# Architectural Layering in Autonomous Robotics: Modular Design and Implementation on the Turtlebot3 Platform using ROS.

Marvin Kohnen

July 7, 2024

## Contents

**Abstract**

This is going to be my abstract.

## 1 Introduction

This is going to be my Introduction

## 2 Related Work and Background

not

### 2.1 Robotic Architecture

#### 2.1.1 The need for Robotic Architecture

"Robot systems differ from other software applications in many ways. Foremost are the need to achieve high-level, complex goals, the need to interact with a complex, often dynamic environment, while ensuring the system's own dynamics, the need to handle noise and uncertainty, and the need to be reactive to unexpected changes." [2] This quote by Coste-Manière and Simmons perfectly introduces the various challenges robotic systems and their developers face. These needs shape their design, as all robotic systems embody *some* architectural structure and style. The architectural structure of a system outlines its division into various subsystems and the interactions between them, while the architectural style refers to the underlying computational principles of a system. [2] Different requirements and applications for robotic systems result in different architectures ("To date, the number of existing architectures has reached several hundred" [4]), some of which I will explore in the following section.

HIER COGNITIVE ARCHITECTURE DISKUTIEREN 2.1 Robotic Architecture 2.1.1 so lassen 2.1.2 Development of Cognitive Architecture und PRA 2.1.3 Layered Architectures

#### 2.1.2 Perceive - Reason - Act

The "Perceive-Reason-Act" (**PRA**) model takes its inspirates from Neuroscience: **P**erception creates a model representing the current world state. The **R**eason component designs a plan using the model to find an appropriate action towards the goal. Finally, **A**ct refers to the translation of the action into actuator commands and therefore resulting in motor output. [16] The rather intuitive PRA approach contains some drawbacks in real world applications: First and foremost, not every aspect of the environment is senseable (such as the inner state of other agents). Furthermore, key elements of the world may be difficult or rather impossible to model. Lastly, sensor and actuator measurements in the real world might differ vastly from the expected values in a clean lab situation. [16]

#### 2.1.3 Layered Architectures

**Layered architectures** take a different approach to resolve the issues that the PRA model encountered. Horizontal layers of control systems are built to let the robot operate at increasing levels of competence and complexity and are assigned with different objectives. They consist of asynchronous modules which are allowed to communicate in a bidirectional manner. [1] Even though the higher levels can subsume the roles of lower levels by suppressing their output, the lower levels maintain their functionality as higher levels are added - resulting in a robust and flexible robot control system. [1] An example of a layered architecture can be seen in Figure 2, created by Brooks in 1986. [1]

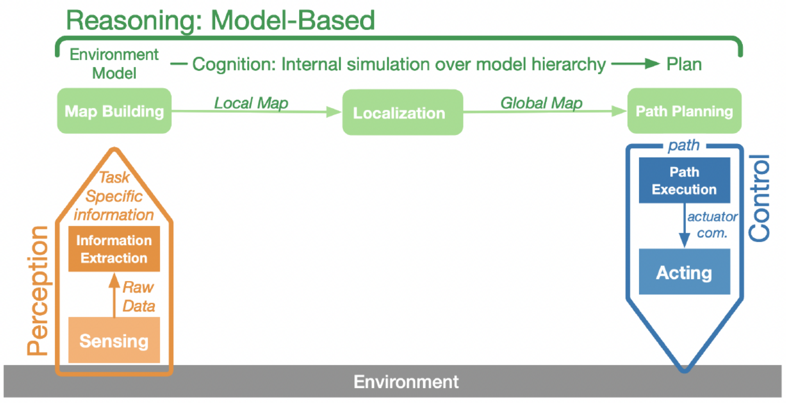


Figure 1: Perceive-Reason-Act [16]

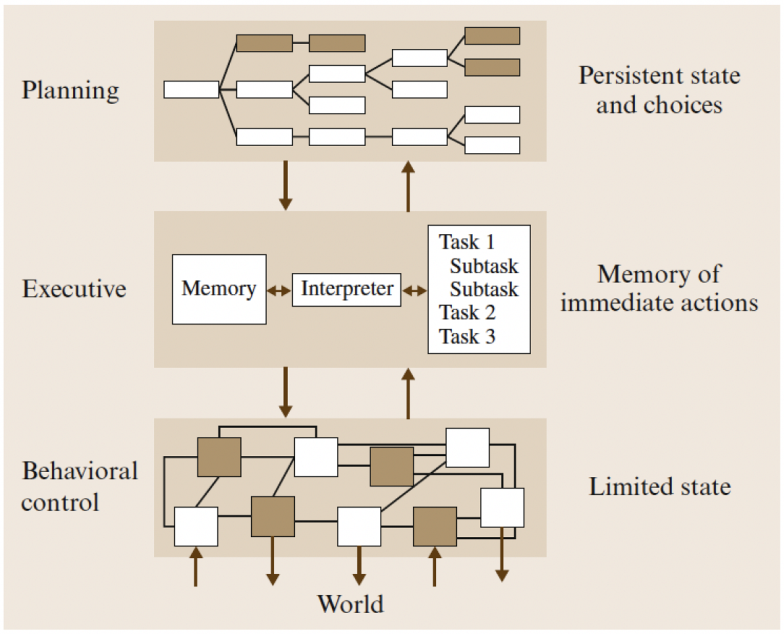


Figure 2: Layered architecture [1]

Historically, layered architectures are composed of three layers: Behavioral layer, Executive layer and Planning layer. [3, 8]

*The Behavioral layer* (elsewhere called the Hardware Abstraction- or Functional layer) is typically found at the bottom of the hierarchy inside a layered architecture and provides a low-level library or interface to access the robot hardware, such as sensors or actuators. [3, 18] It is often composed of very fast control loops, to create operations like path tracking or reflex-like reactions. [16]

*The Planning layer* sits at the top of most architectures, making the decisions to achieve high level goals. This more complex and slower layer then sends the plans to the *Executive layer*, which further decomposes tasks into subtasks. It is responsible for translating high-level plans into low-level behaviors, invoking behaviors at the correct times, monitoring execution, and handling exceptions. [16, 18, 19] As stated earlier, information and control is allowed to flow up and down through these layers, as the Behavioral layer sends back sensor data and actuator information to the executive layer, which then informs the Planning layer on the current status of the assigned tasks. [3] A prototype of this structure can be seen in Figure 3. This architectural concept can even be extended to multiple robots, creating a sort of hive or swarm control, as Simmons et al. have shown in 2002. [18]

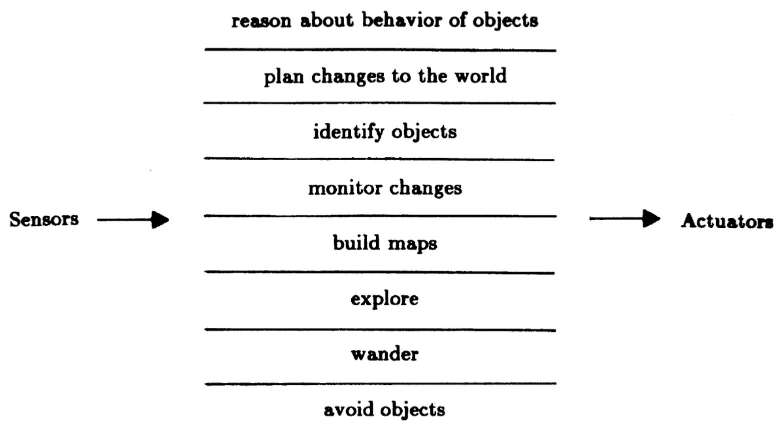


Figure 3: Prototype three-tiered architecture [17]

