN  
NAVIGATION**

<https://docs.nav2.org/index.html>

# Erste Einführung

Terminal 1:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

(alternativ):

```
ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

Teminal 2:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

Terminal 3:

```
ros2 topic list
```

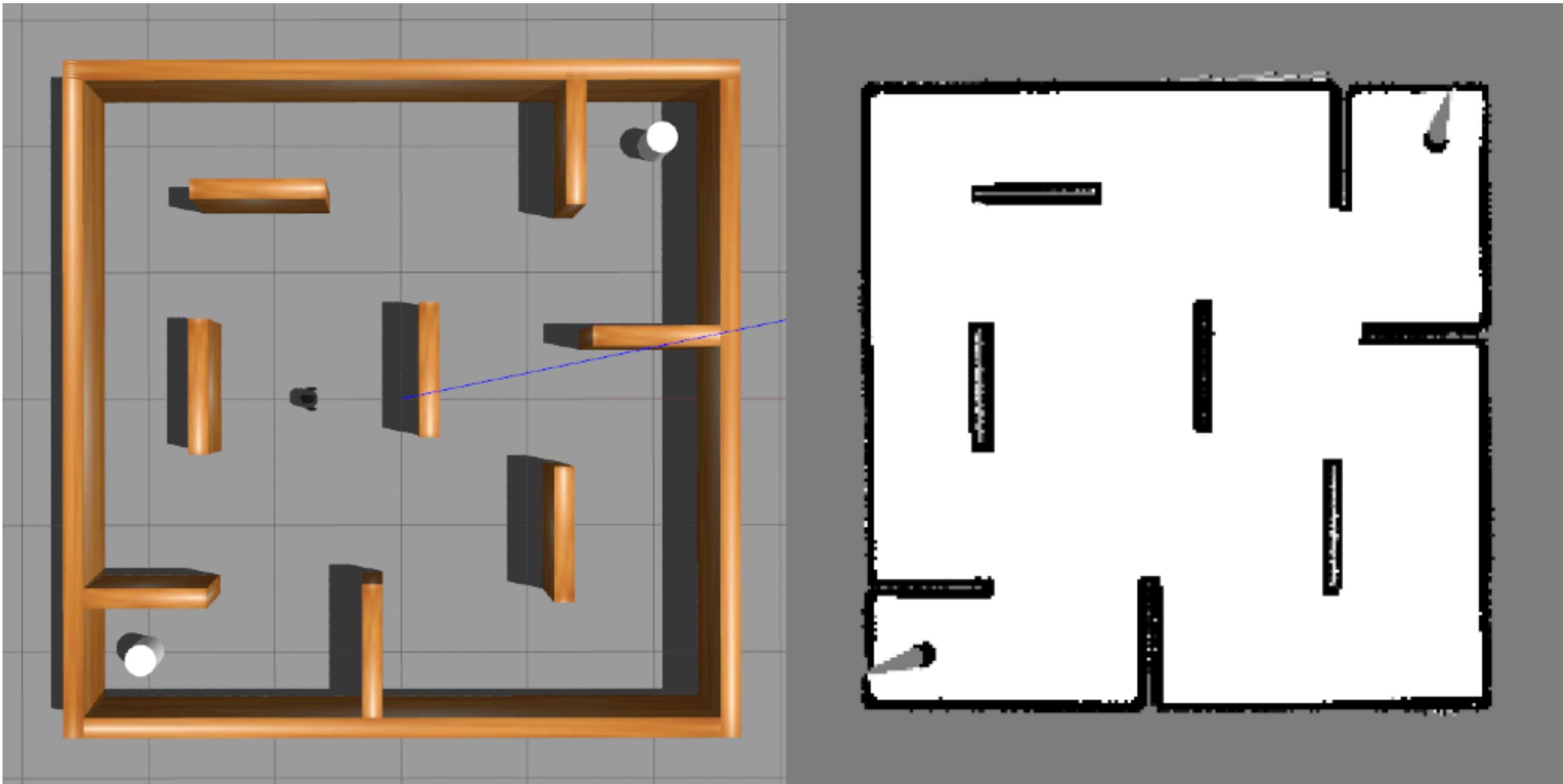
Terminal 4:

```
rqt_graph
```

# Nav2 Navigation Stack

Damit wir navigieren können, müssen wir eine **Karte (Map)** zur Verfügung stellen.

# Erstellen einer Map



# Erstellen einer Map



# Erstellen einer Map

Terminal 1:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Teminal 2:

```
ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

Terminal 3:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

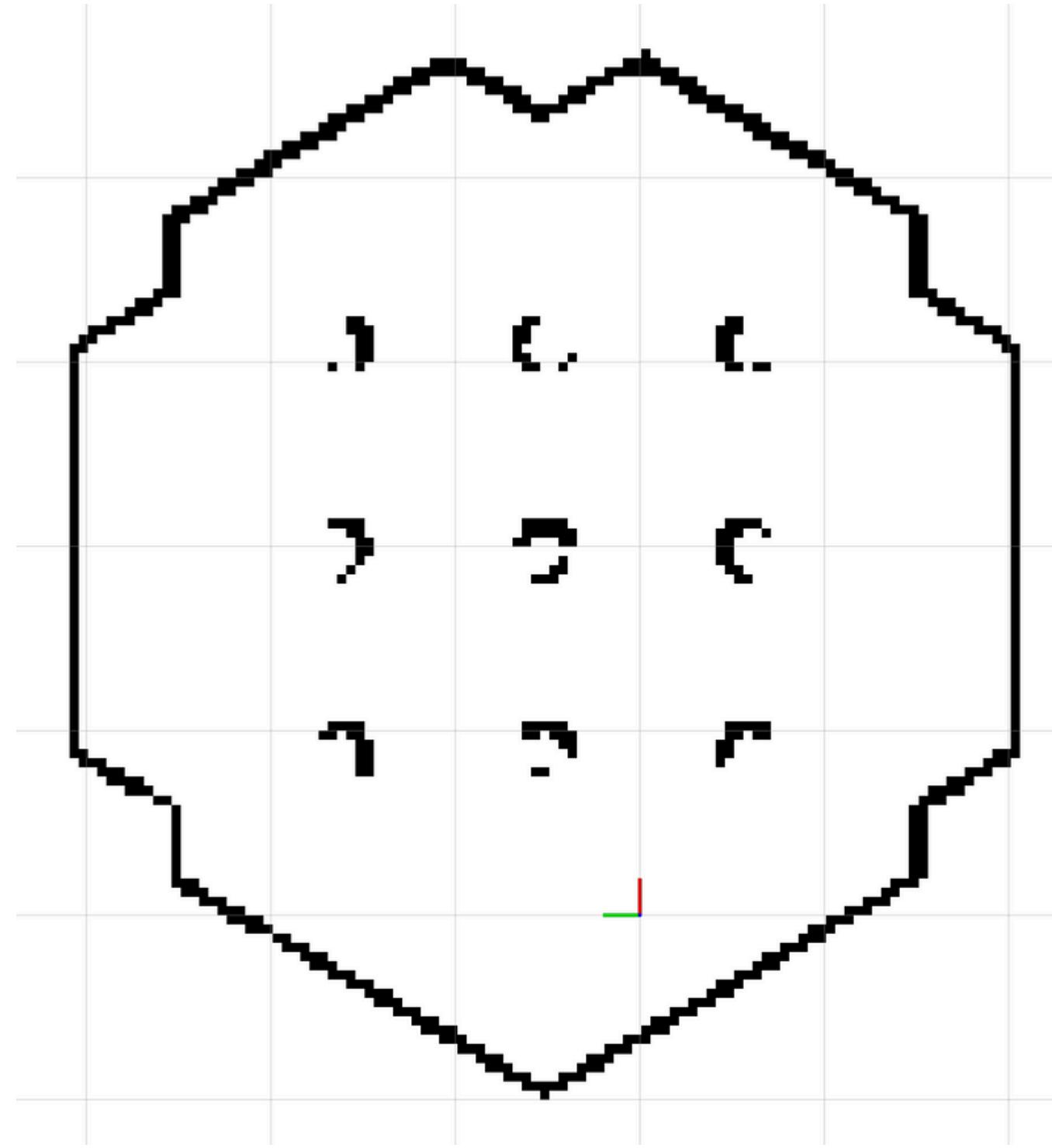
Terminal 4:

```
mkdir maps
```

```
cd maps
```

```
ros2 run nav2_map_server map_saver_cli
```

# Erstellen einer Map



# Coffee Break!



# Navigieren

Terminal 1:

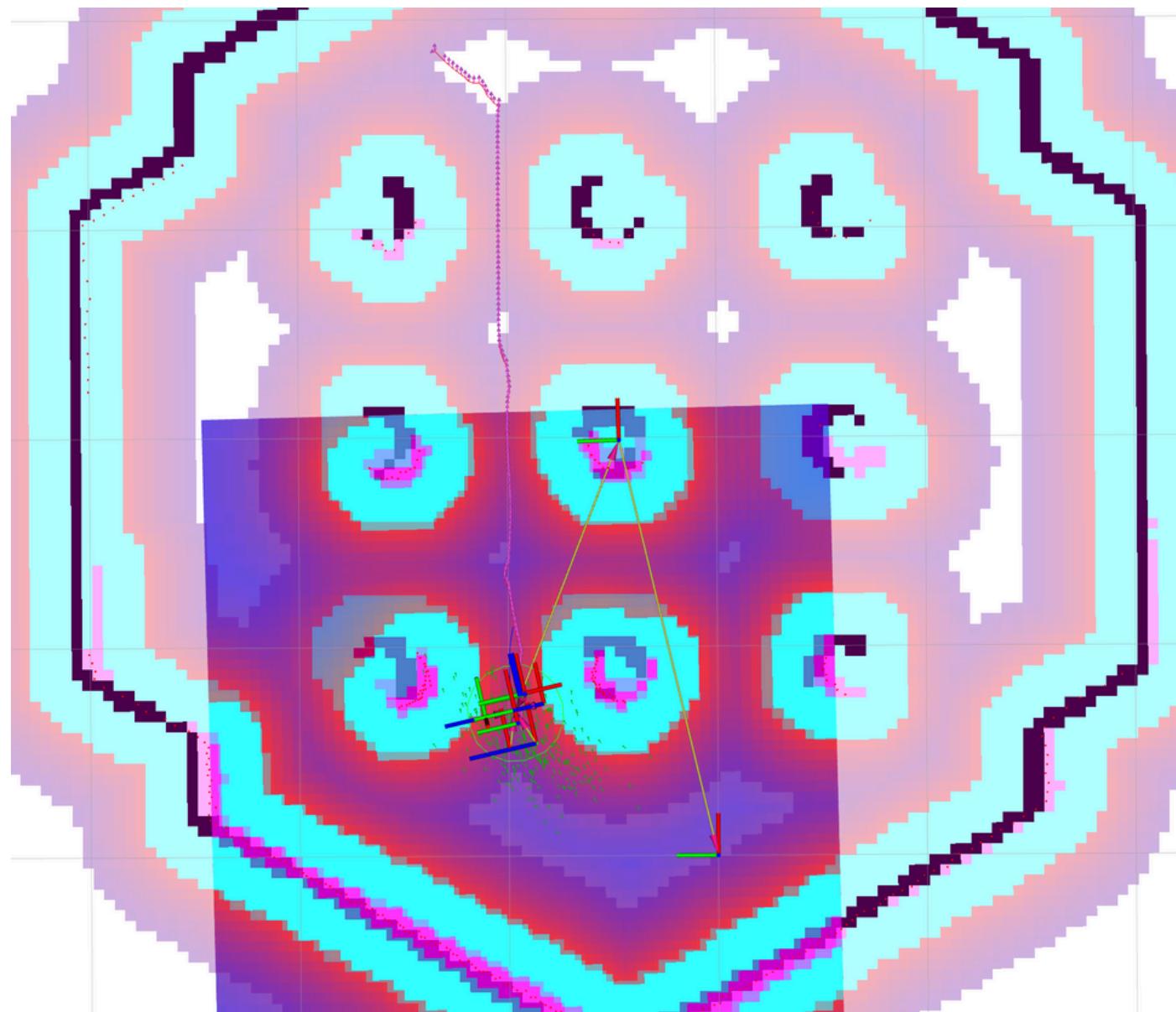
```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Teminal 2:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True map:=<map_name>.yaml
```

# Costmap

Die Costmap hilft dabei, den richtigen und sicheren Weg zum Ziel zu finden. Es wird dabei die von uns erstellte Map verwendet, in der noch einige wichtige Informationen hinzugefügt werden. Der Weg mit den geringsten Kosten wird bevorzugt.



