POLITECNICO DI TORINO

Master's Degree in AUTOMOTIVE ENGINEERING

Master's Degree Thesis

GPS-Driven Autonomous Navigation for Precision Agriculture with AWS

Supervisor

Prof. Guido ALBERTENGO

Candidate

Marco CAMPINI

July 2024

Abstract

This thesis presents the development of a GPS-driven autonomous navigation system designed for precision agriculture, specifically targeting vineyard management. Conducted in collaboration with beSharp spa and commissioned by Q8 Italia, the project aims to enhance the efficiency and accuracy of vineyard operations by enabling a self-driving robot to autonomously navigate and detect grapevine diseases. Leveraging the capabilities of Amazon Web Services (AWS), the system offers scalable, cost-effective, and reliable real-time navigation solutions. The research focuses on the architectural design, implementation process, and challenges encountered during development. By integrating advanced robotics and cloud computing technologies, this work contributes significantly to the field of precision agriculture, setting the foundation for future advancements in autonomous agricultural robotics.

The thesis is organized into four main chapters: an introduction to the research objectives and the state of the art in robot navigation, a detailed discussion of the software architecture, the development and results of simulation and implementation, and a conclusion outlining experimental results and future work. This comprehensive approach not only demonstrates the practical application of the developed system but also provides valuable insights for ongoing and future research in precision agriculture.

Acknowledgements

I would like to express my deepest gratitude to my professor, whose guidance, support, and availability have been invaluable throughout the entirety of this research project.

I am also immensely grateful to beSharp spa and Q8 Italia for providing me with the opportunity to work on such an impactful project. Their support and collaboration have been fundamental to this work, and I appreciate the resources and knowledge they have generously shared.

Lastly, I want to extend my heartfelt thanks to my family. Their unwavering support and belief in me have been a constant source of motivation and strength.

Thank you all.

Table of Contents

Chapter 1

Introduction

Agriculture is one of the fundamental activities of the human society, as it provides bio-fuels and, most importantly, food. To face the challenges and demands of a continuously growing population and the escalating environmental concerns, the agricultural sector is adopting more and more technology-driven solutions to improve the efficiency, productivity and sustainability of farming. The latest approach to agricultural management that exploits advanced technology is known as precision farming, also known as precision agriculture. "Precision agriculture (PA) is a methodology to farm management that uses information technology (IT) to certify that the crops and soil obtain exactly what they require for optimum health and efficiency" [1].

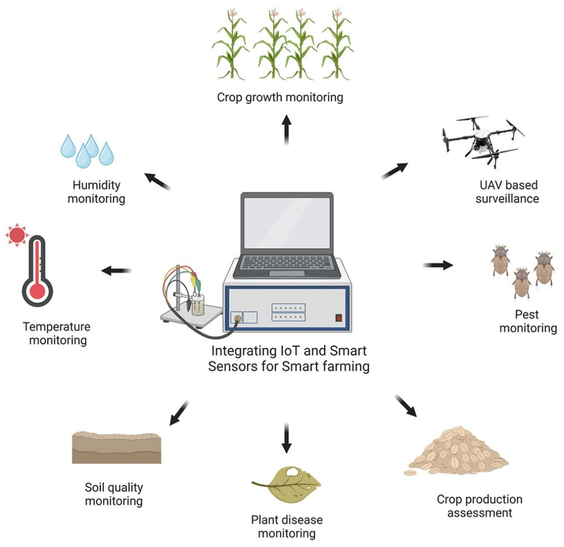


Figure 1.1: Applications of integrated IoT and smart sensors for precision farming [2]

1.1 Objective of the Thesis

This Master's thesis focuses on the development of a GPS-driven autonomous navigation system tailored for precision agriculture, leveraging the robust capabilities of Amazon Web Services (AWS). This research has been conducted in collaboration with the tech consulting company **beSharp spa**, marking the initial phase of a broader and more intricate project commissioned by one of their clients, **Q8 Italia**. The ultimate goal of the comprehensive project is to create a sophisticated self-driving robot that can autonomously navigate through vineyards and accurately detect the presence of specific diseases on grapevines. Achieving this involves integrating advanced technologies in robotics, data analysis, and cloud computing. In this thesis, particular emphasis is placed on the development of the autonomous navigation system, which serves as a foundational component of the overall project. By utilizing AWS, this system is designed to offer reliable, cost-efficient and scalable solutions for real-time navigation. The research details the system architecture, implementation process, and the various challenges encountered and addressed throughout the development phase. This work not only contributes to the field of precision agriculture by enhancing the efficiency and accuracy of vineyard management but also sets the stage for future advancements in autonomous agricultural robotics.

1.2 Organization of the Thesis

The thesis is divided into four chapters, each addressing a distinct aspect of the research problem and contributing to the overall understanding of the topic.

- **Chapter 1 - Introduction:** the first chapter provides an overview of the research topic, stating the research problem, objectives, and the significance of the study. It outlines the research questions and hypotheses that guide the investigation. Additionally, this chapter presents a brief summary of the methodology and the structure of the thesis. Finally, the state of the art in robot navigation is presented.
- **Chapter 2 - Software Architecture:** the second chapter delves into the software architecture relevant to the research problem. It provides a comprehensive overview of the architectural design, including the components, their interactions, and the rationale behind architectural decisions. It discusses different architectural styles and patterns that are applicable and how they influence the overall system design. The chapter also addresses issues such as scalability, performance, and maintainability.
- **Chapter 3 - Development:** the third chapter is dedicated to presenting the main work of the thesis and the results of the research, divided into three main sections: website, simulation and robot. The website section presents the characteristics of the user interface developed to control the robot. The simulation section discusses the setup, execution, and outcomes of various simulations conducted to develop and test the code in a simpler environment. It includes detailed descriptions of the simulation environment, parameters, and results. The robot section covers the practical implementation of the proposed solutions and their validation in real-world scenarios. This section includes a description of the real robot, implementation process, challenges faced, and the results obtained from validation experiments.
- **Chapter 4 - Conclusion:** the final chapter summarizes the key findings of the research and their practical and theoretical implications. It restates the significance of the study and provides concluding remarks. Recommendations for future work are also presented, offering guidance for future researchers.

1.3 State of the Art in Robot Navigation

1.3.1 Introduction

Robot navigation is a pivotal field within robotics, involving the design and implementation of algorithms that enable robots to move autonomously within their environment. This section delves into the state-of-the-art techniques in robot navigation, highlighting the advancements and methodologies currently shaping the field.

1.3.2 Navigation Techniques

Robot navigation techniques can be broadly classified into several categories, each with its unique approach and applications. These categories include reactive navigation, deliberative navigation, hybrid approaches, and learning-based methods.

Reactive Navigation

Reactive navigation systems rely on real-time sensor data to make immediate decisions about movement, often without any global planning. These systems are typically fast and computationally efficient but may struggle with complex environments. Common techniques in reactive navigation include:

- **Potential Fields:** Robots generate a virtual field where obstacles repel and goals attract the robot. While simple to implement, potential fields can suffer from issues like local minima [3].
- **Behavior-Based Control:** This method decomposes navigation into a set of behaviors (e.g., obstacle avoidance, goal seeking) that are executed based on sensor inputs. The Subsumption Architecture is a classic example of this approach [4].

Deliberative Navigation

Deliberative navigation involves constructing a detailed model of the environment and planning paths based on this model. These systems are generally more robust in complex environments but require more computational resources. Key techniques include:

- **Graph-Based Methods:** Methods like A* and Dijkstra's algorithm use graphs to represent the environment and find optimal paths. These methods are highly effective in known environments but can be slow [3].

- **Sampling-Based Planners:** Algorithms like Rapidly-exploring Random Trees (RRT) and Probabilistic Roadmaps (PRM) are used for path planning in high-dimensional spaces. These methods are particularly useful in robotics due to their ability to handle complex spaces [4].

Hybrid Approaches

Hybrid approaches combine the strengths of reactive and deliberative methods to create systems that can handle a wider range of scenarios. These systems typically use a high-level planner to generate a global path and a low-level controller to handle local obstacles and real-time adjustments. Examples include:

- **Hierarchical Planning:** This approach uses a global planner to generate a coarse path and a local planner to refine the path in real-time [4].
- **Elastic Band Method:** This method adjusts a precomputed path dynamically as the robot encounters new obstacles, providing a balance between efficiency and robustness [3].

1.3.3 Learning-Based Navigation

Recent advancements in machine learning have significantly impacted robot navigation, enabling robots to learn from their environments and improve their performance over time. Learning-based navigation methods include:

Reinforcement Learning

Reinforcement learning (RL) allows robots to learn optimal navigation policies through trial and error. Techniques like Q-learning and Deep Q-Networks (DQN) have been applied to navigation tasks, demonstrating the ability to learn effective strategies in complex environments. Key aspects include:

- **Reward Functions:** Defining appropriate reward functions is critical for successful RL applications in navigation [5].
- **Exploration vs. Exploitation:** Balancing exploration of new paths with exploitation of known good paths is a central challenge in RL [5].

Imitation Learning

Imitation learning involves training robots to mimic expert demonstrations. This approach can be more sample-efficient than RL and has been used in various navigation tasks. Techniques include:

- **Behavior Cloning:** Directly mapping states to actions based on expert data [5].
- **Inverse Reinforcement Learning:** Inferring the reward function from expert demonstrations and using it to guide the robot's actions [5].

1.3.4 Simultaneous Localization and Mapping (SLAM)

SLAM is a foundational technology in robot navigation, enabling robots to build a map of their environment while simultaneously localizing themselves within it. Modern SLAM techniques integrate various sensor modalities and leverage advanced algorithms:

Visual SLAM

Visual SLAM uses cameras to capture images of the environment and extract features for mapping and localization. Techniques include:

- **ORB-SLAM:** A robust and versatile visual SLAM system using Oriented FAST and Rotated BRIEF (ORB) features [6].
- **Direct Methods:** These methods, such as LSD-SLAM, use pixel intensities directly rather than feature points, offering advantages in certain scenarios [7].

Lidar-Based SLAM

Lidar-based SLAM utilizes laser scanners to generate highly accurate maps. These systems are particularly effective in environments with rich geometric features. Prominent methods include:

- **Hector SLAM:** Uses scan matching to estimate the robot's pose in real-time [8].
- **Cartographer:** Developed by Google, this method combines real-time constraints with global optimization to produce detailed maps [9].

Chapter 2

Software Architecture

2.1 ROS 2

2.1.1 Introduction

Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [10]. ROS 2 is the next generation of ROS, designed to address limitations of ROS 1 and to add features required for production-grade use cases [11]. This section provides an in-depth look into ROS 2, its architecture, features, and applications.

2.1.2 Background and Motivation

The original ROS, often referred to as ROS 1, was introduced in 2007. Over the years, it has been widely adopted by academia and industry for research and development [10]. However, ROS 1 has several limitations, particularly in terms of real-time capabilities, security, and support for multi-robot systems [11]. These limitations necessitated the development of ROS 2, which aims to provide a more robust and scalable solution suitable for commercial applications [11].

2.1.3 Architecture of ROS 2

ROS 2 retains the core concepts of ROS 1, such as nodes, topics, services, and messages, but it introduces significant changes to the underlying architecture to enhance performance, reliability, and flexibility [11].

Data Distribution Service (DDS)

One of the key architectural changes in ROS 2 is the adoption of the Data Distribution Service (DDS) standard for its communication layer. DDS is a middleware protocol and API standard for data-centric connectivity from the Object Management Group (OMG) [12]. It provides:

- **Real-time Communication:** DDS supports real-time, low-latency communication, which is essential for robotics applications [12].
- **Quality of Service (QoS):** DDS allows fine-grained control over communication parameters such as reliability, durability, and deadline [12].
- **Scalability:** DDS is designed to work efficiently in large and dynamic networks, making it suitable for multi-robot systems [12].
- **Interoperability:** As a standard, DDS enables interoperability between different implementations and platforms [12].

Node and Executor Model

In ROS 2, the concept of nodes remains central. A node is a process that performs computation. Nodes can communicate with each other using topics, services, and actions. ROS 2 introduces the executor model, which is responsible for managing the execution of callbacks for incoming messages and service requests [11]. This model provides:

- **Flexibility:** Different executors can be implemented to suit specific needs, such as real-time constraints or custom scheduling policies [11].
- **Efficiency:** The executor model helps in optimizing the use of system resources by managing the lifecycle and execution context of nodes [11].

Lifecycle Management

ROS 2 introduces a managed lifecycle for nodes, which allows greater control over the state of nodes. The lifecycle states include unconfigured, inactive, active, and finalized [13]. This feature is particularly useful for systems that require deterministic startup and shutdown sequences, ensuring that nodes are correctly initialized and cleaned up [13].

2.1.4 Key Features of ROS 2

Security

Security is a critical requirement for modern robotic systems, especially those used in sensitive or commercial applications. ROS 2 incorporates security features such as [14]:

- **Authentication:** Ensuring that only authorized nodes can communicate [14].
- **Encryption:** Protecting data in transit between nodes [14].
- **Access Control:** Defining policies for what each node can do, including publish/subscribe permissions [14].

Real-time Capabilities

ROS 2 is designed with real-time performance in mind. It leverages DDS's QoS settings to meet the requirements of real-time applications [12]. Additionally, ROS 2 allows the use of real-time operating systems (RTOS) and custom real-time scheduling policies to achieve deterministic behavior [11].

Multi-robot Systems

Supporting multi-robot systems is a significant focus of ROS 2. It enables seamless communication and coordination between multiple robots, which is crucial for collaborative tasks [11]. The use of DDS facilitates efficient data exchange in distributed environments [12].

Cross-platform Support

ROS 2 is designed to be platform-agnostic, supporting a wide range of operating systems, including Linux, Windows, and macOS [11]. This cross-platform compatibility ensures that ROS 2 can be used in diverse development environments and deployed on various hardware platforms [11].

2.1.5 Development and Deployment Tools

Colcon

Colcon is the build tool used in ROS 2, replacing the ROS 1 catkin tool [15]. It provides a more flexible and efficient way to build packages and manage dependencies [15]. Colcon supports parallel builds and can handle large codebases more effectively [15].

ROS 2 Launch System

The ROS 2 launch system allows for the configuration and deployment of complex systems [16]. It provides tools for launching multiple nodes, configuring parameters, and managing the lifecycle of nodes [16]. The launch system supports both Python and XML, offering flexibility in how launch files are written and executed [16].

Simulation

Simulation is a crucial aspect of developing and testing robotic systems. ROS 2 supports various simulation tools, including Gazebo and Ignition [17, 18]. These tools allow developers to create realistic environments and simulate the behavior of robots before deploying them in the real world [17, 18].

2.1.6 Applications of ROS 2

Autonomous Vehicles

ROS 2 is widely used in the development of autonomous vehicles [19]. Its real-time capabilities, robust communication, and security features make it suitable for applications where safety and reliability are paramount [19]. Companies in the automotive industry are adopting ROS 2 for prototyping and deploying autonomous driving solutions [19].

Industrial Automation

In industrial automation, ROS 2 is used to control robotic arms, AGVs (Automated Guided Vehicles), and other machinery [20]. Its support for multi-robot systems and real-time communication ensures efficient and coordinated operation in industrial settings [20].