### 2.1.4 CLARATy

While this three layered architecture is very popular and has been thoroghly investigated by developers, it doesnt come without its drawbacks. The developers of **CLARATy**, a two layered architecture that is used for NASA's space robots [17], identified three problems when designing three layered architectures: Firstly, the developers "expand the capabilities and dominance of the layer within which they are working" [19]. This results in models, in which certain layers are predominantly active. Furthermore, there is still research to be done over the hierarchical superiority of the planning and executive layer of one over the other. [19] Another problem the authors state is the lack of access from the Planning level to the Behavioral control. This forces the system to carry own models on the Planning layer, which are not directly derived from the functional layer, to perform planning tasks. Eventually, this can lead to inconsistencies between the layers and endanger the integrity of the system. [19] Lastly, the authors claim that there is a misconception in equating greater intelligence in a system with increased granularity, suggesting that each part of a system can possess its own distinct hierarchy of granularity. They move on to explain that the Functional Layer contains many nested subsystems, the Executive layer encompasses multiple logic trees to coordinate these subsystems, and the Planning layer includes various timelines and planning horizons with different resolutions. [19] In order to fight these challenges, Volpe et al. proposed a two layered architecture. This offers two main benefits:
Firstly, an explicit Representation of Granularity: CLARATy introduces a third dimension for granularity, allowing for a clearer depiction of the system's complexity and hierarchy. This means that within the Functional layer there's a detailed organization showing how subsystems are nested or grouped together. Secondly, they blended the Planning and Executive layer, resulting in a *decision* layer which is more efficient and allowed the levels to share a common database (depicted in figure 4). [19]

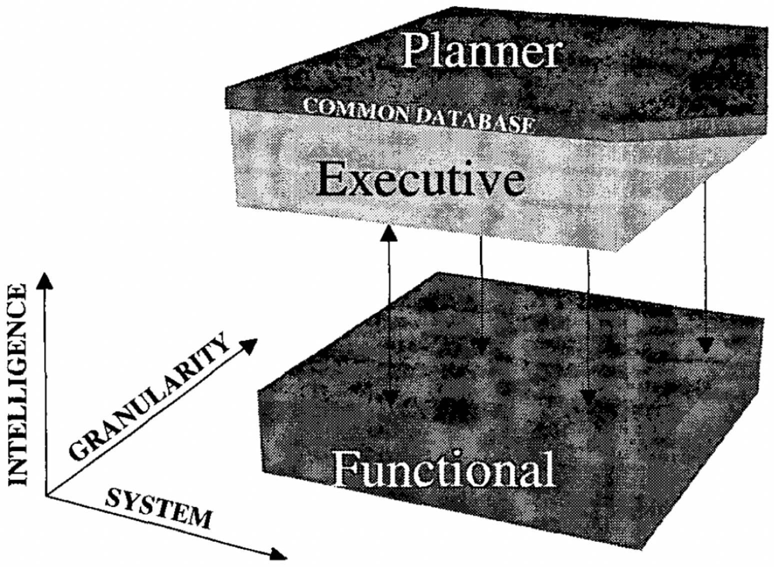


Figure 4: CLARATy architecture [19]

### 2.1.5 Cognitive Architectures

Ich weiß nicht, ob ich hier noch kurz Cognitive Architectures ansprechen sollte, als eine Art Outlook (ca halbe Seite?)

### 2.1.6 Übergang (TITEL?)

Understanding the principles of layered architecture provides a solid foundation for appreciating the architectural choices and design patterns employed in robotic systems. As we have seen, there is no one definitive way to architect these systems. Rather, this framework provides some key characterics by its division of system functionalities into distinct layers, each with a specific role, offers clarity, modularity, and scalability. As we transition our focus to the Robot Operating System (ROS), it becomes evident how ROS embodies these principles through its comprehensive framework for robot software development.

## 2.2 Robot Operating System

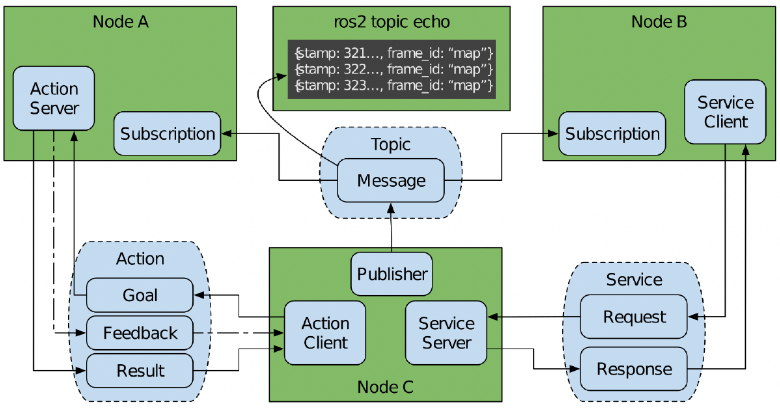
### 2.2.1 ROS 1

The Robot Operating System (ROS) is an open-source operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS is not an operating system in the traditional sense of process management and scheduling. Rather, it is a framework or middleware upon which robotics software is developed, offering a structured communications layer above the host operating systems of a heterogeneous compute cluster. [**ros.orgIntroductionROS2024**, 9] ROS's architecture is designed around the notion of the intercommunication between numerous computing processes (often referred to as nodes) over a peer-to-peer network. This decentralized computing architecture enables the development and integration of complex robotic systems from modular and reusable components or packages. The system facilitates a distributed computing environment, allowing for the separation of tasks such as sensing, perception, decision-making, and actuation across multiple processors and machines. [9] Central to ROS is its communications infrastructure, which enables the exchange of messages in various forms. These fundamental concepts include messages, topics, services and above mentioned nodes. The latter communicate with one another through messages

(”strictly typed data structures” [9]). This communication is implemented as a *publish-subscriber system:* On the one hand Nodes send messages to a topic by publishing the information, on the other hand nodes can receive the published information by subscribing to said topic. Multiple concurrent publishers and subscribers for the same topic may be active at any time. Also, a single node can publish and subscribe to many topics. [9] Additionally, ROS provides capabilities for package management and a vast ecosystem of community-contributed tools and libraries that address various robotics functions ranging from planning and perception to simulation and visualization. [**ros.orgIntroductionROS2024**, 9]