- Der grüne Kreis ist der sogenannte "Footprint", dieser zeigt an, wie groß der Roboter ist.
- Der türkisfarbige Bereich ist die sogenannte Inflationlayer. Hier sind die Kosten sehr hoch, um Kollisionen zu vermeiden. Hindernisse werden somit aufgeblasen (inflated).
- Der lila-rote Bereich beschreibt die Kosten.



# Projekt: Publish Goal Points Package

Ziel ist es, eine Liste an Punkten vorzugeben, die der Turtlebot folgen soll. Sobald wir in die Nähe eines Ziels sind, wollen wir automatisch das nächste Ziel in der Liste verfolgen. Wenn wir alle vorgegebenen Punkte erreicht haben, sollen wir benachrichtigt werden, dass wir fertig sind. Legen wir los!

Terminal 1:

```
cd ~/ros2_ws/src/  
ros2 pkg create --build-type ament_cmake --license Apache-2.0 goal_publisher_package  
code .
```

# 1. Aufgabe: Dependencies

Als Erstes ist es immer gut, die Abhängigkeiten zu definieren, die wir später für unsere Funktion brauchen. Dafür müssen wir in der **package.xml**

Welche von beiden Optionen ist richtig?

Option A:

```
<include>rclcpp</include>  
<msg>geometry_msgs</msg>  
<dep>nav_msgs</dep>
```

Option B:

```
<depend>rclcpp</depend>  
<depend>geometry_msgs</depend>  
<depend>nav_msgs</depend>
```

## 2. Aufgabe: CMakeLists.txt

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(nav_msgs REQUIRED)

add_executable(goal_publisher src/goal_publisher.cpp)

ament_target_dependencies(goal_publisher
    rclcpp
    geometry_msgs
    nav_msgs
)
install(TARGETS
    goal_publisher
    DESTINATION lib/${PROJECT_NAME})
```

# 3. Aufgabe: goal\_publisher.cpp

Ganz oben müssen wir zunächst unsere Header einfügen.

Von geometry\_msgs benötigen wir pose\_with\_covariance\_stamped und pose\_stamped.

Von nav\_msgs benötigen wir odometry.

Außerdem müssen wir noch vector und cmath inkludieren.

```
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/pose_with_covariance_stamped.hpp"
#include "geometry_msgs/msg/pose_stamped.hpp"
#include "nav_msgs/msg/odometry.hpp"
#include <vector>
#include <cmath>
```

## 4. Aufgabe: Publisher and Subscriber

In Zeile des Publishers müssen wir nun unseren Publisher angeben, der die Ziele an unser Navigationssystem schicken will. Nehme Folie 24 als Inspiration, diesmal wollen wir an das Topic mit dem Namen “/goal\_pose” senden mit den Typ `geometry_msgs::msg::PoseStamped`.

In Zeile des Subscribers definieren wir jetzt den Subscriber, der “/odom” abhört und den Typ `nav_msgs::msg::Odometry` besitzt. Außerdem wird jedes Mal, wenn eine Nachricht von /odom ankommt, ein Callback mit dem Namen `&GoalPublisherNode::odom_callback` aufgerufen.

# 5. Aufgabe: Main-Funktion

Ganz unten finden wir die main-Funktion, die eigentlich immer gleich aussieht. Darum kannst du das Meiste aus Folie 26 kopieren und hier einfügen, du musst nur die Name der Node anpassen.

```
rclcpp::init(argc, argv);
auto node = std::make_shared<GoalPublisherNode>();
rclcpp::spin(node);
rclcpp::shutdown();
return 0;
```

# 6. Aufgabe: odom\_callback

In der if-Abfrage, können wir sehen, dass erst ein neues Ziel gesendet wird, wenn wir Nahe genug am derzeitigen Ziel sind. Berechne hierfür die Distanz zum Ziel.

```
double current_x = msg->pose.pose.position.x + 2.0;
double current_y = msg->pose.pose.position.y + 0.5;
double goal_x = current_goal_.pose.position.x;
double goal_y = current_goal_.pose.position.y;

double distance = std::sqrt(std::pow(current_x - goal_x, 2) + std::pow(current_y - goal_y, 2));
```

# Action!

Terminal 1:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Teminal 2:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True map:=<map_name>.yaml
```

VS Code Terminal:

```
ros2 run goal_publisher_package goal_publisher
```