Healthcare Robotics

Healthcare robotics, including surgical robots and assistive devices, benefit from ROS 2's precision and reliability [21]. ROS 2 enables the integration of various sensors and actuators, providing a robust framework for developing complex healthcare applications [21].

Research and Education

ROS 2 continues to be a valuable tool in research and education [22]. Universities and research institutions use ROS 2 to teach robotics concepts and conduct cutting-edge research [22]. Its open-source nature and active community support contribute to its widespread adoption in academia [22].

2.2 Cloud Computing

2.2.1 Introduction to Cloud Computing

Cloud computing is a transformative model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources, such as networks, servers, storage, applications, and services, that can be rapidly provisioned and released with minimal management effort or service provider interaction [23]. This model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.

The concept of cloud computing has evolved over several decades. The term "cloud" is used as a metaphor for the Internet, based on the cloud drawing used in the past to represent the telephone network, and later to depict the Internet in computer network diagrams. The advent of virtualization, web 2.0, and advances in networking have all contributed to the rise of cloud computing.

2.2.2 Characteristics of Cloud Computing

The essential characteristics of cloud computing include:

On-demand Self-service

Users can automatically provision computing capabilities, such as server time and network storage, as needed, without requiring human interaction with each service provider [23]. This characteristic is crucial as it reduces the time needed to deploy new applications and services, thereby increasing efficiency and responsiveness. For example, a developer can quickly deploy a virtual server without waiting for IT to provide the necessary infrastructure.

Broad Network Access

Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms, such as mobile phones, tablets, laptops, and workstations [23]. This ensures that services are accessible from anywhere at any time, provided there is internet connectivity. An example is accessing cloud-based email services like Gmail from any device. This characteristic supports the growing trend of remote work and global collaboration.

Resource Pooling

The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [23]. This creates an economy of scale, as resources are utilized more efficiently. For instance, cloud providers like Amazon Web Services (AWS) can host thousands of virtual servers on fewer physical machines, optimizing resource usage. The pooling of resources helps in cost reduction and resource efficiency.

Rapid Elasticity

Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand [23]. This elasticity is one of the key benefits of cloud computing, allowing businesses to handle varying workloads without investing in permanent infrastructure. An example is an e-commerce website scaling its resources during peak shopping seasons. Elasticity ensures that businesses can manage load spikes effectively without compromising performance.

Measured Service

Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts) [23]. This provides transparency for both the provider and consumer of the utilized service. For instance, a business can monitor its cloud usage and only pay for what it consumes, similar to utility billing. This characteristic enables detailed monitoring and management of resource usage.

2.2.3 Service Models

Cloud computing is commonly classified into three service models:

Infrastructure as a Service (IaaS)

IaaS provides virtualized computing resources over the internet. It serves as the basic physical infrastructure and includes virtual machines, storage, and networks [23]. IaaS is highly scalable and can be adjusted according to the demand. Examples of IaaS providers include AWS EC2, Google Compute Engine, and Microsoft Azure VMs. These services allow businesses to rent virtual servers and storage space, reducing the need for physical hardware. IaaS provides the foundational building blocks for cloud IT and typically includes access to networking features, computers (virtual or on dedicated hardware), and data storage space.

Platform as a Service (PaaS)

PaaS provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the underlying infrastructure typically associated with developing and launching an app [23]. This model supports the complete application lifecycle, including building, testing, deploying, managing, and updating. Examples of PaaS include Google App Engine, Microsoft Azure App Services, and Heroku. These platforms simplify the development process by providing pre-configured environments. PaaS can improve the speed of development of applications, and the integration of databases and other services.

Software as a Service (SaaS)

SaaS allows users to connect to and use cloud-based apps over the Internet. Common examples are email, calendaring, and office tools (such as Microsoft Office 365) [23]. SaaS removes the need for organizations to install and run applications on their computers or in their data centers, which simplifies maintenance and support. Examples include Salesforce, Google Workspace, and Dropbox. These services are subscription-based and accessible via web browsers. SaaS applications are accessed by users via a web browser, and updates and maintenance are managed by the SaaS provider.

2.2.4 Deployment Models

The four primary cloud deployment models include:

Public Cloud

Public clouds are owned and operated by third-party cloud service providers, delivering their computing resources like servers and storage over the Internet [23]. The major public cloud providers include AWS, Microsoft Azure, and Google Cloud Platform. These services offer vast scalability and pay-as-you-go pricing models, making them accessible to businesses of all sizes. Public clouds are suitable for non-sensitive, high-volume workloads where cost and scalability are primary concerns. The infrastructure and services are provided on a shared platform, and the data security and management policies are controlled by the provider.

Private Cloud

A private cloud refers to cloud computing resources used exclusively by a single business or organization. A private cloud can be physically located at your organization's on-site datacenter or hosted by a third-party service provider [23]. Private clouds provide greater control over data, security, and compliance, making them ideal for industries with strict regulatory requirements. Technologies such as VMware, OpenStack, and Microsoft Azure Stack enable the creation of private clouds. Private clouds are customizable and can be tailored to the specific needs of an organization.

Hybrid Cloud

Hybrid clouds combine public and private clouds, bound together by technology that allows data and applications to be shared between them [23]. This approach offers greater flexibility and optimization of existing infrastructure, security, and compliance. For example, a business can use the public cloud for high-volume, low-security needs such as web-based email, and the private cloud for sensitive, business-critical operations like financial reporting. Hybrid cloud solutions are becoming increasingly popular as they provide the best of both worlds. The hybrid model allows for data and applications to move between private and public clouds, offering greater flexibility and more deployment options.

Community Cloud

A community cloud is shared between several organizations from a specific group with common computing concerns (e.g., mission, security requirements, policy, and compliance considerations) [23]. Community clouds can be managed internally or by a third-party and hosted internally or externally. An example is a government cloud used by multiple government departments that need to adhere to similar security standards and compliance requirements. Community clouds provide tailored services that cater to the specific needs of the user group. This model is particularly useful for collaborative projects and joint ventures.

2.2.5 Benefits of Cloud Computing

The advantages of cloud computing include cost efficiency, scalability, flexibility, and disaster recovery capabilities. Cloud services eliminate the capital expense of buying hardware and software and setting up and running on-site datacenters [24].

Cost Efficiency

One of the most significant benefits of cloud computing is cost savings. Businesses can reduce their capital expenditure (CapEx) and operational expenditure (OpEx) by moving to the cloud. The pay-as-you-go model allows organizations to pay only for the resources they use, which can lead to significant savings, especially for small and medium-sized enterprises (SMEs). Moreover, cloud providers often offer discounts for long-term usage and reserved instances. This financial flexibility helps organizations to manage budgets more effectively and invest in other areas of growth.

Scalability and Flexibility

Cloud computing offers unmatched scalability and flexibility. Businesses can scale their resources up or down based on demand, ensuring they only pay for what they use. This is particularly beneficial for businesses with fluctuating workloads, such as e-commerce websites during peak shopping seasons or startups experiencing rapid growth. Additionally, cloud services can be accessed from anywhere, providing flexibility for remote work and global operations. This dynamic scaling ensures that applications can handle peak loads and unforeseen demands efficiently.

Disaster Recovery

Disaster recovery is another critical advantage of cloud computing. Cloud providers offer robust backup and recovery solutions that ensure business continuity in the event of data loss, hardware failure, or natural disasters. Services like AWS Disaster Recovery, Azure Site Recovery, and Google Cloud Disaster Recovery provide automated backups, real-time data replication, and fast recovery times. These services can be more cost-effective and reliable than traditional on-premises disaster recovery solutions. Having a robust disaster recovery plan is essential for maintaining business operations and customer trust.

Collaboration and Productivity

Cloud computing facilitates collaboration and increases productivity by enabling real-time access to data and applications from any location. Tools like Google Workspace, Microsoft Office 365, and Slack allow team members to work together seamlessly, share files, and communicate effectively. This is especially important in today's globalized business environment, where teams are often distributed across different time zones and locations. Enhanced collaboration tools improve project management and accelerate decision-making processes.

2.2.6 Challenges in Cloud Computing

Despite the benefits, cloud computing also presents challenges such as security risks, potential downtime, and the need for internet connectivity. Ensuring the security and privacy of sensitive data stored in the cloud is a major concern [24].

Security and Privacy

Security is one of the most significant challenges in cloud computing. As data is stored off-site, there is a risk of data breaches, unauthorized access, and data loss. Cloud providers implement stringent security measures, including encryption, identity and access management (IAM), and multi-factor authentication (MFA), to protect data. However, businesses must also adopt best practices, such as regular security audits, employee training, and compliance with relevant regulations like GDPR and HIPAA. Businesses should also implement encryption and access controls to protect sensitive information.

Downtime and Availability

Downtime is another concern in cloud computing. While cloud providers offer high availability and uptime guarantees, outages can still occur due to technical issues, cyber-attacks, or natural disasters. Businesses must have contingency plans in place to mitigate the impact of downtime. This can include using multi-cloud strategies, maintaining on-premises backups, and leveraging disaster recovery services offered by cloud providers. Ensuring business continuity requires comprehensive planning and risk management strategies.

Internet Connectivity

Cloud computing relies heavily on internet connectivity. Poor or unstable internet connections can hinder access to cloud services, affecting productivity and business operations. Organizations must invest in reliable, high-speed internet connections and consider using hybrid or multi-cloud models to reduce dependency on a single provider or location. Businesses should also have backup internet connections to ensure continuous access to cloud services.

2.2.7 Amazon Web Services (AWS)

Amazon Web Services (AWS) is one of the most comprehensive and widely adopted cloud platforms in the world. AWS offers over 200 fully featured services from data centers globally [25]. AWS provides a wide range of services that cater to various use cases, from startups to large enterprises and government agencies.

AWS Services

Some of the prominent services provided by AWS include:

- **Compute**: Amazon Elastic Compute Cloud (EC2) provides scalable computing capacity in the cloud [25]. EC2 instances can be customized with various CPU, memory, and storage configurations to suit different workloads. EC2 Auto Scaling helps automatically adjust the number of instances based on demand, ensuring applications run smoothly.
- **Storage**: Amazon Simple Storage Service (S3) is an object storage service offering industry-leading scalability, data availability, security, and performance [25]. S3 is designed to store and retrieve any amount of data from anywhere. It provides 99.999999999% durability and comprehensive security and compliance capabilities.
- **Database**: Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud [25]. RDS supports several database engines, including Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle, and Microsoft SQL Server. It automates tasks such as backups, patching, and scaling.
- **Networking**: Amazon Virtual Private Cloud (VPC) lets you provision a logically isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you define [25]. VPC provides advanced security features such as network segmentation, security groups, and network access control lists (ACLs).
- **Machine Learning**: AWS offers a suite of machine learning services, including Amazon SageMaker, which enables developers and data scientists to build, train, and deploy machine learning models at scale [25]. Other services include Amazon Comprehend for natural language processing (NLP), Amazon Rekognition for image and video analysis, and Amazon Lex for building conversational interfaces.

Advantages of AWS

AWS provides several advantages such as high availability, security, flexibility, and cost-efficiency. It supports a variety of use cases, including disaster recovery, data lakes and analytics, machine learning, and more [25].

- **High Availability**: AWS data centers are built in clusters in various global regions, providing redundancy and ensuring high availability. Services like Amazon Route 53 and Elastic Load Balancing (ELB) distribute incoming application traffic across multiple targets, enhancing fault tolerance.

- **Security**: AWS complies with numerous security standards and certifications, including ISO 27001, HIPAA, SOC 1/2/3, and GDPR. AWS provides a comprehensive set of security tools and features, such as AWS Identity and Access Management (IAM), AWS Key Management Service (KMS), and AWS Shield for DDoS protection.
- **Flexibility**: AWS supports a wide range of operating systems, programming languages, and application architectures. This flexibility allows organizations to use the technologies they are most comfortable with and integrate AWS services with their existing IT infrastructure.
- **Cost Efficiency**: AWS offers a pay-as-you-go pricing model, which means businesses only pay for the resources they use. AWS also provides various pricing options, including reserved instances, spot instances, and savings plans, to help organizations optimize their costs.

Use Cases

AWS is used by a diverse set of customers, including startups, enterprises, and government agencies. Some common use cases include:

- **Web Hosting**: AWS provides scalable and reliable web hosting solutions, including static websites using Amazon S3 and dynamic websites using EC2, Elastic Beanstalk, and RDS.
- **Big Data and Analytics**: AWS offers a range of services for big data and analytics, such as Amazon Redshift for data warehousing, Amazon EMR for big data processing using Apache Hadoop and Spark, and AWS Glue for data integration.
- **DevOps**: AWS supports DevOps practices with services like AWS CodePipeline for continuous integration and delivery (CI/CD), AWS CodeBuild for building and testing code, and AWS CodeDeploy for automated deployments.
- **Internet of Things (IoT)**: AWS IoT Core enables secure device connectivity and management, allowing organizations to collect and analyze data from IoT devices at scale. Other services include AWS IoT Greengrass for edge computing and AWS IoT Analytics for processing and analyzing IoT data.

2.2.8 Future Trends in Cloud Computing

The future of cloud computing includes increased adoption of multi-cloud strategies, edge computing, and serverless architectures. Companies will continue to innovate with cloud services to stay competitive in the digital economy [24].

Multi-cloud Strategies

Many organizations are adopting multi-cloud strategies, where they use services from multiple cloud providers to avoid vendor lock-in, increase resilience, and optimize costs. This approach allows businesses to leverage the best features of each cloud provider and create a more flexible and robust IT infrastructure. By distributing workloads across multiple clouds, organizations can improve performance and reduce the risk of downtime.

Edge Computing

Edge computing involves processing data closer to the source of data generation, rather than relying on a centralized cloud infrastructure. This reduces latency and bandwidth usage, making it ideal for applications requiring real-time data processing, such as autonomous vehicles, industrial IoT, and smart cities. Cloud providers are expanding their edge computing offerings, with services like AWS Outposts, Azure Stack, and Google Anthos. Edge computing is expected to grow significantly as the number of IoT devices and real-time applications increases.

Serverless Architectures

Serverless computing allows developers to build and run applications without managing the underlying infrastructure. Services like AWS Lambda, Azure Functions, and Google Cloud Functions enable developers to focus on writing code while the cloud provider handles server provisioning, scaling, and maintenance. This approach simplifies application development and reduces operational overhead. Serverless architectures are well-suited for microservices, event-driven applications, and scenarios with unpredictable workloads.

Artificial Intelligence and Machine Learning

The integration of artificial intelligence (AI) and machine learning (ML) with cloud services is transforming various industries. Cloud providers offer AI and ML services that simplify the development and deployment of intelligent applications. Examples include AWS SageMaker, Google AI Platform, and Azure Machine Learning. These services provide pre-built algorithms, data labeling tools, and scalable infrastructure for training and deploying models. AI and ML are being used in areas such as predictive analytics, natural language processing, image recognition, and autonomous systems.

Quantum Computing

Quantum computing is an emerging technology that has the potential to solve complex problems that are beyond the capabilities of classical computers. Cloud providers are investing in quantum computing research and offering quantum computing services, such as Amazon Braket, IBM Quantum, and Microsoft Azure Quantum. These services provide access to quantum processors and simulators, enabling researchers and developers to experiment with quantum algorithms and applications. Quantum computing is expected to revolutionize fields such as cryptography, material science, and complex system simulations.