### 2.2.2 ROS2 and its improvements

Even though ROS layed the groundwork for the modern robotic industry, the growing interest of industrial applications showed the limitations of ROS, which was built as a research platform. The urge for security, reliability in non-traditional environments and support for large scale embedded systems became the driver for the creation of ROS2, the second generation of the Robotic Operation System. [6] It introduces Node Lifecycle management, replaces the custom ROS1 middleware with industry standard middleware DDS (Data Distribution Service), better Real-Time support and adds Actions as a communicational tool. Actions in ROS2 are designed for long-duration tasks that require feedback to the caller during execution, which makes them distinct from the simpler service-call model. [6] A typical node interface for ROS2 can be seen in the figure 5 below. eventuell ROS1 nicht erklären und alles für ROS2 zusammenfassen?



```
{stamp: 321..., frame_id: "map"}
{stamp: 322..., frame_id: "map"}
{stamp: 323..., frame_id: "map"}
```

Figure 5: ROS2 Node interfaces: topics, services and actions. [6]

## 2.3 How do you architect your robots?

### 2.3.1 State of the Practice and Guidelines for ROS-based systems

Malavolta et al. (2020) conducted a study to answer three research questions: the architecture-relevant characteristics of ROS-based systems, the quality requirements considered in their design, and how to guide roboticists in architecting such systems. The authors used a two-part approach, mining ROS repositories and surveying developers who contributed to these repositories. The study found that ROS systems are becoming large and complex, and while there are many open source packages and examples on how to use ROS, engineering robots with specific properties is still mostly an art of trial and error.The findings show that the most mentioned quality requirements related to ROS architecture were maintainability, performance, and reliability. The study also identified 39 guidelines for architecting ROS-based systems. These guidelines are useful for both roboticists aiming to develop high-quality robots and architecture researchers looking for evidence-based indications on how real-world ROS systems should be architected. I will now present 7 out of those 39 guidelines, which were evaluated as the most impactful for architectural design of robotic systems. [7]

### 2.3.2 Guidelines

1. **Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.**

   This first guideline emphasizes the importance of using standardized ROS message formats, particularly those from the common_msgs and std_msgs packages. Adopting these standards facilitates easier reuse, upgrade, and replacement of ROS nodes by allowing for straightforward topic remapping. Standardized messages enhance development efficiency by making compatible tools, such as visualizers and SLAM algorithms, easier available. They also improve node testability through isolation testing with ROS bags and simplify the integration of new sensors and hardware, as many manufacturers support ROS out of the box. However, since the definitions of standardized message formats are subject to change, it's crucial to design systems to be as decoupled as possible from specific message formats to ensure future compatibility and ease of updates. [7]irgendwie eine komische aussage finde ich

2. **Group nodes and interfaces into cohesive sets, each with their own responsibilities and well-defined dependencies.**

   Growing complexity poses a serious challenge while designinng ROS-based software architectures, as it can result in hundreds of interdependent nodes leading to technical debt, limitations to certain ROS packages and reduce adaptability and extendability of the whole system. To mitigate these issues, the guideline suggests grouping nodes and interfaces into cohesive sets with defined responsibilities and dependencies. This structured approach aids in maintaining a clear understanding of the system's architecture, allowing for easier evolution and maintenance of the system. [7]

3. **The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at runtime.**

   ROS inherits a certain flexibility that allows developers to create nodes without prescribed behaviors, offering great freedom but posing challenges for testability, reliability, and maintainability of stateful nodes. This guideline states the importance of treating node lifecycles as a crucial aspect of system design, suggesting documentation of node lifecycles to improve interaction, predictability, and test case development. As stated earlier, the development of ROS 2 already addresses some of these issues by supporting managed nodes with defined lifecycles (Unconfigured, Inactive, Active, Finalized) that can be inspected and controlled, enhancing testability and system management. Additionally, some ROS packages, even outside ROS 2's managed nodes, incorporate lifecycle considerations into their APIs, enabling runtime configurability and reflection to bolster system adaptability and autonomous capabilities. [7] Defining and enforcing the lifecycle of ROS nodes "can enhance the system in terms of run-time configurability and reflection, which can be exploited for providing autonomous capabilities". [7]

4. **Assign meaningful names to components (e.g., nodes, topics, services) and group them by adopting standard prefixes/suffixes.**

   This pretty self explanatory problem is especially importan in ROS-based systems, as topics and services are created programmatically by the nodes at run-time and their identifiers are simple strings. [7]

5. **Nodes directly interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system**

   This guideline essentially boils down to creating uniformity in messaging interfaces, which allows developers to easily swap between simulations and hardware applications, while reducing modification cost to a minimum. [7]

6. **ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.**

   This principle highlights the challenges developers face when working with hardware sensors in robotics: Sensors often produce data in bursts, can degrade, and become less accurate over time. To handle these challenges, developers are advised to design their systems with flexibility in mind, by implementing load balancing nodes to manage sudden data surges and designing nodes to be resilient to gaps in sensor data. This resilience is crucial for the reliability of state estimation nodes, where faults can cause significant system-level failures. I expect this guideline to not be of great relevance in my project, due to the rather small(-ish) dataload. Nonetheless, a cruicial thought to keep in mind, when designind robotic systems. [7]

7. **Avoid persisting raw data if only part of it will be used.**

   This guideline refers to the obstacle of how extensive data logging or storage operations can negatively impact the system's performance during its operation. Developers are advised to selectively record only the necessary data to avoid these performance issues. [7]

Our architecture will try to adhere to these guidelines as much as possible, in order to create a robust and performant robotic system.

## 2.4 Niryo One Arm Architecture

The Niryo One is a collaborative and open-source 6-axis robot designed for research and higher education in the context of the industry 4.0. Its architecture has been mentioned by Malavolta et al. as an example, that adheres very well to above stated guidelines. [7] Niryo One operates upon a layered architecture, which is composed of five layers. Each layer is further subcategorized into different packages, among them pre-built ROS packages, imported libraries or self-created packages.
At the lowest level we can find the **Hardware layer**. This layer contains a package *rpi* that handles all the external hardware (everything else than the motors) of Niryo One, and provides many utilities for the Raspberry Pi 3, such as Wi-FI, LEDs, Buttons etc. Furthermore, a *driver* package is built, which provides an interface to ros_control and handles the hardware control of motors. The other two packages found in figure 6 (*dynamixel_sdk* and *mcp_can_rpi*) are imported libraries to stabilize functionality of the Raspberry pi and control the actuators. [11]

Next up, a **control level** is implemented, which is entirely made up of the pre-built ros packages *ros_control* and *joint trajectory controller*. The former is a set of packages that include controller interfaces, controller managers, transmissions and hardware interfaces [15], while the latter provides a controller for executing joint-space trajectories on a group of joints. [12]

The actual motion control is invoked by the next higher level, the **Motion Planning layer**. Not only is there a *description* package, containing the URDF file and meshes for Niryo One (which is essentially a description for the robot), but also the widely used *moveit* packages, which is already part of the ROS ecosystem. The motion planning layer is responsible for finding inverse kinematics and building a path for the robot.[10, 14] For each point, each axis are given a specific position, velocity, and acceleration. This path is sent to the joint trajectory controller for the actual hardware execution. [10]

On top of that, the **Commander layer** provides a higher level interface between the client and the underlying robot commands. All commands, that the user inputs (e.g. ”move joints”), go through this layer. The *commander* package handles concurrent requests, validates parameters and calls required controllers and then returns appropriate messages and status. Furthermore, a *python_api* package allows for easier access for other developers and tries to reduce the complexity of ROS for beginners. [10, 11]

On the highest level, the architecture features a **External communication layer**. This layer allows for communication outside the ROS ecosystem. First a TCP server is established in the *modbus* package. It offers a simple way for developers to create programs for robot to control them via remote communication on a computer, on a mobile or any device with network facilities. Lastly, the ROS-packages *rosbridge* and *joy* are imported, in order to allow communication between ROS1 and ROS2 and the use of the joystick controllers. [10, 11]

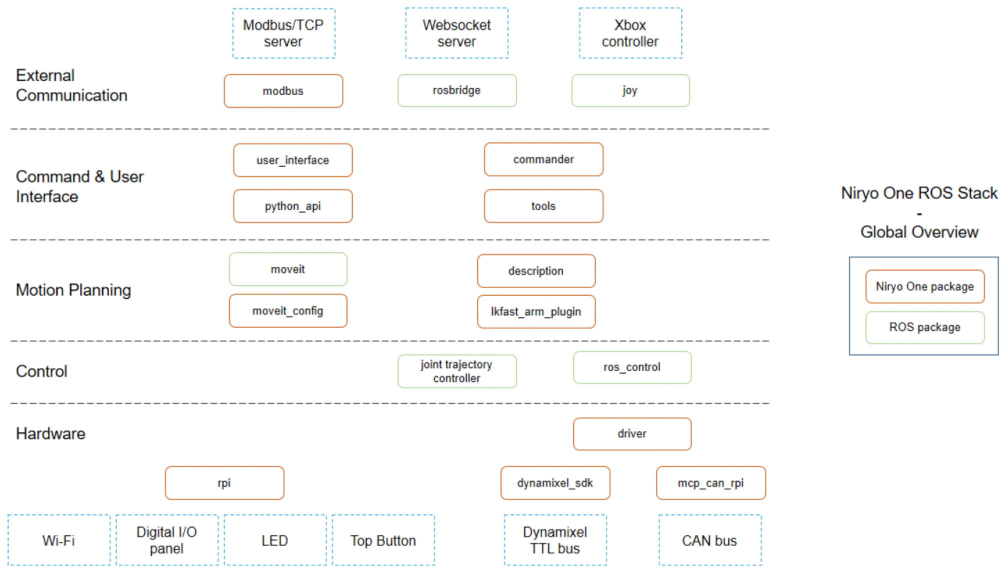


Figure 6: Niryo One ROS Stack overview [11]

After this thorough examination of existing architectures, we will now introduce our own approach to a layered robotic system.

# 3 Methodology

## 3.1 Frameworks and Setup

The architectural system is employed on the Turtlebot3 Burger Bot platform. This robot is frequently used in education and research because of its size, affordability and wide range of adaptability and expandability. A brief overview of the robots composition is shown in Figure 7.

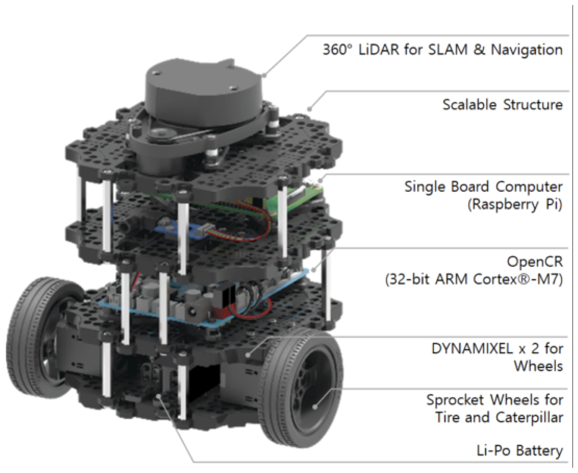

360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 7: Turtlebot3

The Architecture is built on the Ubuntu 22.04 LTS Operating System and ROS2 Humble for both the remote PC and the raspberry pi4 on the Turtlebot3. Any simulation is done with Gazebo Fortress (as required for ROS2 Humble) and RVIZ, while the behavior of the robot is established in various Python and Bash scripts. SSH protocol is used to establish a connection between the remote Interface and the Turtlebot3 in order to remove some computational from the raspberry pi.

## 3.2 Architecture and Design

The architecture is structured into multiple distinct layers as shown in Figure 8, which grow in complexity the higher the layer is placed in the diagram.

At the foundational level, the *Hardware Control* package serves as the primary interface with the robot's physical components. It is the only layer [apart from the nav2 package, which we will get to later on] that processes and publishes commands directly to the robot's motors and sensors. This module's design includes a prioritization mechanism that manages conflicting commands to ensure that higher priority tasks, such as emergency stops, preempt other actions. The Hardware Control acts as of library of functions for the other layers to call upon, in order to create movement, e.g. "Drive Forward", "Turn Left", "Turn Right", "Turn by Angle" etc.").

Above that is the rather straightforward *Obstacle Avoidance* layer, which continuously reads sensor output from the 'LaserScan' message [13] and calls upon the hardware layer to immediately stop and rotate the robot. This command is sent with the highest possible priority.

For map management, the architecture integrates a SLAM Toolbox Das ist das erste mal, dass SLAM erwähnt wird - Muss ich das noch erklären in einem Extra abschnitt?. This tool is critical for building and updating the map while the robot navigates through its environment. The SLAM process simultaneously localizes the robot within the map and updates the map's structure based on new information gathered from the robot's sensors.

In order to make use of the SLAM Toolbox and create a map, the robot has to wander around in its environment. This is realised through the *Random Explore* layer, which does exactly what the name suggests: Randomly explore the surrounding area by invoking movement through the Hardware layer with lowest priority commands.

In the same package as the SLAM Node, a *Map Change Detection* Module is located, which continuously monitors the environment for changes, updating the navigation map on a set frequency (currently 30 seconds). It uses a change detection algorithm that compares the current map state with new data collected by the robot, ultimately deciding when the map is robust enough to request the navigational layer to take over the robots movements.

At the highest level, this *Navigation Module* provides autonomous navigation capabilities by generating random goals within the environment and employing the A* algorithm for pathfinding. This module dynamically calculates the most efficient paths to the designated targets, considering the current environmental layout and obstacles as previously mapped by the system. It is important to note, that two concurrent packages are placed inside this layer: One is custom built while the other employs the nav2 system, a commonly used navigational framework which provides a wide range of algorithms and functionalities to facilitate Path planning and directed movement. [5]

Lastly, there are a couple more modules which provide analytical output for this thesis such as Maps, Heatmaps or position logging. These modules dont impact the robots behaviour in any case. Nonetheless, they are part of the architecture and are worth mentioning. Most importantly, a Heatmap Generator is placed in this *Analysis layer*. As soon as any navigational functionality is invoked, it maps the robots position from the /odom frame to the /map frame and therefore creates a heatmap that matches the given dimensions by the SLAM Toolbox. Some examples of these heatmaps will be shown in Chapter (INPUT CHAPTER!).