2.2.9 Project Infrastructure

The infrastructure of our project is built on Amazon Web Services (AWS), leveraging several of its services to create a robust and scalable system. The primary components of our infrastructure include Amazon S3, AWS CloudFront, AWS Lambda, Amazon DynamoDB, and the WebSocket API. Each component plays a crucial role in ensuring the seamless operation of the static website and real-time communication with the robot. This section details the architecture and functionalities of these components.

Amazon S3 and AWS CloudFront

The static website is hosted on Amazon Simple Storage Service (S3). Amazon S3 is a scalable object storage service that allows us to store and retrieve any amount of data at any time. The website content, including HTML, CSS, and JavaScript files, are stored in an S3 bucket. The bucket is configured to be publicly accessible, ensuring that users can access the website via the internet.

To enhance the performance and security of the website, we utilize AWS CloudFront, a content delivery network (CDN). CloudFront distributes the content globally through a network of edge locations, reducing latency and improving load times for users regardless of their geographic location. The S3 bucket serves as the origin for the CloudFront distribution, which caches the website content at the edge locations. The link to the distribution is the following: `https://d20ch7cibo2tal.cloudfront.net`

WebSocket API and AWS Lambda

The WebSocket API is a crucial component of our system, enabling real-time communication between the website and the robot. This API is powered by three AWS Lambda functions and Amazon API Gateway.

AWS Lambda Functions

Lambda is a serverless compute service that allows us to run code without provisioning or managing servers. We use three Lambda functions for the WebSocket API:

- **Connect Function:** This function is triggered when a user establishes a connection to the WebSocket API. It handles the initial handshake and stores the user's connection ID in a DynamoDB table.
- **Disconnect Function:** This function is triggered when a user disconnects from the WebSocket API. It removes the user's connection ID from the DynamoDB table, ensuring that only active connections are maintained.
- **Message Function:** This function is triggered when a message is sent through the WebSocket API. It processes the message and sends it to all the active connections whose ids are stored in the DynamoDB table.

The code of each lambda function is reported in Appendix A.

Amazon API Gateway

Amazon API Gateway is used to create, publish, maintain, monitor, and secure the WebSocket API. It handles the routing of messages to the appropriate Lambda functions and manages the WebSocket connections. API Gateway ensures the scalability and reliability of the WebSocket API, accommodating fluctuating numbers of concurrent connections.

Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. In our project, DynamoDB is used to store the connection IDs of active WebSocket connections. The database is designed to handle high-velocity data, making it ideal for tracking real-time connection states.

Integration and Operation

The integration of these AWS services forms a cohesive and efficient infrastructure. The static website hosted on S3 and distributed via CloudFront serves as the user interface, providing a platform for users to interact with the robot in real-time. The WebSocket API, facilitated by API Gateway and Lambda functions, enables bi-directional communication, allowing users to send GPS waypoints to the robot and receive real-time updates on its position.

The connection IDs stored in DynamoDB ensure that messages are correctly routed to active connections, maintaining the integrity and reliability of the communication channel. This architecture not only supports the current requirements of our project but also provides a scalable foundation for future enhancements and expansions.

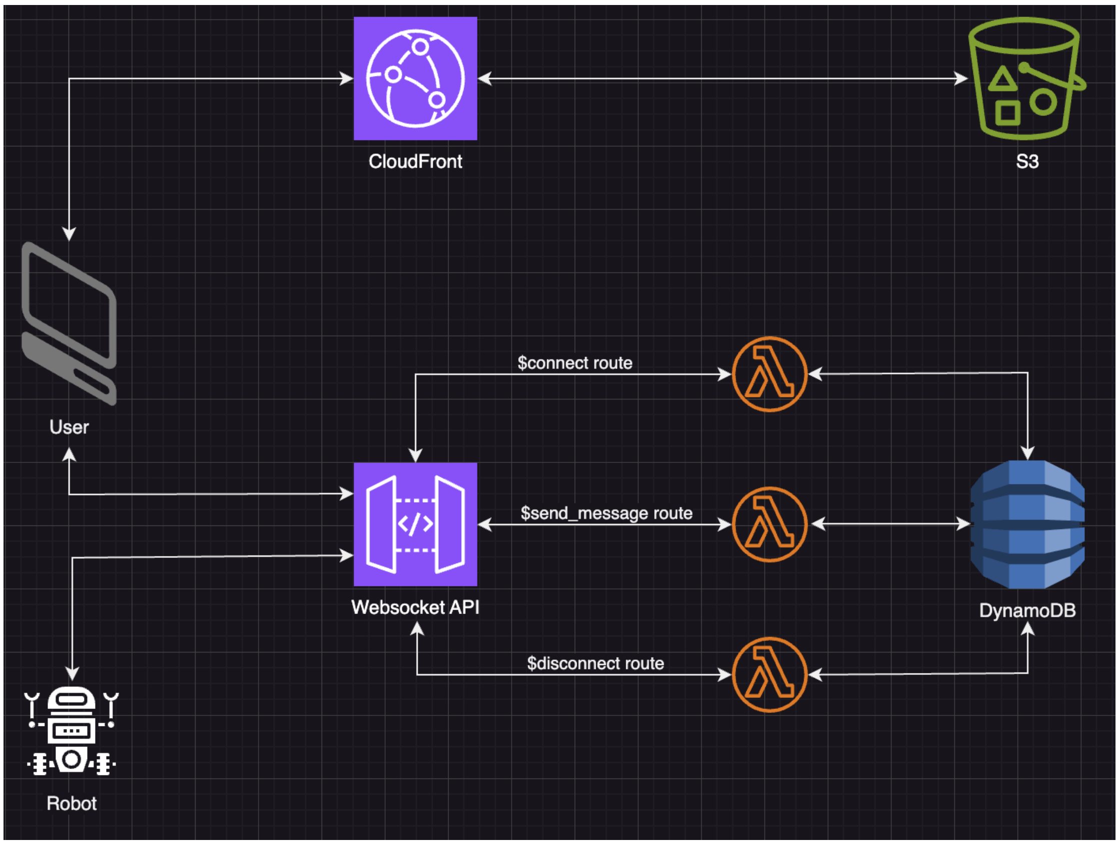


Figure 2.1: Overall AWS Infrastructure for the Project [26]

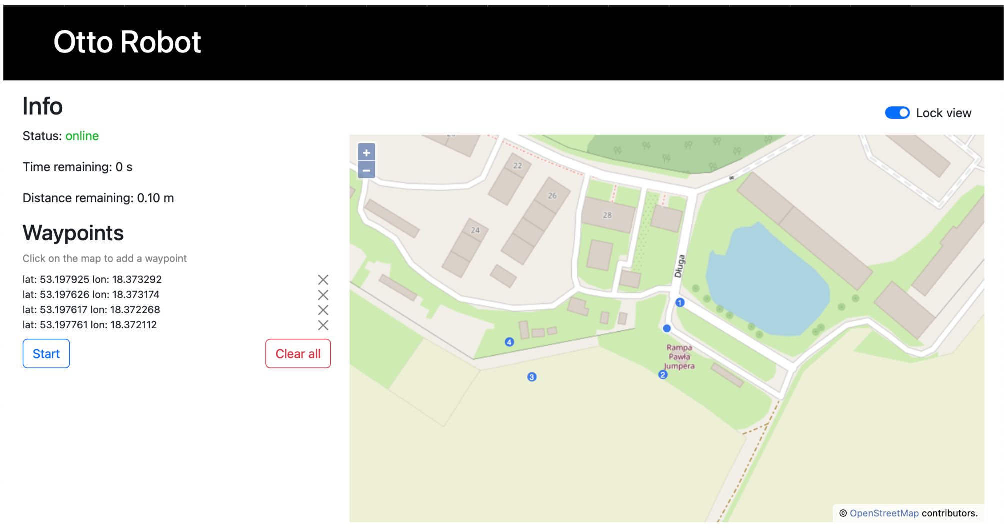
The AWS infrastructure of our project demonstrates the power and flexibility of cloud services in building scalable and efficient systems. By leveraging S3, CloudFront, Lambda, API Gateway, and DynamoDB, we have created an infrastructure that supports real-time communication and ensures a seamless user experience. This architecture not only meets the current needs of our project but also lays the groundwork for future developments and enhancements.

Chapter 3

Development

3.1 Website

In order to have a user interface to control and monitor the robot, we decided to create a static website, that is hosted on an AWS S3 bucket. In front of the bucket, we put a CloudFront distribution, which can be accessed at this link: https://d20ch7cibo2tal.cloudfront.net


Otto Robot

Info

Status: online

Time remaining: 0 s

Distance remaining: 0.10 m

Waypoints

Click on the map to add a waypoint

lat: 53.197925 lon: 18.373292
lat: 53.197626 lon: 18.373174
lat: 53.197617 lon: 18.372268
lat: 53.197761 lon: 18.372112

Figure 3.1: Website

The key features are presented below.

Status and Info Display

- The status of the robot is shown. It displays **online** while the robot sends position data through the websocket. It displays **disconnected** when the websocket connection times out. It displays **offline** if the websocket connection is up, but the robot is not sending any messages.
- It displays the time remaining and the distance remaining for the robot to reach its destination.

Map Interface

- The map shows the current position of the robot, marked with a blue dot.
- Users can click on the map to add waypoints, which are destinations the robot should navigate to.
- The map view can be zoomed in and out using the plus and minus buttons.

Waypoints Management

- There is a section labeled **Waypoints** where users can start the robot's navigation by clicking the **Start** button.
- The waypoints added on the map (blue points with white number) are sent to the robot through the websocket connection for navigation.

Lock View Feature

- There is a toggle switch labeled **Lock view**. When this is enabled, the map will automatically move to keep the robot in view as it navigates.
- If the user manually pans the map, the **Lock view** feature is automatically disabled.

Websocket Communication

- The website connects to a websocket to communicate with the robot.
- The robot sends data about its current position and navigation status back to the website in real-time.

Overall, this website allows users to interactively control the robot's movements by setting waypoints on the map and monitoring its progress. For the complete implementation code, please refer to Appendix B.

3.2 Simulation

In the development of robotic systems, simulation plays a crucial role in ensuring the feasibility and reliability of various algorithms and functionalities. For our project, we decided to commence with simulations before transitioning to the real robot for final testing. The primary reason for this approach is the significant ease and flexibility provided by simulation environments.

Simulating the robot in a controlled virtual environment allows for extensive testing and iteration without the risk of damaging hardware or encountering unpredictable real-world variables. Tools like Gazebo Ignition and RViz2 offer comprehensive platforms for creating detailed simulations, visualizing sensor data, and debugging the robot's performance. These simulations help in refining navigation algorithms, tuning control parameters, and optimizing sensor integration.

Once the simulation phase provided satisfactory results, we transitioned to deploying and testing our systems on the actual robot. The final tests in a real-world environment were essential for validating the robustness and practicality of our solutions. This staged approach ensured that our development process was efficient and minimized potential risks associated with direct hardware testing.

3.2.1 Gazebo Ignition

Gazebo Ignition is a collection of libraries designed to develop robot applications. It provides a robust framework for simulating robots in complex environments. The main features of Gazebo Ignition include:

- **Realistic Environment Simulation:** It offers high-fidelity physics simulation, accurate sensor models, and detailed 3D environments.
- **Modular Design:** Gazebo Ignition is modular, allowing users to select only the components they need.
- **Scalability:** It supports simulations from small robots to large, complex systems.
- **Integration with ROS2:** Seamlessly integrates with ROS2, enabling the use of advanced robotic algorithms and communication protocols.

Gazebo Ignition comprises several components:

Ignition Gazebo: The primary simulation server.

Ignition Physics: Handles the physics simulations.

Ignition Rendering: Manages the 3D rendering.

Ignition Sensors: Provides sensor models.

Ignition Transport: Facilitates communication between components.

The simulation environment in Gazebo Ignition is highly customizable, allowing the addition of various obstacles, terrains, and other elements to create a realistic testing ground for the robot.

3.2.2 RViz2

RViz2 is a 3D visualization tool for ROS2 that allows users to visualize robot states, sensor data, and environment maps. It plays a crucial role in debugging and monitoring the robot's performance in simulations and real-world scenarios.

Key Features

- **Visualization of Sensor Data:** Displays data from various sensors such as LiDAR, cameras, and IMUs.
- **Robot Model Visualization:** Shows the robot's URDF model and its current state.
- **Interactive Markers:** Enables interactive control and manipulation of the robot.
- **Plugins:** Supports various plugins for additional functionalities.

RViz2 provides an intuitive interface for visualizing and debugging the Husarion robot's navigation and control systems.

3.2.3 Husarion Robot Model

In the development of our robotic system, we chose to work with the Husarion ROSbot XL due to the availability of both the URDF (Unified Robot Description Format) model and the physical robot.

URDF Model

The Husarion robot's URDF model is essential for simulating and visualizing the robot in Gazebo Ignition and RViz2. The URDF model includes descriptions of the robot's physical and visual properties, such as:

- **Links:** Represent the physical parts of the robot.
- **Joints:** Define the connections and degrees of freedom between links.

- **Sensors:** Specify the locations and types of sensors on the robot.
- **Actuators:** Define the motors and their control parameters.

The URDF model can be equipped with 3 different types of sensors:

- **LiDAR:** Used for distance measurement, 3D mapping and environmental awareness.
- **Camera:** Used for object recognition, scene understanding, and tasks that require visual cues.
- **IMU:** Used for stabilization, orientation and movement.

For the LiDAR and the camera we had many different options to choose from. Our setup for the simulated robot was the following:

- **LiDAR:** Slamtec RPLiDAR S1
- **Camera:** Intel RealSense Depth Camera D435

GPS Plugin

GPS sensors are crucial for autonomous navigation, enabling robots and vehicles to follow predefined coordinates with precision. They provide essential geolocation data, allowing for accurate positioning and path planning. Integrating GPS with other sensors, such as IMUs, cameras, and LiDAR, enhances navigation performance, ensuring efficiency and reliability. Since the ROSbot XL URDF model does not include a GPS sensor, we had to create a Gazebo Ignition plugin to simulate one. To do so, we had to follow the steps listed below:

- Create URDF model of the GPS
- Add the GPS to the robot's URDF file
- Create a ROS-Gazbeo bridge to publish the GPS data on a ROS topic.

To create a URDF model of the GPS, we wrote a `.xacro` file detailing all the sensor's characteristics. This file, reported in Appendix C, follows a structure similar to the other sensor components provided by Husarion in [27].

We then proceeded by adding the component to the URDF model of the robot. To do so, we added the following lines of code to the *rosbot_xl.urdf.xacro* file from the Husarion github repository [27]:

```
<xacro:include filename="$(find ros_components_description)/urdf/
    my_navsat.urdf.xacro" />
<xacro:my_navsat
  parent_link="body_link"
  xyz="-0.03 0.0 0.5"
  rpy="0.0 0.0 0.0" />
```

At this point, we had to create the ROS-Gazebo bridge to publish the GPS data on a ROS topic. We created a `.yaml` file with the following characteristics:

```
---
  - topic_name: <robot_namespace>/gps/fix
    ros_type_name: sensor_msgs/msg/NavSatFix
    gz_type_name: gz.msgs.NavSat
```

Finally, we modified the ROSbot XL *spawn.launch.py* file to automatically create the bridge every time the robot is spawned. Here is what was added to the file:

```

...

def launch_gz_bridge(context: LaunchContext, *args, **kwargs):

    ...

    ign_navsat_bridge = Node(
        package="ros_gz_bridge",
        executable="parameter_bridge",
        name="ros_gz_navsat_bridge",
        parameters=[{"config_file":
   namespaced_gz_navsat_remappings_file}],
        remappings=[
            ("/tf", "tf"),
            ("/tf_static", "tf_static"),
        ],
        output="screen",
        namespace=namespace,
    )

    ...

    return [ign_lidar_bridge, ign_camera_bridge, imu_remapping,
   ign_navsat_bridge, point_cloud_tf]

```

```
...
```

Now our plugin is working and publishing the GPS data on the */gps/fix* ROS topic.

Integration with ROS2 Navigation

The Husarion robot is integrated with the ROS2 navigation stack [28], often referred to as Nav2, which includes various packages for localization, path planning, and obstacle avoidance. Nav2, or Navigation2, is the evolution of the original ROS Navigation Stack, designed to be more flexible, scalable, and suitable for a wider range of applications. It is built on ROS2, which provides enhanced middleware, real-time capabilities, and improved security. Nav2 is modular, allowing developers to customize and extend its functionalities based on specific needs. Its key functionalities are:

- **Localization:** Nav2 helps determine the robot's position within a given map using various algorithms and sensor data. This is crucial for the robot to understand its location and navigate effectively.
- **Path Planning:** It calculates the optimal path from the robot's current location to the desired target, taking into account the robot's dynamics and environmental constraints. This ensures efficient and smooth navigation.
- **Obstacle Avoidance:** Nav2 continuously monitors the environment for obstacles and dynamically adjusts the robot's path to avoid collisions. This is essential for safe navigation in dynamic and cluttered environments.

By integrating Nav2 into their robotic systems, developers can create robust, efficient, and autonomous robots capable of navigating complex environments with ease. Successful integration and simulation in Gazebo Ignition, along with visualization in RViz2, ensure that the Husarion robot performs as expected in real-world applications.

3.2.4 Virtual Machine

Developing an autonomous guide for a robot involves running complex simulations and visualizations, which demand significant computational resources. For this reason we decided to create an AWS EC2 to run all the simulations.

AWS EC2 Specifications

AWS EC2 instances provide scalable computing capacity in the cloud. The specific AMI used, "High-End Ubuntu 22 Desktop - NICE DCV for NVIDIA-GPU 3D instances + CUDA", is optimized for high-performance computing (HPC) and includes support for NVIDIA GPUs and CUDA, which are essential for running intensive robotics simulations. This AMI was found on AWS Marketplace at the following link: aws.amazon.com/marketplace

Advantages Over Personal Computers

Personal computers, even those with high-end specifications, may struggle with the demands of running multiple simulations and visualizations concurrently. The computational power provided by AWS EC2 instances ensures that all tasks can be performed smoothly and efficiently, reducing the risk of system crashes and slowdowns.

Collaboration Benefits

Multiple team members can easily collaborate on the project by accessing the same EC2 instance. This eliminates the inconsistencies and synchronization issues that can arise when working on separate personal computers.

Pay-as-You-Go Model

AWS operates on a pay-as-you-go pricing model, meaning users only pay for the computing power they actually use. This can be more cost-effective than investing in high-end personal computers, especially for projects with fluctuating computational needs.

Maintenance and Upgrades

AWS handles the maintenance and upgrading of hardware, ensuring that users always have access to the latest technology without the additional cost and effort of maintaining personal high-end machines.

Data Protection

AWS provides robust security features, including data encryption, network firewalls, and identity management. These features ensure that sensitive project data is protected against unauthorized access and cyber threats.

Reliability

AWS guarantees high availability and reliability of its services. This minimizes downtime and ensures that simulations and developments can proceed without interruption.

DCV Viewer

To connect to our AWS EC2 that runs all the simulations we used a tool called DCV Viewer.

DCV Viewer is a client application developed by NICE, an Amazon Web Services (AWS) company, for connecting to and accessing remote desktops and graphical applications hosted on cloud or on-premises servers. It is a component of NICE DCV (Desktop Cloud Visualization), a high-performance remote display protocol designed to deliver rich graphical and interactive applications over varying network conditions.

DCV Viewer allows users to securely connect to remote desktops or applications, providing a seamless and high-quality user experience. It supports multiple platforms including Windows, Linux, and macOS, and can be used to access both 2D and 3D applications hosted on remote servers. The viewer communicates with the NICE DCV server, which streams the graphical output from the remote desktop or application to the viewer, while also transmitting user inputs (like keyboard and mouse actions) back to the server.

There are several advantages to using DCV Viewer for accessing AWS EC2 instances or other remote resources:

- High Performance: NICE DCV is optimized for high performance, enabling smooth interaction with both 2D and 3D applications. It leverages efficient compression and adaptive streaming techniques to deliver low latency and high frame rates, even over WAN connections [29].

- Security: DCV supports encryption of all communication between the client and the server, ensuring that data remains secure. It integrates with AWS security services, providing robust authentication and authorization mechanisms [30].

- Platform Independence: The viewer is available for various operating systems, making it versatile and accessible from different devices. This cross-platform compatibility ensures users can connect from their preferred environments [31].

- Scalability: As part of the AWS ecosystem, NICE DCV can easily scale with the needs of the organization. Users can take advantage of AWS's elastic infrastructure to scale up or down based on demand, optimizing costs and performance [32].

- Collaboration: DCV supports collaborative sessions where multiple users can connect to the same remote desktop or application, facilitating teamwork and collaborative workflows [33].

- Cost Efficiency: By using remote desktops and applications, organizations can reduce the need for high-end local hardware. Instead, they can leverage the computational power of EC2 instances, which can be more cost-effective and easier to manage [34].

- Ease of Use: The DCV Viewer offers a user-friendly interface, making it straightforward for users to connect to remote resources without requiring extensive configuration or technical knowledge [35].

EC2 Setup

In order to be able to simulate the ROSbot XL on our EC2 we had to configure the machine. The commands used for the setup can be found in Appendix F.

After completing the installation, we ensured everything was functioning properly by executing the example ROS2 packages in separate terminals to confirm the correct operation of both C++ and Python:

Listing 3.1: Terminal 1

```
# TERMINAL 1
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_cpp talker
```

Listing 3.2: Terminal 2

```
# TERMINAL 2
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_py listener
```

With these commands, we started a talker node (written in C++) that publishes messages on a topic, and a listener node (written in Python) that is subscribed to that topic and receives the messages sent by the talker.

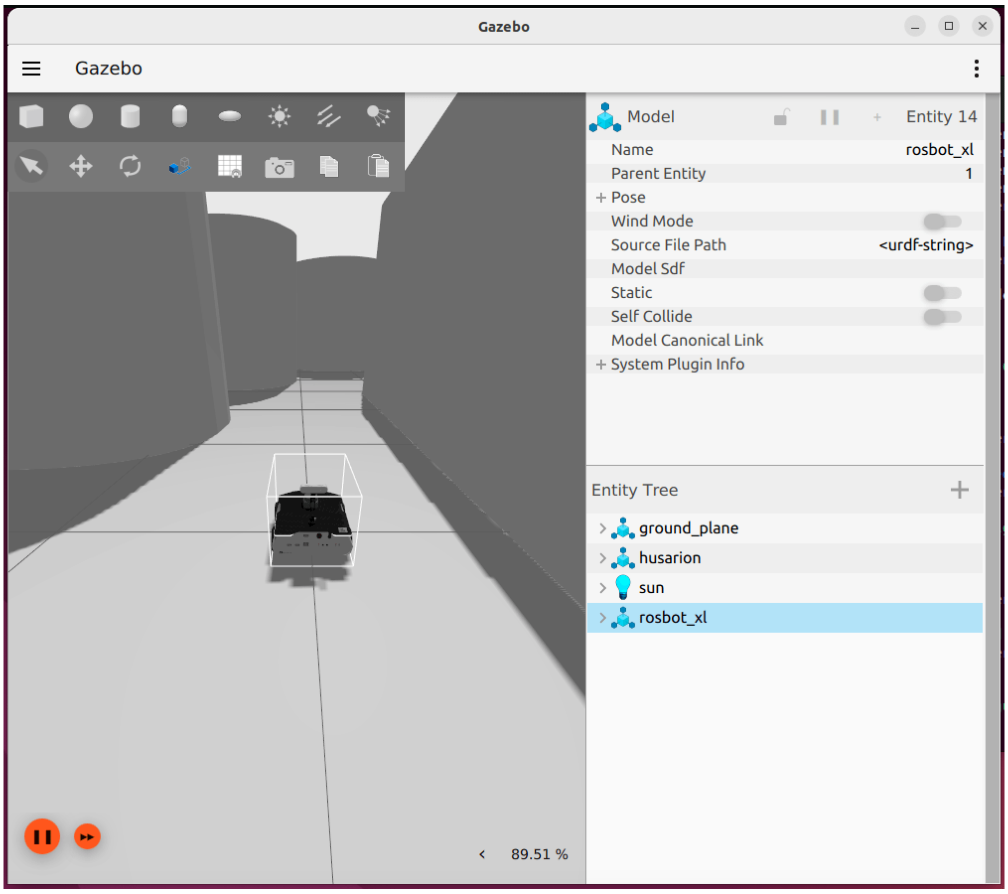
To execute the tutorials suggested by Husarion, we completed some more steps, provided in Appendix G.

Finally, we ran the first simulation with the command:

Listing 3.3: Launching Gazebo

```
ROSBOT_SIM
```

And got the following result:



Name	rosbot_xl
Parent Entity	1
+ Pose	
Wind Mode	
Source File Path	<urdf-string>
Model Sdf	
Static	
Self Collide	
Model Canonical Link	
+ System Plugin Info	

Figure 3.2: First simulation

We then proceeded to install some other useful packages:

Listing 3.4: Install packages

```
git clone -b ros2 https://github.com/husarion/tutorial_pkg.git
colcon build --symlink-install
git clone https://github.com/robo-friends/m-explore-ros2.git
colcon build --symlink-install
```

Our EC2 is now ready.

3.2.5 Map

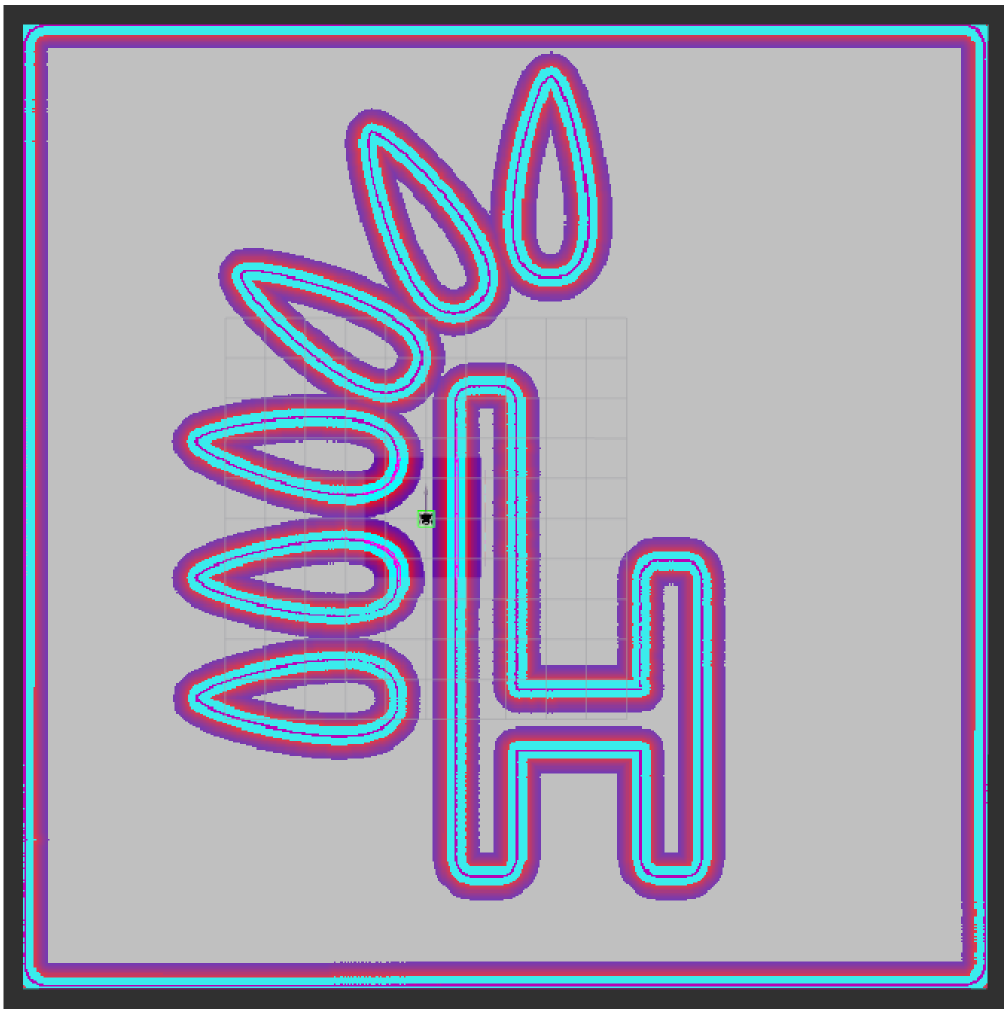


Figure 3.3: Husarion map

For the purpose of evaluating the performance and capabilities of the ROSbot-XL, we utilized the default map provided by Husarion within the Ignition Gazebo simulation environment. This map serves as a comprehensive and realistic testbed for autonomous robotics research and development.

The Husarion default map features a variety of structural elements that present common challenges encountered in real-world scenarios. The key features include:

- **Unique Layout:** The map includes distinct shapes and pathways, resembling an abstract design with interconnected loops and paths. This unique layout challenges the robot's navigation and path-planning algorithms to handle complex routes and sharp turns.

- **Varied Path Widths:** The map features paths of varying widths, requiring the robot to adapt to narrow and wide spaces, thus testing its ability to navigate through different spatial constraints effectively.
- **Simulated Obstacles:** The design includes multiple enclosed spaces and loop-like structures that act as obstacles, testing the robot's obstacle detection and avoidance systems.

Utilizing this map, we conducted a series of simulations to assess the navigation performance of the ROSbot-XL. The structured environment of the Husarion default map enabled the thorough testing of key functionalities such as SLAM (Simultaneous Localization and Mapping), autonomous navigation, and obstacle avoidance. The map's diverse settings provided a robust platform for identifying strengths and potential areas for improvement in the robot's operational capabilities.

The unique layout and varied path widths of the Husarion map presented a comprehensive challenge for the ROSbot-XL, ensuring that the simulations encompassed a wide range of scenarios that the robot might encounter in real-world applications.