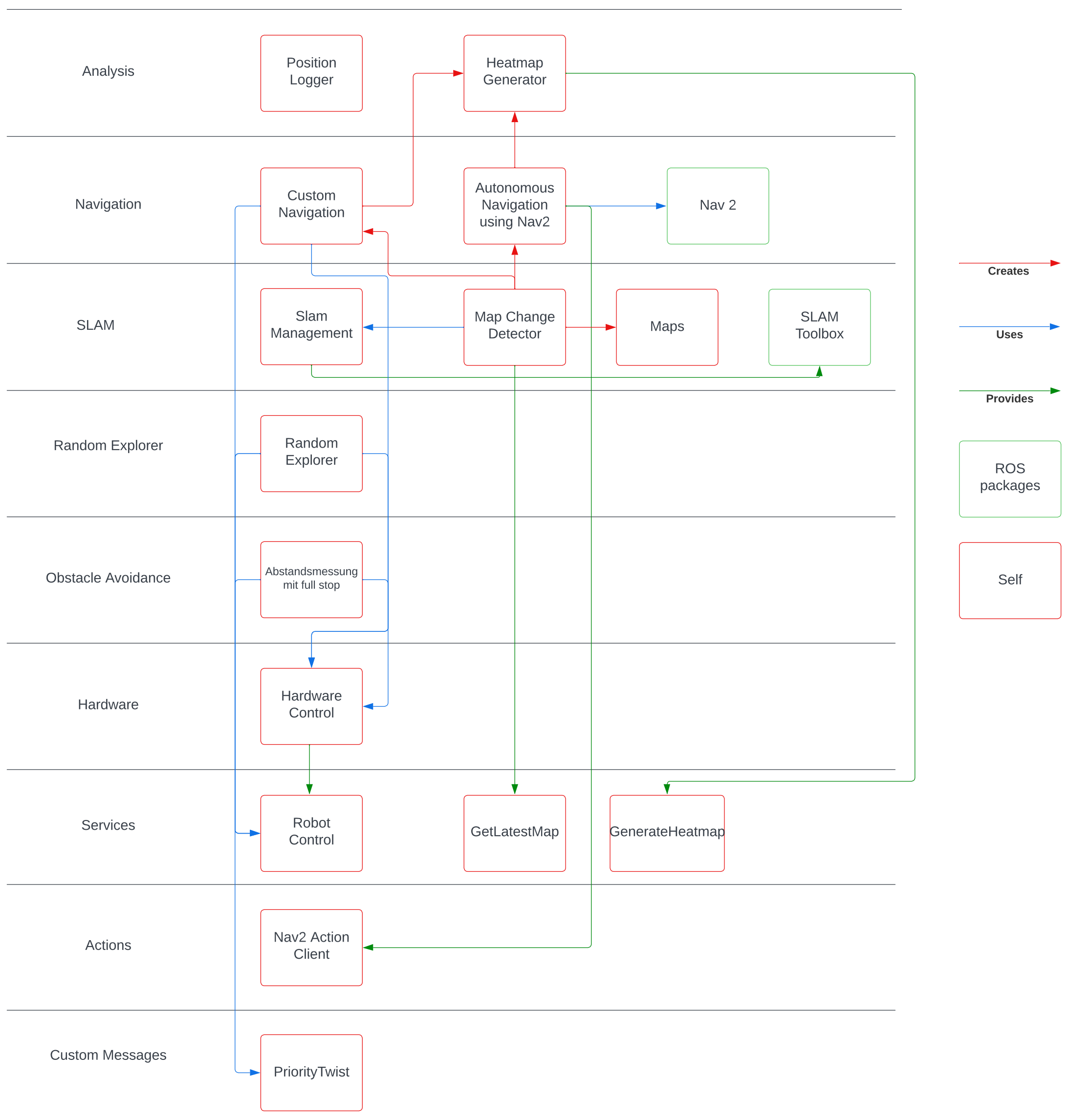


Figure 8: Architecture Overview

## 4 Comparison to Malavolta et al

Since our architecture has now been established, we can discuss its adherance to the earlier introduced guidelines by Malavolta et al. Then we will move on to qualitative and quantitative analysis of the system in the form of an ablation study and the comparison of the generated heatmaps.

1. **Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.**

   The architecture uses mostly standardized ROS message formats, such as LaserScan and Odometry. The external tools used in the system, such as SLAM Toolbox and Nav2, also use standardized ROS message formats: LaserScan, Odometry and Twist. The only time the architecture deviates from this guideline is through the introduction of the custom PriorityTwist message. This design choice was made consciiously, since the other possible solution for the prioritization mechanism involved environmental variables or global variables which not only are not common practice in Python, but also are very hard to maintain and reduce portability of the system. Furthermore, the priority is rather straightforward to understand, since it basically contains a normal Twist message at its core with only an added integer field.

2. **Group nodes and interfaces into cohesive sets, each with their own responsibilities and well-defined dependencies.**

   Since each layer provides a specific functionality in the greater system, dependencies are kept to a minimum. Apart from the standard ROS message dependencies, most layers only depend on the Hardware Control package (or rather the Robot Control service). An exeption makes the Navigational layer, which is connected to the vast framework of Nav2. These dependencies are necessary and therefore not in our control when designing this architecture.

3. **The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at runtime.**

   Each package in the system invokes and manages its own nodes, which is a result of the ROS2 Humble middleware with its managed nodes in order to enable runtime configura- bility. Furthermore, any node that is not initialized by the user at the start of robot operation has a well-defined lifecycle. Since the architecture is autonomous, most nodes are not designed to be updatable at runtime once their lifecycle has been invoked.

4. **Assign meaningful names to components (e.g., nodes, topics, services) and group them by adopting standard prefixes/suffixes.**

   The architecture establishes a standardized name convention. Any package is named "turtle-bot3 packagename", with the corresponding node being named after the package name. Moreover, the entry points in the respective setup.py files follow the same name convention, resulting in rather straightforward terminal commands (e.g. ros2 run turtle- bot3 hardware control hardware control) ist sowas interessant? ich weiß es nicht The service "Robot Control" can be rather vaguely interpreted and is subject to change.

5. **Nodes directly interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system**

   When moving from simulator to hardware devices, the user only has to change a single flag in the bringup. Namely, the SLAM Toolbox expect a a boolean parameter "use sim time", which is set to true in simulation. Other than that, the system is indifferent to the usage of simulation or hardware application.

6. **ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.**

   As expected, the system encounters a rather small dataload from the lidar sensor. Furthermore, this dataload is comparably consistent. The absence of big sensor data has lead to little emphasis on this guideline. Therefore any addition to this system that increases sensor load needs to be tested thoroughly. Since the Obstacle Avoidance has its own managed node, faces a very small data load and operates on highest possible priority, the architectures robustness to system-level failure is supposedly quite high.

7. **Avoid persisting raw data if only part of it will be used.**

   Apart from the navgiational map created by the SLAM Toolbox and any data created for user visualization, no sensor data is persisted. Furthermore, any outdated map is deleted instantly from the system by the map change detection node.

Overall, the architecture adheres quite well to the guidelines wherever possible, apart from Guideline 6 which fell a little bit out of scope. Obviously, given the short time span of this project, there is always room to improve. Nontheless, the architecture is robust, performant and easily adaptable for other roboticists, which was one of the primary concerns of the study by Malavolta et al.

## 5 Ablation Study

In order to further discuss the robustness of the architecture, an ablation study was conducted (see Figure 9). The effect of an omitted layer on the other modules is categorized into three cases. Firstly, a layer can be completely unaffected, resulting in no changes to performance or robustness. Secondly, a layer can be indirectly affected by changes in other layers—meaning the layer itself does not directly access the contents or functionality provided by the omitted package but faces secondary consequences, such as an incomplete mapping of the environment. Thirdly, directly affected layers explicitly access the ommited layer and face severe repercussions in terms of unmet dependencies, unavailable services or other integral parts of the architecture. On the one hand, the omission of higher layers [1] has no effect on the levels below it, which accounts for the lower triangular matrix of *unaffected layers*. On the other hand, the removal of lower layers has varying impact on the levels above it.

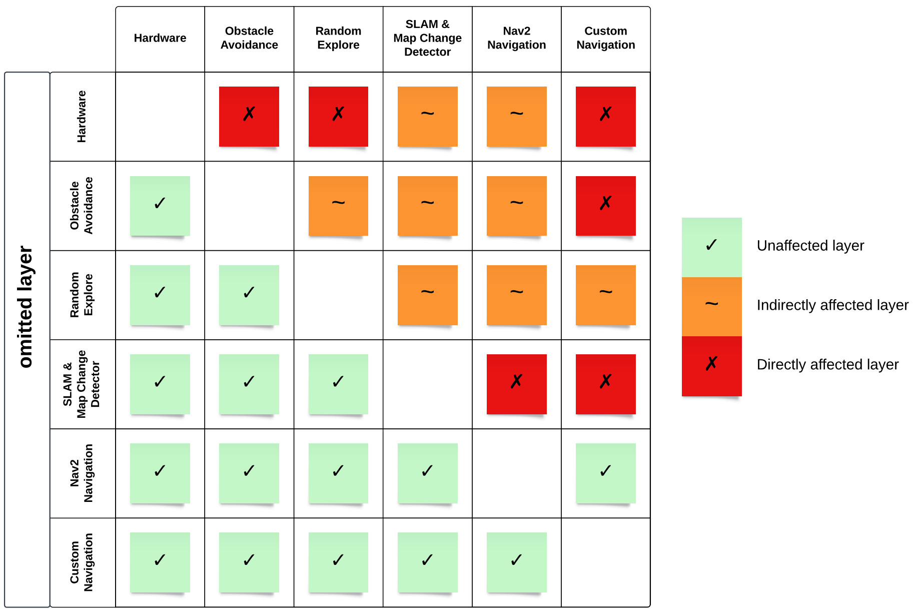
As expected, the ablation of the Hardware Control layer has a direct negative impact on all layers that rely on it to invoke movement, namely Obstacle Avoidance, Random Explore and the Custom Navigation. It also indirectly affects layers that rely on the robot being moved around (SLAM layer) and therefore by extension also the Navigation using Nav2, even though it overwrites any other movement commands and controls the robot on its own.

As a result of ommiting the Obstacle Avoidance module, the Random Explore layer cannot safely navigate the robot in an unknown environment, as the robot is very likely to crash into the first obstacle in its path. This culminates in an unfinished map, making any kind of navigation obsolete. The Custom Navigation also continuously accesses certain flags set in the Obstacle Avoidance package and is therefore directly affected.

The ablation of the Random Explore layer inherits the same problems we just described, with the exception that the Custom Navigation is not directly accessing said layer.

Clearly, the exclusion of the SLAM and Map Change Detector layer results in both navigational systems failing on multiple levels: Not only is no data being published to the /map topic, that both navigation systems rely on, but also are both packages never invoked in the first place. Lastly, it is evident that the two navigational packages do not interfere with one another and can functionally coexist - provided a given priority between the two navigational system is established. This means they are practically interchangeable and therefore qualify as a unaffected layer.

[1] *Higher* and *lower* layers are to be interpreted as shown in Figure 8



| omitted layer | Hardware | Obstacle Avoidance | Random Explore | SLAM & Map Change Detector | Nav2 Navigation | Custom Navigation |
|---|---|---|---|---|---|---|
| Hardware | | ✗ | ✗ | ~ | ~ | ✗ |
| Obstacle Avoidance | ✓ | | ~ | ~ | ~ | ✗ |
| Random Explore | ✓ | ✓ | | ~ | ~ | ~ |
| SLAM & Map Change Detector | ✓ | ✓ | ✓ | | ✗ | ✗ |
| Nav2 Navigation | ✓ | ✓ | ✓ | ✓ | | ✓ |
| Custom Navigation | ✓ | ✓ | ✓ | ✓ | ✓ | |

| | |
|---|---|
| ✓ | Unaffected layer |
| ~ | Indirectly affected layer |
| ✗ | Directly affected layer |

Figure 9: Ablation Study on layer functionality

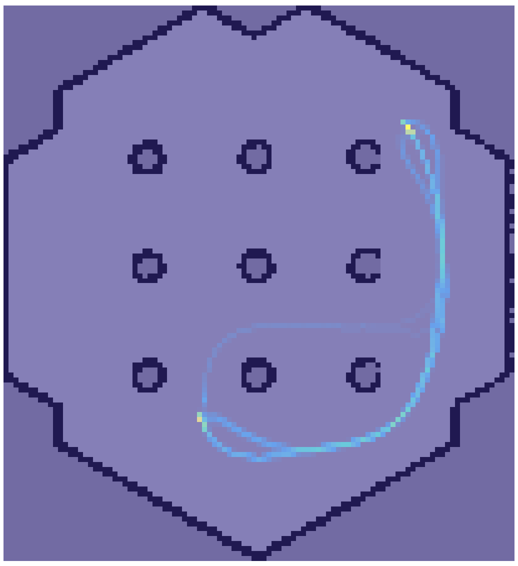
It becomes evident that the lower levels are vital for the performance and robustness of the architecture, while higher layers are more likely to have less devastating effects on overall stability or simply enhance the system's functionality. This additive nature results in a more adaptive and flexible design, as higher-level modules can easily be replaced by others with similar purposes. In contrast while examining the upper triangular matrix of figure 9, it is inevitable to mention the dependency of the higher levels on the lower layers. This interdependence limits the overall flexibility of the architecture by how resilient and adaptable the lower layers are designed.