GPS Integration

In order to read GPS data while navigating through the map, we had to edit the map file by adding the following code:

```
    <spherical_coordinates>
      <surface_model>EARTH_WGS84</surface_model>
      <world_frame_orientation>ENU</world_frame_orientation>
      <latitude_deg>53.1978</latitude_deg>
      <longitude_deg>18.3732</longitude_deg>
      <elevation>0</elevation>
      <heading_deg>0</heading_deg>
    </spherical_coordinates>
```

Now our GPS plugin is able to provide us with coordinates.

3.2.6 Nodes and Launch Files

In this subsection we will report all the ROS2 nodes and launch files that were created during the project. All the code is reported in Appendix D.

IMU Remapping

This node was created to publish the IMU data on a specific topic. As a matter of fact, the Navsat Transform Node of the Robot Localization package [36], which is used to transform GPS coordinates into local coordinates used by the robot, is subscribed to the `/imu` topic, while the IMU broadcaster publishes the IMU data on the `/imu_broadcaster/imu` topic. Thus, the remapping is necessary for the Navsat Transform Node to work.

Write GPS

The goal of this node is to write the data sent by the GPS in a `.txt` file which will be read to send the GPS data through the websocket.

Websocket Connection

The function of this node is to manage the websocket connection and all the messages (sent and received). Through the websocket we send the GPS coordinates of the robot and the navigation status, including distance and time remaining. On the other hand, we receive navigation goals as GPS coordinates.

Coordinates Conversion

The following node is used to convert GPS coordinates into local coordinates used by the robot to navigate. The code is taken from the Nav2 GPS Waypoint Follower Demo package by ROS Navigation [37] and modified to suit our needs. The node will be started when a series of position goals is received from the websocket, it will convert the GPS positions to local coordinates and it will publish them to the `/gps_waypoints` topic. Once all the conversions have ended, the node will publish a message on the `/control_navigator` topic to start the navigation.

Smart Navigator

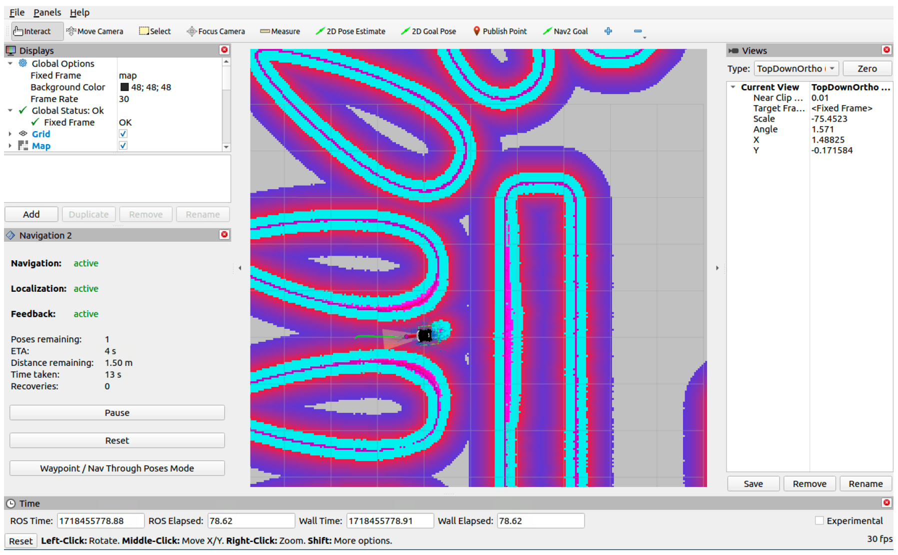
This node is responsible for the navigation of the robot. It will listen to the topics `/gps_waypoints` and `/control_navigator`. When the `"Start"` message is published on the `/control_navigator` topic, the node will start the navigation through the converted coordinates that it received on the `/gps_waypoints` topic. While navigating, the node also writes the navigation feedback on a file, which is read by the Websocket Connection Node and sent to the website to keep the user interface up to date.

Autonomous Navigation

This launch file is used to start all the nodes and launch a series of launch files that are necessary for the autonomous navigation to work.

3.2.7 Results

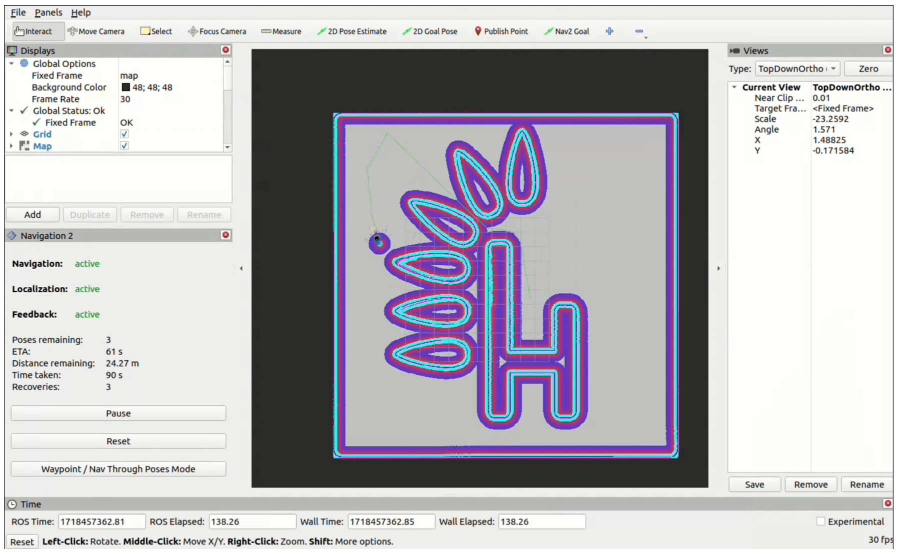
After the development of the code, a series of simulations were conducted to verify its functionality and accuracy. The results of the simulations were highly satisfactory, as the robot was able to successfully navigate to the goal whenever a valid goal was set.



```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
Map                    ✓

Navigation:    active
Localization:  active
Feedback:      active

Poses remaining:      1
ETA:                  4 s
Distance remaining:   1.50 m
Time taken:           13 s
Recoveries:           0

Type: TopDownOrtho
Current View     TopDownOrtho ...
  Near Clip ...  0.01
  Target Fra...  <Fixed Frame>
  Scale          -75.4523
  Angle          1.571
  X              1.48825
  Y              -0.171584

ROS Time: 1718455778.88   ROS Elapsed: 78.62   Wall Time: 1718455778.91   Wall Elapsed: 78.62
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Zoom. Shift: More options.   30 fps
```

Figure 3.4: Valid goal

Even when multiple waypoints were provided, the Husarion ROSbot XL managed to easily complete the whole navigation.

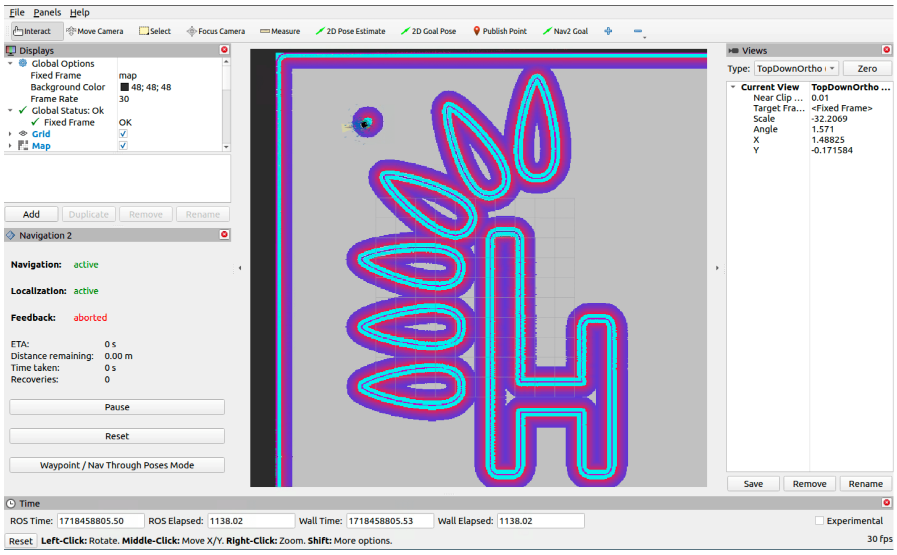


```
Displays
Global Options
  Fixed Frame         map
  Background Color    48; 48; 48
  Frame Rate          30
Global Status: Ok
  Fixed Frame         OK
Grid                  ✓
Map                   ✓

Navigation 2
Navigation:     active
Localization:   active
Feedback:       active

Poses remaining:      3
ETA:                  61 s
Distance remaining:   24.27 m
Time taken:           90 s
Recoveries:           3

Views
Type: TopDownOrtho
Current View   TopDownOrtho ...
  Near Clip ...  0.01
  Target Fra...  <Fixed Frame>
  Scale          -23.2592
  Angle          1.571
  X              1.48825
  Y              -0.171584

Time
ROS Time: 1718457362.81  ROS Elapsed: 138.26  Wall Time: 1718457362.85  Wall Elapsed: 138.26

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Zoom. Shift: More options.   30 fps
```

Figure 3.5: Multiple waypoints

A goal can be considered valid if it is inside the map and does not overlap with an obstacle.

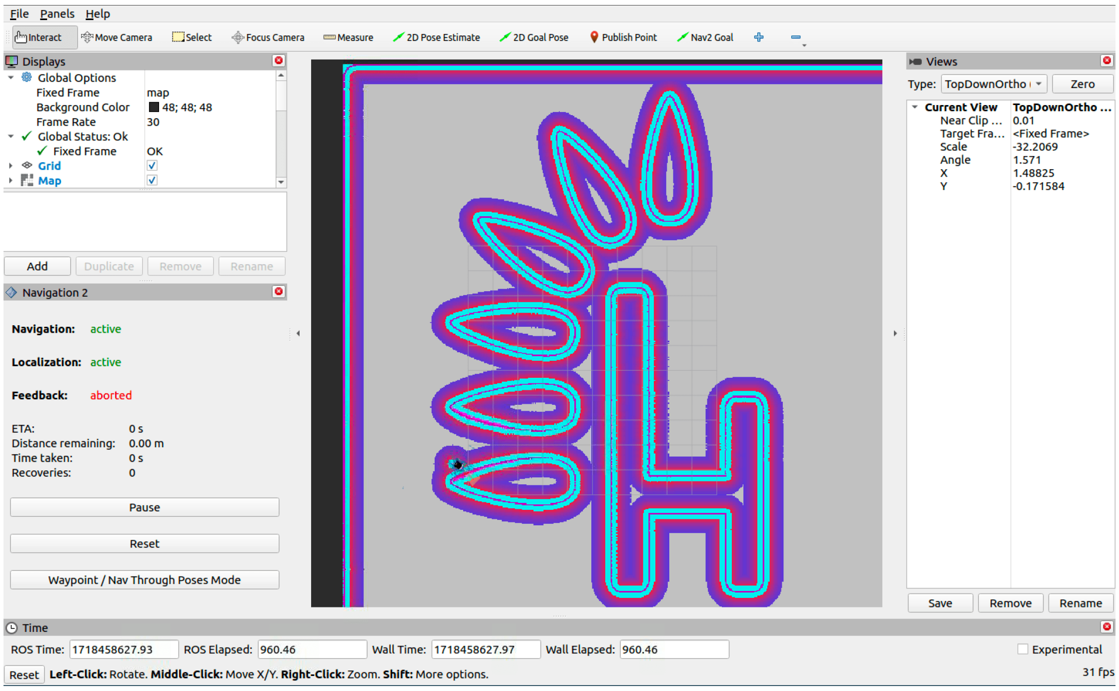
If a goal outside the map is provided, the robot remains stationary while attempting to generate a path to the unreachable point. However, it consistently fails, and after several attempts, it aborts the navigation process.



```
Displays
Global Options
  Fixed Frame         map
  Background Color    48; 48; 48
  Frame Rate          30
Global Status: Ok
  Fixed Frame         OK
Grid                  ✓
Map                   ✓

Navigation 2
Navigation:     active
Localization:   active
Feedback:       aborted

ETA:                  0 s
Distance remaining:   0.00 m
Time taken:           0 s
Recoveries:           0

Views
Type: TopDownOrtho
Current View   TopDownOrtho ...
  Near Clip ...  0.01
  Target Fra...  <Fixed Frame>
  Scale          -32.2069
  Angle          1.571
  X              1.48825
  Y              -0.171584

Time
ROS Time: 1718458805.50  ROS Elapsed: 1138.02  Wall Time: 1718458805.53  Wall Elapsed: 1138.02

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Zoom. Shift: More options.   30 fps
```

Figure 3.6: Invalid goal - outside of map

Conversely, when a navigation goal intersects with an obstacle, the robot navigates towards it and attempts various strategies to overcome the obstruction. After several unsuccessful attempts, it ultimately gives up and aborts the navigation process.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
Map	✓

Navigation: active

Localization: active

Feedback: aborted

ETA:	0 s
Distance remaining:	0.00 m
Time taken:	0 s
Recoveries:	0

Type: TopDownOrtho

Current View	TopDownOrtho ...
Near Clip ...	0.01
Target Fra...	<Fixed Frame>
Scale	-32.2069
Angle	1.571
X	1.48825
Y	-0.171584

ROS Time: 1718458627.93 ROS Elapsed: 960.46 Wall Time: 1718458627.97 Wall Elapsed: 960.46 Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options. 31 fps

Figure 3.7: Invalid goal - obstacle overlap

3.3 Robot

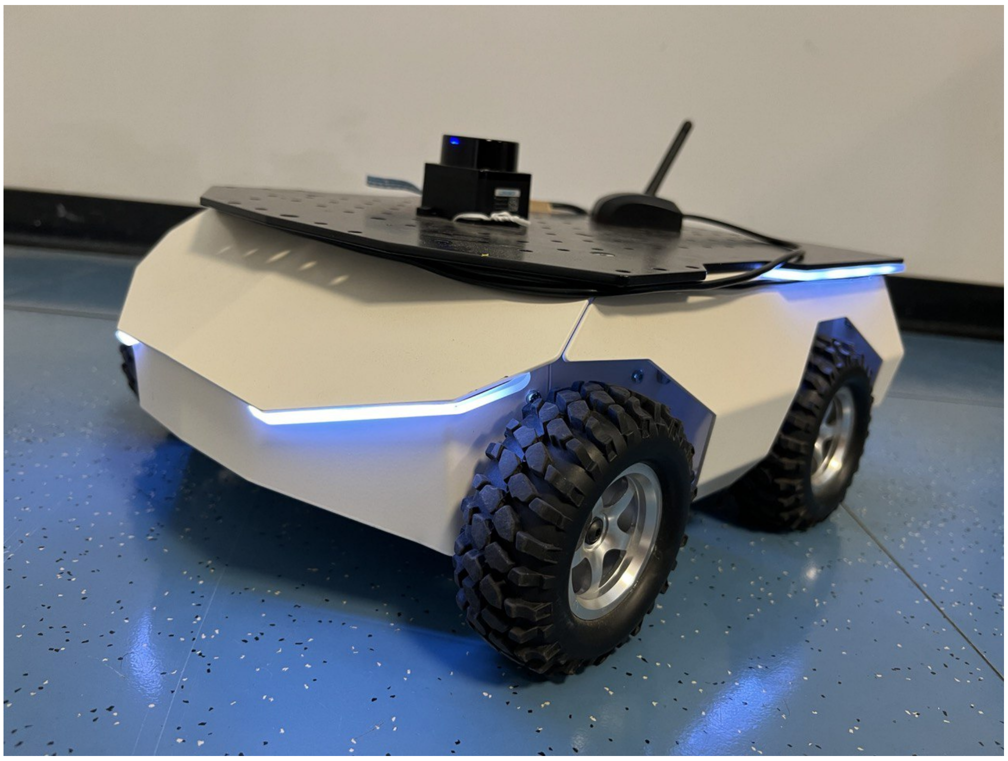


Figure 3.8: Husarion ROSbot XL

After working with the simulated model, we moved on to the real robot, which was provided by **Q8 Italia**. As mentioned before, we worked on a Husarion ROSbot XL. The **Husarion ROSbot XL** is a robust and versatile mobile robot platform designed for advanced research, education, and prototyping applications in the field of robotics. It is equipped with a variety of sensors and computing power, making it suitable for complex autonomous navigation, SLAM (Simultaneous Localization and Mapping), and other advanced robotics tasks.

Key Features

- **Modular Design**: The ROSbot XL's modular architecture allows for easy customization and expansion, enabling users to add various sensors, actuators, and other peripherals to tailor the robot to specific research needs.
- **Powerful Computing**: It comes with a Raspberry Pi 4 capable of running ROS (Robot Operating System), which is essential for developing and testing advanced algorithms in robotics.

- **Sensor Suite**: The robot is equipped with an array of sensors, including LiDAR, cameras, IMU (Inertial Measurement Unit), and GPS, providing rich data for navigation and environmental interaction.
- **ROS Compatibility**: As the name suggests, the ROSbot XL is fully compatible with ROS, allowing users to leverage a vast ecosystem of libraries and tools for robotics development.
- **Autonomous Capabilities**: With its powerful hardware and sensor suite, the ROSbot XL can perform autonomous navigation, obstacle avoidance, and mapping, making it ideal for applications in research and education.

Overall, the Husarion ROSbot XL is a comprehensive and flexible platform that supports a wide range of robotics applications, from academic research to industrial prototyping.

3.3.1 Setup

The first step was to download the HusarionOS image from `https://husarion.com/software/os/installation/#robot-setup-guide` and install it on the Raspberry Pi 4 using RaspberryPi Imager. We then proceeded by following the setup guide at `https://husarion.com/software/os/installation/#robot-setup-guide` and finally we completed the `how to start` tutorial at `https://husarion.com/tutorials/howtostart/`

Remote Access

In order to configure remote access to the robot we decided to use a tool called Tailscale to create a free VPN. This way we will always be able to connect via ssh even when the robot is connected to a different Wi-Fi connection with respect to our computer in a more secure way. Since we wanted to be able to see the robot's visual outputs on our computer using the ssh connection we had to configure X11 Forwarding by editing the `/etc/ssh/sshd_config` file both on the robot and on our machine. The changes that were made are presented below.

Listing 3.5: X11 Forwarding configuration - robot

```
X11Forwarding yes
X11DisplayOffset 10
#X11UseLocalhost yes
```

Listing 3.6: X11 Forwarding configuration - machine

```
X11Forwarding yes
#X11DisplayOffset 10
#X11UseLocalhost yes
```

Then, on the robot we launched:

```
echo "export DISPLAY=:10" >> ~/.bashrc
source ~/.bashrc
```

And on our computer:

```
xhost +
ssh -X husarion@husarion
```

Finally, we were able to easily connect to our robot and to forward the visual outputs of the robot onto our machine.

3.3.2 Nodes and Launch Files

The nodes and launch files on the real robot are largely consistent with those presented in the Simulation section. However, there are some minor modifications specific to the robot context, which are detailed below.

GPS

The equipped GPS is a G Mouse VK-162, a very practical GPS receiver that connects to our robot through a USB port. We created a ROS node, called GPS Broadcaster Node, to read the GPS data and publish it on the `/gps/fix` topic. Also, as the Write GPS Node from the simulation, it writes the GPS coordinates on a file. This way the Websocket Connection Node is able to read the file and send the information to the webpage. The implementation of the node can be found in Appendix E.

LiDAR

A FHL-LD19P LiDAR was used for this project. It was connected to our robot via USB port and configured following the setup guide at https://wiki.youyeetoo.com/en/Lidar/D300#h-53-operation-based-on-ros2-under-linux. The guide assisted us in constructing a package containing a launch file that initiates the reading of LiDAR data and broadcasts it to the appropriate topic.

Autonomous Navigation Launch File

This LiDAR launch file and the GPS Broadcaster Node were added to the Autonomous Navigation Launch File D.6, in order to automatically start them when launching the navigation package.

3.3.3 Results

In this section, we present the results of testing the autonomous navigation system of our robot in two different environments: beSharp's roof and a field near beSharp.

Mapping the Environments

The first phase of the test involved mapping the areas using the Simultaneous Localization and Mapping (SLAM) package. The mapping process on beSharp's roof was straightforward and resulted in an accurate map of the area.

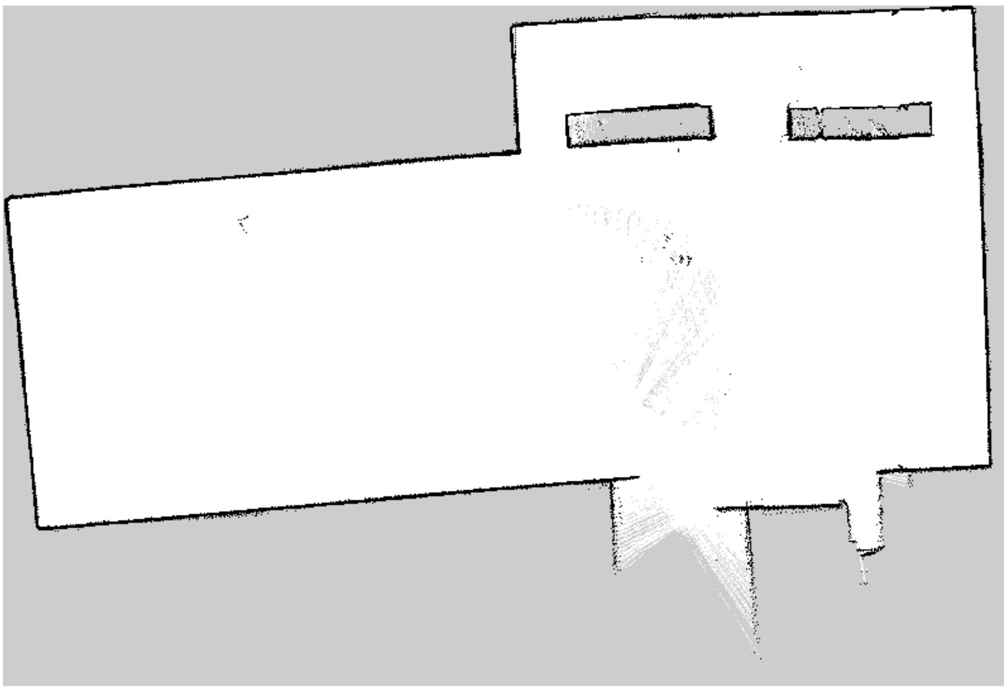


Figure 3.9: Roof map

However, mapping the field presented significant challenges. The lidar sensor frequently detected grass as obstacles, resulting in a map filled with numerous false obstacles. These unreal obstacles made navigation impossible, as the robot perceived the area as cluttered with barriers, thus failing to generate a viable path.

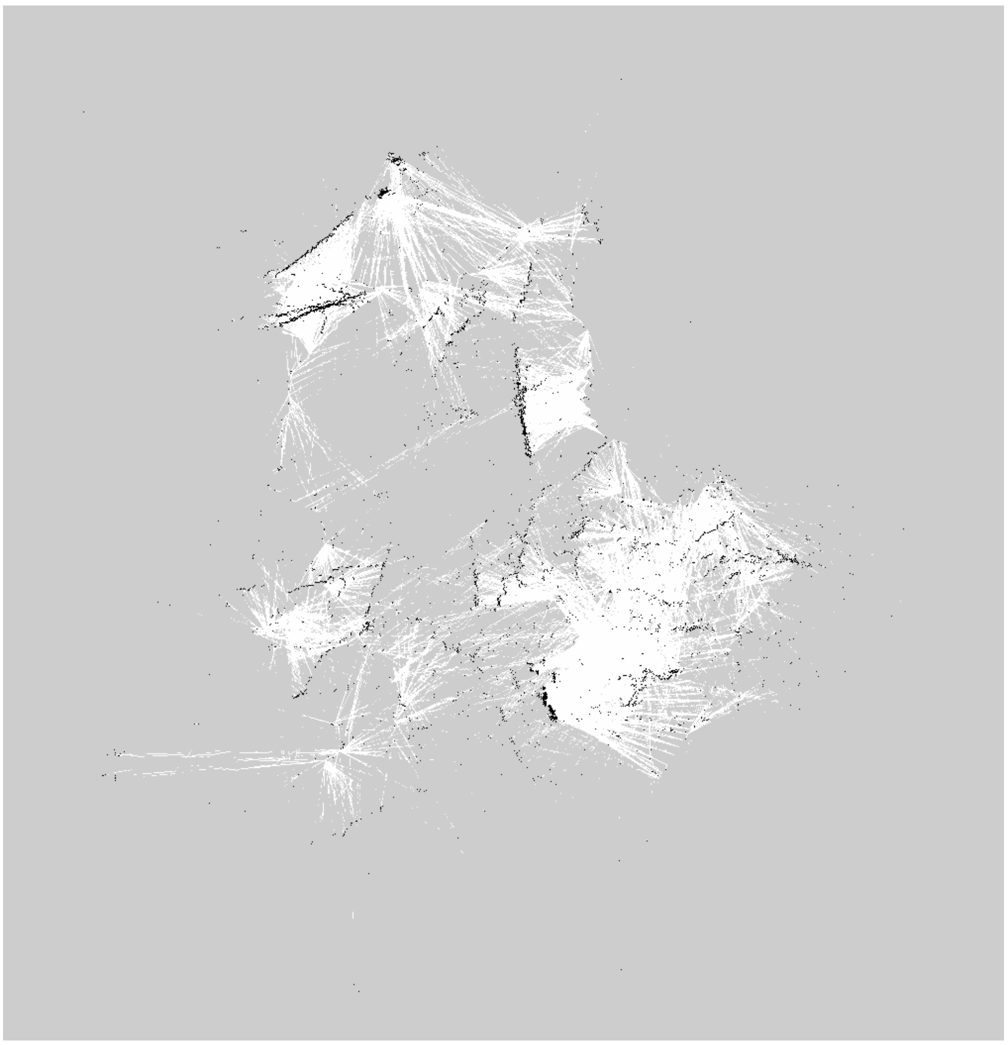


Figure 3.10: Field map

To address this issue, we decided to use an empty map for the field test.

Navigation Performance

After the mapping phase, we proceeded to the navigation tests.
Following a series of unsuccessful attempts, we finally achieved successful GPS-driven autonomous navigation with our robot. Despite the initial challenges, we managed to complete the navigation tasks, demonstrating the effectiveness and reliability of our system.

Discussion

These tests highlight the critical importance of high-accuracy sensors for autonomous navigation tasks. The discrepancies in the field test underscore the necessity for a more reliable lidar system that can distinguish between real obstacles and irrelevant objects like grass. Additionally, for vineyard navigation, a more precise positioning system is essential for setting accurate waypoints. A differential GPS can be employed for this purpose, offering centimeter-level accuracy. In conclusion, the real-world tests of our autonomous navigation system were successful, despite the challenges faced.

Chapter 4

Conclusion

In this last chapter we present the obtained results and comment on what the next steps might be.

4.1 Experimental Results

In conclusion, the experiments conducted as part of this thesis demonstrated promising results in the development of a GPS-driven autonomous navigation system for precision agriculture. The system successfully leveraged the robust capabilities of Amazon Web Services (AWS) to provide a scalable and efficient solution for real-time navigation. However, several challenges remain that need to be addressed in future research to enhance the system's performance and applicability.

One of the primary issues encountered was the initial localization of the robot. Accurate initial positioning is crucial for the robot's navigation system to function optimally, and improvements in this area are necessary. Additionally, while the current implementation utilized an indoor robot, a more suitable outdoor robot would better reflect real-world agricultural applications. The limitations of using a 2D LiDAR instead of a 3D LiDAR also impacted the precision and reliability of the navigation system, highlighting another area for potential enhancement. Furthermore, the use of a standard GPS introduced some inaccuracies that could be mitigated by employing a differential GPS system for higher precision.

4.2 Future Work

Looking forward, several next steps are proposed to address these challenges and advance the current system:

- **Enhancing Initial Localization:** Implementing advanced algorithms and techniques for more accurate initial localization of the robot to improve navigation accuracy.
- **Transition to Outdoor Robots:** Utilizing an outdoor robot that is better suited for agricultural environments to test and validate the system in real-world conditions.
- **Upgrading to 3D LiDAR:** Integrating a 3D LiDAR sensor to enhance the perception capabilities of the robot, enabling more precise mapping and obstacle avoidance.
- **Adopting Differential GPS:** Using a differential GPS system to significantly improve the accuracy of the robot's positioning.
- **AWS GreenGrass for Code Deployment:** Deploying code to the robots using AWS GreenGrass, which allows for efficient and scalable edge computing, ensuring seamless updates and management of the system.

This project was conducted as a proof of concept (POC), and the insights gained from this phase provide a strong foundation for future research and development. By addressing the identified challenges and following the proposed next steps, the autonomous navigation system can be further refined to meet the demanding requirements of precision agriculture. This will ultimately contribute to the advancement of agricultural robotics, enhancing the efficiency and sustainability of farming practices.