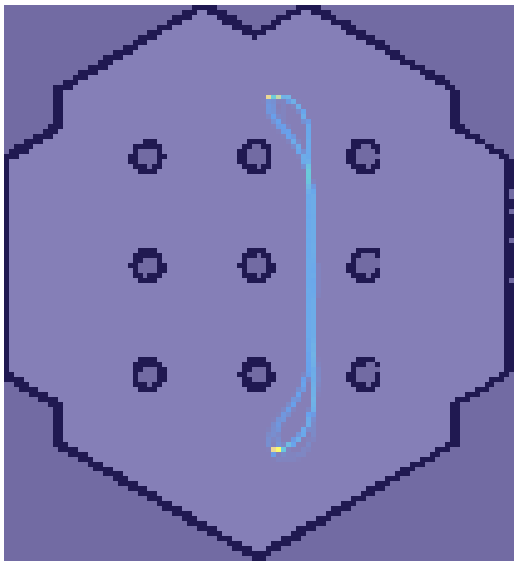
# 6 Quantitative Analysis

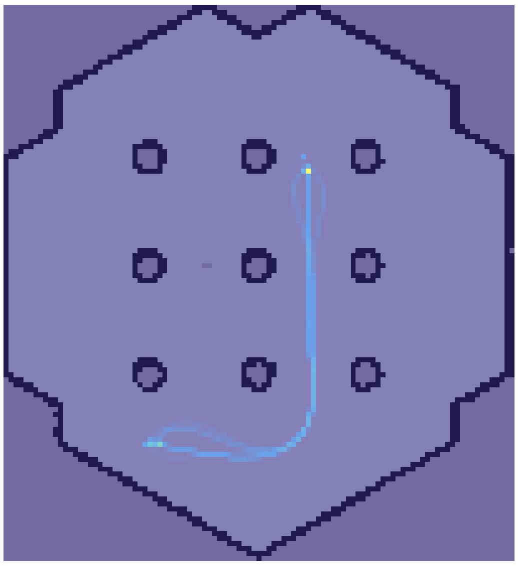
Performing quantitative research on a layered architecture for robotic systems is relatively limited by the quantifiability of certain layers. I therefore analyzed the performance of the two navigational systems. This analysis is mainly conducted in Gazebo and RViz, the prevalent simulation environments for robotic systems and ROS based applications. Then I will move on to present the application of the architecture on the real Turtlebot3 platform.

## 6.1 Simulation

The robot was provided with four distinct coordinate sets to navigate between. Each tuple of coordinates was targeted 20 times. While moving, two key metrics were collected: Distance traveled and the time used to get there. The full datasets for each iteration can be seen in the Appendix(2, 3, 4, and 5)


(a) Coordinate Set 1


(b) Coordinate Set 2


(c) Coordinate Set 3

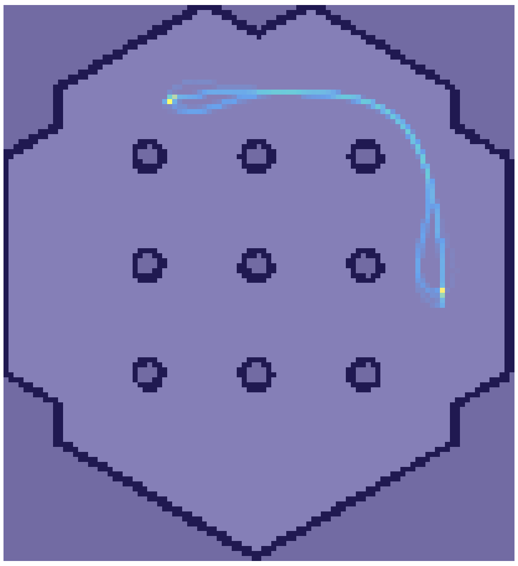

(d) Coordinate Set 4

Figure 10: Heatmaps using Nav2 package for all Coordinate Sets (n=20)

| Coordinate Set | Mean Distance Traveled | Variance Distance Traveled | Mean Time Used (s) | Variance Time Used (s) |
|---|---|---|---|---|
| 0 | 5.306683 | 0.148930 | 27.932736 | 2.545896 |
| 1 | 4.065978 | 0.222748 | 21.998405 | 1.444408 |
| 2 | 4.395432 | 0.149748 | 22.895346 | 1.698574 |
| 3 | 4.173309 | 0.148584 | 22.000601 | 1.441569 |

Table 1: Summary Statistic for Navigation using Nav2 package (n=20)

### 6.2 Simulation

### 6.3 Real World Application

## 7 Discussion

## 8 References

## References

[1] R. Brooks. "A Robust Layered Control System for a Mobile Robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23. ISSN: 0882-4967. DOI: 10.1109/JRA.1986.1087032. URL: http://ieeexplore.ieee.org/document/1087032/ (visited on 11/06/2023).

[2] E. Coste-Maniere and R. Simmons. "Architecture, the Backbone of Robotic Systems". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. 2000 ICRA. IEEE International Conference on Robotics and Automation. Vol. 1. San Francisco, CA, USA: IEEE, 2000, pp. 67–72. ISBN: 978-0-7803-5886-7. DOI: 10.1109/ROBOT.2000.844041. URL: http://ieeexplore.ieee.org/document/844041/ (visited on 03/07/2024).

[3] Jun-Young Jung et al. "Three-Layered Hybrid Architecture for Emotional Reactive System". In: *2008 International Conference on Control, Automation and Systems*. 2008 International Conference on Control, Automation and Systems (ICCAS). Seoul, South Korea: IEEE, Oct. 2008, pp. 2728–2731. ISBN: 978-89-950038-9-3 978-89-93215-01-4. DOI: 10.1109/ICCAS.2008.4694221. URL: http://ieeexplore.ieee.org/document/4694221/ (visited on 03/10/2024).

[4] Iuliia Kotseruba and John K. Tsotsos. "40 Years of Cognitive Architectures: Core Cognitive Abilities and Practical Applications". In: *Artificial Intelligence Review* 53.1 (Jan. 2020), pp. 17–94. ISSN: 0269-2821, 1573-7462. DOI: 10.1007/s10462-018-9646-y. URL: http://link.springer.com/10.1007/s10462-018-9646-y (visited on 03/07/2024).

[5] Steve Macenski et al. "The Marathon 2: A Navigation System". Version 2. In: (2020). DOI: 10.48550/ARXIV.2003.00368. URL: https://arxiv.org/abs/2003.00368 (visited on 05/08/2024).

[6] Steven Macenski et al. "Robot Operating System 2: Design, Architecture, and Uses in the Wild". In: *Science Robotics* 7.66 (May 11, 2022), eabm6074. ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/10.1126/scirobotics.abm6074 (visited on 11/05/2023).

[7] Ivano Malavolta et al. “How Do You Architect Your Robots?: State of the Practice and Guidelines for ROS-based Systems”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice.* ICSE ’20: 42nd International Conference on Software Engineering. Seoul South Korea: ACM, June 27, 2020, pp. 31–40. ISBN: 978-1-4503-7123-0. DOI: 10.1145/3377813.3381358. URL: https://dl.acm.org/doi/10.1145/3377813.3381358 (visited on 11/05/2023).

[8] R. Peter Bonasso et al. “Experiences with an Architecture for Intelligent, Reactive Agents”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (Apr. 1997), pp. 237–256. ISSN: 0952-813X, 1362-3079. DOI: 10.1080/095281397147103. URL: http://www.tandfonline.com/doi/abs/10.1080/095281397147103 (visited on 03/10/2024).

[9] Morgan Quigley et al. “ROS: An Open-Source Robot Operating System”. In: *ICRA workshop on open source software* 3.3.2 (May 2009), p. 5.

[10] Niryo Robotics. *Get Started with Niryo One ROS Stack*. Mar. 8, 2024. URL: https://niryo.com/docs/niryo-one/developer-tutorials/get-started-with-the-niryo-one-ros-stack/.

[11] Niryo Robotics. *Niryo One ROS*. Mar. 8, 2024. URL: https://github.com/NiryoRobotics/niryo_one_ros.

[12] ROS.org. *Joint Trajectory Controller*. Mar. 5, 2024. URL: http://wiki.ros.org/joint_trajectory_controller.

[13] ROS.org. *LaserScan Message*. Mar. 5, 2024. URL: http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html.

[14] ROS.org. *MoveIt Concepts*. Mar. 5, 2024. URL: https://moveit.ros.org/documentation/concepts/.

[15] ROS.org. *ROS Control*. Mar. 5, 2024. URL: http://wiki.ros.org/ros_control.

[16] Malte Schilling. “Autonome Systeme Und Mobile Roboter: Architectures” (Universität Münster). June 13, 2023. URL: https://www.uni-muenster.de/LearnWeb/learnweb2/course/view.php?id=69302.

[17] Bruno Siciliano and Oussama Khatib, eds. *Springer Handbook of Robotics*. Springer Handbooks. Cham: Springer International Publishing, 2016. 285-291. ISBN: 978-3-319-32550-7 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1. URL: https://link.springer.com/10.1007/978-3-319-32552-1 (visited on 03/08/2024).

[18] Reid Simmons et al. “A Layered Architecture for Coordination of Mobile Robots”. In: *Multi-Robot Systems: From Swarms to Intelligent Automata*. Ed. by Alan C. Schultz and Lynne E. Parker. Dordrecht: Springer Netherlands, 2002, pp. 103–112. ISBN: 978-90-481-6046-4 978-94-017-2376-3. DOI: 10.1007/978-94-017-2376-3_11. URL: http://link.springer.com/10.1007/978-94-017-2376-3_11 (visited on 03/10/2024).

[19] R. Volpe et al. “The CLARAty Architecture for Robotic Autonomy”. In: *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*. 2001 IEEE Aerospace Conference Proceedings. Vol. 1. Big Sky, MT, USA: IEEE, 2001, pp. 1/121–1/132. ISBN: 978-0-7803-6599-5. DOI: 10.1109/AERO.2001.931701. URL: http://ieeexplore.ieee.org/document/931701/ (visited on 03/08/2024).

## 9 Appendix

| Coordinate Set | Distance Traveled | Time Used (s) |
|---|---|---|
| 1 | 5.130298 | 27.998535 |
| 1 | 4.869894 | 26.000022 |
| 1 | 5.453286 | 26.000290 |
| 1 | 4.974142 | 26.000858 |
| 1 | 5.387319 | 25.999217 |
| 1 | 5.434525 | 27.999560 |
| 1 | 5.122722 | 26.000072 |
| 1 | 5.448368 | 28.000674 |
| 1 | 5.180251 | 25.999200 |
| 1 | 5.629328 | 29.999985 |
| 1 | 5.352114 | 28.000899 |
| 1 | 5.404368 | 28.000448 |
| 1 | 5.164623 | 25.998860 |
| 1 | 5.398202 | 27.974977 |
| 1 | 5.169252 | 26.025875 |
| 1 | 5.619464 | 29.999127 |
| 1 | 5.327660 | 27.999871 |
| 1 | 5.592663 | 29.760104 |
| 1 | 5.213084 | 26.239961 |

Table 2: Nav2 Data - Coordinate Set 1

| Coordinate Set | Distance Traveled | Time Used (s) |
|---|---|---|
| 2 | 5.311640 | 27.996754 |
| 2 | 4.080332 | 22.000514 |
| 2 | 3.818398 | 19.998617 |
| 2 | 3.761347 | 19.999943 |
| 2 | 3.828356 | 19.999907 |
| 2 | 3.963116 | 22.000073 |
| 2 | 3.884740 | 19.999886 |
| 2 | 4.296424 | 24.000082 |
| 2 | 3.873013 | 20.002518 |
| 2 | 3.839976 | 19.997566 |
| 2 | 3.811117 | 20.000155 |
| 2 | 3.832355 | 20.000026 |
| 2 | 3.866210 | 19.999748 |
| 2 | 3.848264 | 20.001474 |
| 2 | 3.854485 | 19.998312 |
| 2 | 3.844553 | 20.001375 |
| 2 | 3.891537 | 19.998770 |
| 2 | 3.788880 | 20.000184 |
| 2 | 3.852515 | 19.999944 |

Table 3: Nav2 Data - Coordinate Set 2

| Coordinate Set | Distance Traveled | Time Used (s) |
|---|---|---|
| 3 | 7.566510 | 39.683514 |
| 3 | 4.266853 | 22.316174 |
| 3 | 4.407925 | 22.550062 |
| 3 | 4.395692 | 23.449422 |
| 3 | 4.502186 | 23.999573 |
| 3 | 4.534671 | 23.999918 |
| 3 | 4.417903 | 24.000293 |
| 3 | 4.435943 | 23.999711 |
| 3 | 4.346739 | 24.000350 |
| 3 | 4.278771 | 23.999819 |
| 3 | 4.281278 | 22.000120 |
| 3 | 4.267426 | 21.999995 |
| 3 | 4.340623 | 21.999955 |
| 3 | 4.260607 | 21.999936 |
| 3 | 4.306719 | 22.000231 |
| 3 | 4.243368 | 22.000067 |
| 3 | 4.409689 | 21.999886 |
| 3 | 4.227193 | 22.000825 |
| 3 | 4.373677 | 21.998897 |

Table 4: Nav2 Data - Coordinate Set 3

| Coordinate Set | Distance Traveled | Time Used (s) |
|---|---|---|
| 4 | 6.377680 | 31.996258 |
| 4 | 4.325679 | 22.000514 |
| 4 | 4.237888 | 19.998617 |
| 4 | 3.761347 | 19.999943 |
| 4 | 4.828356 | 19.999907 |
| 4 | 4.963116 | 22.000073 |
| 4 | 4.884740 | 19.999886 |
| 4 | 4.296424 | 24.000082 |
| 4 | 3.873013 | 20.002518 |
| 4 | 3.839976 | 19.997566 |
| 4 | 3.811117 | 20.000155 |
| 4 | 3.832355 | 20.000026 |
| 4 | 3.866210 | 19.999748 |
| 4 | 3.848264 | 20.001474 |
| 4 | 3.854485 | 19.998312 |
| 4 | 3.844553 | 20.001375 |
| 4 | 3.891537 | 19.998770 |
| 4 | 3.788880 | 20.000184 |
| 4 | 3.852515 | 19.999944 |

Table 5: Nav2 Data - Coordinate Set 4