Appendix A

Websocket functions

Listing A.1: Connect function

```python
import json
import boto3
import os

dynamodb = boto3.client('dynamodb')

def lambda_handler(event, context):
    print(event)
    connectionId = event['requestContext']['connectionId']

    dynamodb.put_item(
        TableName=os.environ['WEBSOCKET_TABLE'],
        Item={'connection_id': {'S': connectionId}}
    )

    return {}
```

Listing A.2: Disconnect function

```python
import json
import boto3
import os

dynamodb = boto3.client('dynamodb')

def lambda_handler(event, context):
    print(event)
    connectionId = event['requestContext']['connectionId']

    dynamodb.delete_item(
        TableName=os.environ['WEBSOCKET_TABLE'],
        Key={'connection_id': {'S': connectionId}}
    )

    return {}
```

Listing A.3: Message function

```
import json
import boto3
import os

dynamodb = boto3.client('dynamodb')


def lambda_handler(event, context):
    print(event)
    message = json.loads(event['body'])['text']
    print(message)
    sender_id = event['requestContext']['connectionId']
    paginator = dynamodb.get_paginator('scan')
    connectionIds = []

    apigatewaymanagementapi = boto3.client(
        'apigatewaymanagementapi',
        endpoint_url = "https://" + event["requestContext"]["
    domainName"] + "/" + event["requestContext"]["stage"]
    )

    for page in paginator.paginate(TableName=os.environ['
    WEBSOCKET_TABLE']):
        connectionIds.extend(page['Items'])

    for connectionId in connectionIds:
        if connectionId['connection_id']['S'] != sender_id:
            apigatewaymanagementapi.post_to_connection(
                Data=json.dumps(message),
                ConnectionId=connectionId['connection_id']['S']
            )
            print('sent')

    return {}
```

Appendix B

Website

Listing B.1: HTML and JavaScript

```html
<!DOCTYPE html>
<html>
<head>
  <title>OpenLayers Click Example</title>
  <link rel="stylesheet" href="https://openlayers.org/en/v6.15.1/css/
    ol.css" type="text/css">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/
    bootstrap.min.css" rel="stylesheet" integrity="sha384-
    QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH"
    crossorigin="anonymous">
  <link rel="stylesheet" type="text/css" href="style/style.css">
  <script src="https://openlayers.org/en/v6.15.1/build/ol.js"></
    script>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/
    bootstrap.bundle.min.js" integrity="sha384-
    YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz"
    crossorigin="anonymous"></script>
</head>
<body>
  <div class="container" id="title-container">
    <h1 id="title">Otto Robot</h1>
  </div>
  <div class="container-fluid" id="map-container">
    <div class="row">
      <div class="col-lg-4">
        <!-- Sidebar -->
        <div class="sidebar">
          <h2>Info</h2>
          <p>Status: <span id="status" class="light-grey-status">
    offline</span></p>
          <p>Time remaining: <span id="timeRemaining"></span></p>
        <p>Distance remaining: <span id="distanceRemaining"></span></p>

        <h3>Waypoints</h3>
        <p id="waypoint-tip">Click on the map to add a waypoint</p>
        <ul id="dynamic-list" class="list-group">
        </ul>
      </div>
      <button type="button" id="sendBtn" class="btn btn-outline-primary disabled" onclick="sendBtnClick();">Start</button>
      <button type="button" id="removeBtn" class="btn btn-outline-danger right-alignment hidden" onclick="removeBtnClick();">Clear all</button>
    </div>
    <div class="col-lg-8">
      <!-- OpenLayers Map Container -->
      <div id="map" class="map">
        <div class="form-check form-switch right-alignment" id="switchDiv">
          <input class="form-check-input" type="checkbox" role="switch" id="flexSwitchCheckChecked" checked>
          <label class="form-check-label" for="flexSwitchCheckChecked">Lock view</label>
        </div>
      </div>
    </div>
  </div>
</div>

<script>

function connectWebSocket() {
    const websocketURL = 'wss://7xuofkfz0a.execute-api.eu-west-1.amazonaws.com/production/';
    socket = new WebSocket(websocketURL);
    socket.onopen = function(event) {
        console.log('WebSocket connection opened:', event);
        robotStatus.innerText = 'offline';
        robotStatus.classList.remove('light-green-status');
        robotStatus.classList.add('light-grey-status');
    };
    socket.onmessage = function(event) {
        var data = JSON.parse(event.data);
        if (data.sender == 'robot') {
          lat = data.gps_data[0];
          lon = data.gps_data[1];
          if (lockView){
            updateMap(lat, lon);
          }
          updateGpsMarker(lat, lon, '#3B78EA');
          resetHeartbeat(lat, lon);
          if (data.time_remaining != -1.0){
            updateTimeRemaining(data.time_remaining);
          }
          if (data.distance_remaining != -1.0){
            updateDistanceRemaining(data.distance_remaining);
          }
          robotStatus.innerText = 'online';
          robotStatus.classList.add('light-green-status');
          robotStatus.classList.remove('light-grey-status');
        }
    };
    socket.onclose = function(event) {
        console.log('WebSocket connection closed:', event);
        robotStatus.innerText = 'disconnected (refresh the page)';
        robotStatus.classList.remove('light-green-status');
        robotStatus.classList.add('light-grey-status');
    };
    socket.onerror = function(event) {
        console.error('WebSocket error:', event);
    };
}

function updateTimeRemaining(time){
  const timeSpan = document.getElementById('timeRemaining');
  var hours = Math.floor(time / 3600);
  var minutes = Math.floor((time % 3600) / 60);
  var seconds = time % 60;
  timeText = Math.ceil(seconds) + " s";
  if (minutes > 0){
    timeText = minutes + " m " + timeText;
  }
  if (hours > 0){
    timeText = hours + " h " + minutes + " m " + timeText;
  }
  timeSpan.innerHTML = timeText;
}

function updateDistanceRemaining(distance){
  const distanceSpan = document.getElementById('distanceRemaining');
  distanceSpan.innerHTML = distance.toFixed(2) + ' m';
}


function updateMap(lat, lon) {
    map.getView().animate({
      center: ol.proj.fromLonLat([lon, lat]),
      duration: 300
    });
}

function updateGpsMarker(lat, lon, circleColor) {
  var markerFeature = gpsSource.getFeatures()[0];
  if (markerFeature.getStyle().getImage().getRadius() == 0){
    map.getView().animate({
      center: ol.proj.fromLonLat([lon, lat]),
      duration: 300,
      zoom: 18
    });
    markerFeature.getGeometry().setCoordinates(ol.proj.fromLonLat([
lon, lat]));
  }
  var newStyle = new ol.style.Style({
    image: new ol.style.Circle({
      radius: 6,
      fill: new ol.style.Fill({
          color: circleColor
      }),
      stroke: new ol.style.Stroke({
          color: '#F1F4FA',
          width: 2
      })
    })
  });
  markerFeature.setStyle(newStyle);
  var line = new ol.geom.LineString([markerFeature.getGeometry().
getCoordinates(), ol.proj.fromLonLat([lon, lat])]);
  var step = 0;
  var key = setInterval( function() {
    if (step < 100) {
      step++;
      markerFeature.getGeometry().setCoordinates(line.
getCoordinateAt(step/100));
      //marker.setGeometry(new Point(line.getCoordinateAt(step/100)
));
    } else {
      clearInterval(key);
    }
  }, 10);
  map.updateSize();
  if (waypointsList.length > 0){
    distanceFromWaypoint = calculateDistance({lat: lat, lon: lon},
waypointsList[0].data);
    if (distanceFromWaypoint < precision){
      var myList = document.getElementById('dynamic-list');
      myList.removeChild(myList.firstChild);
      removeMarker(waypointsList[0].markerId);
    }
  }
}


// You can send messages to the WebSocket server using socket.send
()
function sendMessage(waypoints) {
  const message = {
      type: 'default',
      text: {sender: 'frontend', data: waypoints}
  };
  console.log(message)
  socket.send(JSON.stringify(message));
}


function addMarker(event) {
  var coordinates = event.coordinate;
  var marker = new ol.Feature({
      geometry: new ol.geom.Point(coordinates),
      text: (markerSource.getFeatures().length + 1).toString()
  });
  marker.setId(markerIdCounter);
  markerSource.addFeature(marker);
  sendBtn.classList.remove('disabled');
  removeBtn.classList.remove('hidden');

  var coords = ol.proj.transform(coordinates, 'EPSG:3857', 'EPSG
:4326');
  var longitude = coords[0];
  var latitude = coords[1];
  waypointsList.push({markerId: markerIdCounter, data: {lat:
latitude, lon: longitude}})
}


var styleFunction = function(feature) {
  var text = feature.get('text'); // Get the text for the marker
  var style = new ol.style.Style({
      image: new ol.style.Circle({
          radius: 6, // Radius of the circle
          fill: new ol.style.Fill({
              color: '#3B78EA' // Fill color of the circle
          })
      }),
      text: new ol.style.Text({
          text: text, // Set the text for the marker
          fill: new ol.style.Fill({
              color: 'white' // Text color
          })
      })
  });
  return style;
};


function updateList(lat, lon) {
  var listItem = document.createElement('li');
  listItem.className = 'points-list';
  listItem.textContent = 'lat: ' + lat.toFixed(6) + ' lon: ' + lon.
toFixed(6);

  var deleteButton = document.createElement('button');
  deleteButton.className = 'btn-close right-alignment float-right';
  deleteButton.onclick = function() {
    var markerId = parseInt(this.parentNode.dataset.markerId);
    removeMarker(markerId);
    this.parentNode.remove();
  };
  listItem.dataset.markerId = markerIdCounter;
  listItem.appendChild(deleteButton);
  list.appendChild(listItem);
}

function removeMarker(markerId) {
  var markerToRemove = markerSource.getFeatureById(markerId);
  markerSource.removeFeature(markerToRemove);
  markerSource.getFeatures().forEach(function(markerFeature,
markerIndex) {
    markerFeature.set('text', (markerIndex + 1).toString());
  });

  removeWaypointFromArray(markerId);

  if (markerSource.getFeatures().length == 0){
    sendBtn.classList.add('disabled');
    removeBtn.classList.add('hidden');
    removeBtn.innerHTML = "Clear all";
    sendBtn.innerHTML = "Start";
  }
}

function removeWaypointFromArray(markerId) {
  for (var i = 0; i < markerIdCounter; i++) {
    try{
      if (waypointsList[i].markerId === markerId) {
        waypointsList.splice(i, 1);
        break;
      }
    } catch{
    }
  }
}

function extractCoordinates() {
  var coordinatesArray = [];
  for (var i = 0; i < waypointsList.length; i++) {
    coordinatesArray.push(waypointsList[i].data);
  }
  return coordinatesArray;
}

function removeBtnClick() {
  for (var i = 0; i < markerIdCounter; i++) {
    removeMarker(i);
  }
  list.innerHTML = "";
  markerIdCounter = 0;
  navigating = false;
}

function sendBtnClick() {
  if (robotStatus.innerText !== 'online'){
    alert('Wait for the robot to connect and try again.');
    return
  }
  waypoints = extractCoordinates();
  sendMessage(waypoints);
  removeBtn.innerHTML = "Cancel";
  sendBtn.innerHTML = "Started!";
  sendBtn.classList.add('disabled');
  navigating = true;
}


function resetHeartbeat(lat, lon) {
    clearTimeout(heartbeatTimer);
    heartbeatTimer = setTimeout(function() {
      var robotStatus = document.getElementById('status');
      robotStatus.innerText = 'offline';
      robotStatus.classList.remove('light-green-status');
      robotStatus.classList.add('light-grey-status');
      navigating = false;
      updateGpsMarker(lat, lon, 'grey');
    }, heartbeatThreshold);
```

```
}

function calculateDistance(point1, point2) {
  var dx = point1.lat - point2.lat;
  var dy = point1.lon - point2.lon;
  return Math.sqrt(dx * dx + dy * dy);
}


let socket;
var heartbeatTimer;
var heartbeatThreshold = 5000; // 5 seconds
var markerIdCounter = 0;
var lockView = true;
var lockViewSwitch = document.getElementById('
  flexSwitchCheckChecked');
var sendBtn = document.getElementById('sendBtn');
var removeBtn = document.getElementById('removeBtn');
var list = document.getElementById('dynamic-list');
var waypointsList = [];
var distanceFromWaypoint = 0;
var navigating = false;
var precision = 2e-6;
var robotStatus = document.getElementById('status');


connectWebSocket();
var map = new ol.Map({
  target: 'map',
  layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM()
    })
  ],
  view: new ol.View({
    center: ol.proj.fromLonLat([0.0, 0.0]),
    zoom: 3
  })
});
  // Create marker
var gpsMarker = new ol.Feature({
  geometry: new ol.geom.Point(ol.proj.fromLonLat([0.0, 0.0]))
});

// Style for the marker
var markerStyle = new ol.style.Style({
    image: new ol.style.Circle({
        radius: 0,
        fill: new ol.style.Fill({
            color: '#3B78EA'
        }),
        stroke: new ol.style.Stroke({
            color: '#F1F4FA',
            width: 2
        })
    })
});

gpsMarker.setStyle(markerStyle);

var gpsSource = new ol.source.Vector({
    features: [gpsMarker]
});

var gpsLayer = new ol.layer.Vector({
    source: gpsSource
});

map.addLayer(gpsLayer);

var markerSource = new ol.source.Vector();
var markerLayer = new ol.layer.Vector({
    source: markerSource
});
markerLayer.setStyle(styleFunction);
map.addLayer(markerLayer);

// Add a click event handler to the map
map.on('click', function(evt) {
  var coords = ol.proj.transform(evt.coordinate, 'EPSG:3857', 'EPSG
  :4326');
  var longitude = coords[0];
  var latitude = coords[1];
  //updateMap(latitude, longitude);
  //sendMessage(latitude, longitude);
  addMarker(evt);
  updateList(latitude, longitude);
  markerIdCounter++;
});

map.on('pointerdrag', function(evt){
  lockViewSwitch.checked = false;
  lockView = lockViewSwitch.checked;
});

lockViewSwitch.addEventListener('change', function(evt){
  lockView = lockViewSwitch.checked;
});
```

```

  </script>
</body>
</html>
```

Listing B.2: CSS

```
#title-container{
    background-color: black;
    min-width: 100%;
    margin: 0;
    padding: 1.5% 5vw;
  }
  #title{
    color: white;
  }
  #map{
    width: 100%;
    height: 500px;
  }
  #map-container{
    margin: 1%;
    max-width: 98%;
  }
  #dynamic-list{
    max-height: 300px;
    overflow-y: auto;
    margin-bottom: 2%;
    min-width: 100%;
  }
  .points-list{
    font-size: small !important;
    min-width: 100%;
  }
  #waypoint-tip{
    font-size: small;
    color: grey;
    margin: 0 0 2% 0;
  }
  .light-green-status{
    color: #20B736;
  }
  .light-grey-status{
    color: #9E9E9E;
  }
  .right-alignment{
    float: right;
  }
  #switchDiv{
    margin: 2%;
  }
  .hidden{
    display: none;
  }
```

Appendix C

GPS URDF Model

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:macro name="my_navsat"
               params="parent_link xyz rpy
                       topic:=gps/fix">

    <joint name="${parent_link.rstrip('_link')}_to_navsat_joint" type
    ="fixed">
      <origin xyz="${xyz}" rpy="${rpy}" />
      <parent link="${parent_link}" />
      <child link="navsat_link" />
    </joint>

    <link name="navsat_link">
      <visual>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
        <geometry>
          <mesh filename="package://ros_components_description/meshes
    /donut1.dae" />
        </geometry>
      </visual>

      <!-- base and head collision -->
      <collision>
        <origin xyz="0.0 0.0 ${0.0613/2.0}" rpy="0.0 0.0 0.0" />
        <geometry>
          <box size="0.0556 0.0556 0.0413" />
        </geometry>
      </collision>

      <inertial>
```

```
        <origin xyz="0.0 0.0 ${0.0613/2.0 + 0.0018237}" rpy="0.0 0.0
    0.0" />
        <mass value="0.115033" />
        <inertia ixx="0.00004115765" ixy="0.0"             ixz="0.0"
                                     iyy="0.00004115765"   iyz="0.0"
                                                           izz="
    0.00004956023" />
      </inertial>
    </link>


      <gazebo reference="${parent_link}">
        <sensor type="navsat" name="navsat_sensor">

          <topic>${topic}</topic>
          <frame_id>${parent_link}</frame_id>
          <ignition_frame_id>${parent_link}</ignition_frame_id>

          <update_rate>10.0</update_rate>

          <always_on>1</always_on>
          <visualize>false</visualize>
          <ros>
            <namespace></namespace>
          </ros>
        </sensor>
      </gazebo>
  </xacro:macro>
</robot>
```

Appendix D

Nodes and Launch Files - Simulation

Listing D.1: IMU Remapping Node

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu

class ImuRemappingNode(Node):
    def __init__(self):
        super().__init__('imu_remapping')
        self.imu_sub = self.create_subscription(Imu, '/
    imu_broadcaster/imu', self.imu_callback, 10)
        self.imu_pub = self.create_publisher(Imu, '/imu', 10)

    def imu_callback(self, msg: Imu):
        self.imu_pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = ImuRemappingNode()

    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Listing D.2: Write GPS Node

```python
import rclpy
```

```
from rclpy.node import Node
from sensor_msgs.msg import NavSatFix

class WriteGpsNode(Node):
    def __init__(self):
        super().__init__('write_gps')
        self.gps_sub = self.create_subscription(NavSatFix, '/gps/fix'
    , self.write_gps_callback, 10)
        self.i = 1

    def write_gps_callback(self, msg: NavSatFix):
        if self.i >= 10:
            self.i = 1
            coords = f'{msg.latitude}-{msg.longitude}'
            write_gps(coords)
        self.i += 1

def write_gps(coords):
    with open('/home/ubuntu/gps_position/gps_position.txt', 'w') as f
    :
        f.write(coords)

def main(args=None):
    rclpy.init(args=args)
    node = WriteGpsNode()

    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Listing D.3: Websocket Connection Node

```
import websocket
import json
import rclpy
from rclpy.node import Node
import os
import re
from sensor_msgs.msg import NavSatFix
import threading
import time

class WebSocketNode(Node):
    def __init__(self):
        super().__init__('WebSocket')
        # WebSocket
```

```python
        websocket.enableTrace(True)
        self.i = 1
        self.ws = websocket.WebSocketApp(
            "wss://7xuofkfz0a.execute-api.eu-west-1.amazonaws.com/production/",
            on_open=self.on_open,
            on_message=self.on_message,
            on_error=self.on_error,
            on_close=self.on_close
        )
        self.ws.run_forever()


    def on_open(self, ws):
        self.get_logger().info('WebSocket: connection open!')
        self.send_data()

    def on_message(self, ws, message):
        msg = json.loads(message)
        if msg['sender'] != 'robot':
            data = msg['data']
            self.get_logger().info(f"Received: {data}")
            command = "ros2 run nav2_gps_waypoint_follower_demo logged_waypoint_follower '["
            for coord in data:
                command += "{\"lat\": " + str(coord['lat']) + ", \"lon\": " + str(coord['lon']) + "}, "
            command = command[:-2] + "]'"
            self.get_logger().info(command)
            os.system(command)

    def on_error(self, ws, error):
        print(f"Error: {error}")

    def on_close(self, ws, close_status_code, close_msg):
        print("WebSocket closed")

    def send_data(self):
        with open('/home/ubuntu/gps_position/gps_position.txt', 'r') as gps_file:
            gps_line = gps_file.readline()
        lat, lon = gps_line.split('-')

        with open('/home/ubuntu/navigator_feedback/navigator_feedback.txt', 'r') as nav_file:
            nav_line = nav_file.readline()
        distance_pattern = r"distance_remaining=(\d+\.\d+)"
        distance_matches = re.search(distance_pattern, nav_line)
        if distance_matches:
            distance_remaining = float(distance_matches.group(1))
        else:
            distance_remaining = -1.0

        time_pattern = r"estimated_time_remaining=builtin_interfaces
    \.msg\.Duration\(sec=(\d+), nanosec=(\d+)\)"
        time_matches = re.search(time_pattern, nav_line)
        if time_matches:
            estimated_sec = int(time_matches.group(1))
            estimated_nsec = int(time_matches.group(2))
            time_remaining = estimated_sec + estimated_nsec / 1e9
        else:
            time_remaining = -1.0

        message = {
            'type': 'default',
            'text': {'sender': 'robot', 'gps_data': [float(lat),
    float(lon)], 'distance_remaining': distance_remaining, '
    time_remaining': time_remaining}
        }
        self.ws.send(json.dumps(message))
        threading.Timer(1, self.send_data).start()
        return


def main(args=None):
    rclpy.init(args=args)
    node = WebSocketNode()

    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Listing D.4: Coordinates Conversion Node

```
import rclpy
import sys
import json
from robot_localization.srv import FromLL
from rclpy.node import Node
from nav2_gps_waypoint_follower_demo.utils.gps_utils import
    latLonYaw2Geopose
from geometry_msgs.msg import PoseStamped
from std_msgs.msg import String


```

```python
class GpsWpCommander(Node):

    def __init__(self, wps_file_path):
        super().__init__('minimal_client_async')
        self.pose_publisher = self.create_publisher(PoseStamped, '/gps_waypoints', 10)
        self.control_navigator_publisher = self.create_publisher(String, '/control_navigator', 10)
        self.localizer = self.create_client(FromLL, '/fromLL')
        while not self.localizer.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')

    def start_wpf(self, wps):

        wpl = []
        for wp_ll in wps:
            wp = latLonYaw2Geopose(wp_ll['lat'], wp_ll['lon'], 0.0)
            self.req = FromLL.Request()
            self.req.ll_point.longitude = wp.position.longitude
            self.req.ll_point.latitude = wp.position.latitude
            self.req.ll_point.altitude = wp.position.altitude

            log = 'long={:f}, lat={:f}, alt={:f}'.format(self.req.ll_point.longitude, self.req.ll_point.latitude, self.req.ll_point.altitude)
            self.get_logger().info(log)

            self.future = self.localizer.call_async(self.req)
            rclpy.spin_until_future_complete(self, self.future)

            self.resp = PoseStamped()
            self.resp.header.frame_id = 'map'
            self.resp.header.stamp = self.get_clock().now().to_msg()
            self.resp.pose.position = self.future.result().map_point

            log = 'x={:f}, y={:f}, z={:f}'.format(self.future.result().map_point.x, self.future.result().map_point.y, self.future.result().map_point.z)
            self.get_logger().info(log)

            self.resp.pose.orientation = wp.orientation
            self.pose_publisher.publish(self.resp)
            wpl += [self.resp]

        control_msg = String()
        control_msg.data = "Start"
        self.control_navigator_publisher.publish(control_msg)

```

```
def main():
    rclpy.init()

    gps_wpf = GpsWpCommander(default_yaml_file_path)
    wps = [{'lat': 0.0, 'lon': 0.0},]
    if len(sys.argv) > 1:
        wps = json.loads(sys.argv[1])
    gps_wpf.start_wpf(wps)


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

Listing D.5: Smart Navigator Node

```
import json
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator

class smartNavigatorNode(Node):
    def __init__(self):
        super().__init__('SmartNavigator')
        self.navigator = BasicNavigator()
        self.waypoints = []
        self.create_subscription(PoseStamped, '/gps_waypoints', self.
    add_waypoints, 10)
        self.create_subscription(String, '/control_navigator', self.
    control_navigator, 10)

    def add_waypoints(self, msg):
        self.waypoints.append(msg)

    def control_navigator(self, msg):
        if msg.data == 'Start':
            self.navigator.goThroughPoses(self.waypoints)
            self.waypoints = []
            self.write_nav_feedback()


    def write_nav_feedback(self):
        while not self.navigator.isTaskComplete():
            with open('/home/ubuntu/navigator_feedback/
    navigator_feedback.txt', 'w') as f:
                f.write(str(self.navigator.getFeedback()))


```

```python

def main(args=None):
    rclpy.init(args=args)
    node = smartNavigatorNode()

    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Listing D.6: Autonomous Navigation Launch File

```python
import os

from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument,
    IncludeLaunchDescription
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node


def generate_launch_description():
    tutorial_dir = get_package_share_directory('tutorial_pkg')
    map_yaml_file = LaunchConfiguration('map')
    params_file = LaunchConfiguration('params_file')
    use_sim_time = LaunchConfiguration('use_sim_time')

    declare_map_yaml_cmd = DeclareLaunchArgument(
        'map',
        default_value=os.path.join(tutorial_dir, 'maps', 'map.yaml'),
        description='Full path to map yaml file to load',
    )

    declare_params_file_cmd = DeclareLaunchArgument(
        'params_file',
        default_value=os.path.join(tutorial_dir, 'config', '
    navigation.yaml'),
        description='Full path to the ROS2 parameters file to use for
     all launched nodes',
    )

    declare_use_sim_time_cmd = DeclareLaunchArgument(
        'use_sim_time', default_value='false', description='Use
    simulation (Gazebo) clock if true'
    )

    ### ROSBOT_SIM
    rosbot_sim_file_dir = os.path.join(
        get_package_share_directory('rosbot_xl_gazebo'), 'launch', 'simulation.launch.py'
    )

    rosbot_sim_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([rosbot_sim_file_dir]),
    )

    ### NAVIGATION
    navigation_launch_file_dir = os.path.join(
        get_package_share_directory('tutorial_pkg'), 'launch', 'navigation.launch.py'
    )

    navigation_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([navigation_launch_file_dir]),
        launch_arguments={
            'map': map_yaml_file,
            'params_file': params_file,
            'use_sim_time': use_sim_time,
        }.items(),
    )

    ### IMU
    imu_node = Node(
        package='imu_remapping',
        executable='launch',
        name='imu_remapping',
        output='screen',
    )

    ### NAVSAT TRANSFORM
    navsat_node = Node(
        package='robot_localization',
        executable='navsat_transform_node',
        name='navsat_transform_node',
        output='screen',
    )

    ### WRITE GPS
    write_gps_node = Node(
        package='write_gps',
        executable='launch',
        name='write_gps',
        output='screen',
    )

    ### WEBSOCKET
    websocket_node = Node(
        package='autonomous_gps_navigation',
        executable='websocket_connection',
        name='websocket_connection',
        output='screen',
    )

    ### NAVIGATOR
    navigator_node = Node(
        package='smart_navigator',
        executable='launch',
        name='smart_navigator',
        output='screen'
    )

    ld = LaunchDescription()

    ld.add_action(declare_map_yaml_cmd)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(declare_use_sim_time_cmd)

    ld.add_action(navigation_launch)
    ld.add_action(rosbot_sim_launch)

    ld.add_action(imu_node)
    ld.add_action(navsat_node)
    ld.add_action(write_gps_node)
    ld.add_action(websocket_node)
    ld.add_action(navigator_node)

    return ld
```

Appendix E

Nodes and Launch files - Robot

Listing E.1: GPS Broadcaster Node

```
import serial
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import NavSatFix

port = '/dev/ttyACM0'
baud = 9600
gps = serial.Serial(port, baudrate = baud, timeout = 0.5)

class GpsBroadcasterNode(Node):
    def __init__(self):
        super().__init__('gps_broadcaster')
        self.gps_pub = self.create_publisher(NavSatFix, '/gps/fix',
    10)
        self.get_logger().info('Started!')
        self.get_logger().info('Publishing...')
        self.create_timer(0.1, self.publish_gps)
        self.i = 1

    def write_gps_position(self, coords):
        with open('/home/husarion/gps_position/gps_position.txt', 'w'
    ) as f:
            f.write(coords)

    def dms_to_dd(self, coord, dir):
        coord = int(coord) + (coord % 1)*100/60
        if dir in ['W', 'S']:
            coord *= -1
        return coord

    def read_gps_data(self):
        gps_line = gps.readline().decode().strip()
        latitude = 0.0
        longitude = 0.0
        altitude = 0.0
        try:
            if gps_line.find('GGA') > 0:
                gps_data = gps_line.split(',')
                if gps_data[2] != '':
                    latitude = round(float(gps_data[2])/100, 6)
                    lat_dir = gps_data[3]
                    latitude = round(self.dms_to_dd(latitude, lat_dir
), 6)
                if gps_data[4] != '':
                    longitude = round(float(gps_data[4])/100, 6)
                    lon_dir = gps_data[5]
                    longitude = round(self.dms_to_dd(longitude,
lon_dir),6)
                if gps_data[9] != '':
                    altitude = float(gps_data[9])
                alt_unit = gps_data[10]
        except Exception as e:
            self.get_logger().info(str(e))
        if latitude != 0.0:
            self.i = 1
            coords = f'{latitude}-{longitude}-{altitude}'
            self.write_gps_position(coords)
        self.i += 1
        return [latitude, longitude, altitude]

    def publish_gps(self):
        lat, lon, alt = self.read_gps_data()
        if lat == 0.0 and lon == 0.0:
            with open('/home/husarion/gps_position/gps_position.txt',
 'r') as gps_file:
                gps_line = gps_file.readline()
            lat, lon, alt = gps_line.split('-')
        msg = NavSatFix()
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'body_link'

        msg.latitude = float(lat)
        msg.longitude = float(lon)
        msg.altitude = float(alt)

        self.gps_pub.publish(msg)

```

```
def main(args=None):
    rclpy.init(args=args)
    node = GpsBroadcasterNode()

    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Appendix F

EC2 Setup

Listing F.1: Configuring system locale

```
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # verify settings
```

Listing F.2: Enabling Ubuntu Universal Repository

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Listing F.3: Adding GPG key for ROS2

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master
    /ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
```

Listing F.4: Adding ROS2 Repository

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/
    keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/
    ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo
    tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

Listing F.5: Installing Development Tools

```
sudo apt update && sudo apt install ros-dev-tools
```

Listing F.6: Installing ROS2

```
sudo apt install ros-humble-desktop
sudo apt install ros-humble-ros-base
```

Appendix G

Husarion Tutorials

Listing G.1: Installing Additional Packages

```
sudo apt install ros-humble-slam-toolbox
sudo apt install ros-humble-navigation2
sudo apt install ros-humble-nav2-bringup
sudo apt install python3-pip
sudo apt install ros-humble-gazebo-ros-pkgs
sudo apt install ros-humble-robot-localization
```

Listing G.2: Setting up Gazebo Environment

```
mkdir -p rosbot_ws/src
cd rosbot_ws
git clone https://github.com/husarion/rosbot_xl_ros src/
```

Listing G.3: Installing Work Tools

```
sudo apt-get update
sudo apt install ros-dev-tools
vcs import src < src/rosbot_xl/rosbot_xl_hardware.repos
vcs import src < src/rosbot_xl/rosbot_xl_simulation.repos
sudo rosdep init
rosdep update --rosdistro $ROS_DISTRO
rosdep install -i --from-path src --rosdistro $ROS_DISTRO -y
```

Listing G.4: Building Libraries

```
export HUSARION_ROS_BUILD=simulation
source /opt/ros/humble/setup.bash
colcon build --symlink-install
```

Listing G.5: Updating .bashrc

```
echo 'source ~/rosbot_ws/install/setup.bash' >> ~/.bashrc
```

Listing G.6: Creating Alias

```
echo "alias ROSBOT_SIM='ros2 launch rosbot_xl_gazebo simulation.launch.py'" >> ~/.bashrc
. ~/.bashrc
```

Bibliography

[1] A.B. Zachariah. *Precision Agriculture and the Future of Farming*. Arcler Education Incorporated, 2018. ISBN: 9781773612836. URL: https://books.google.it/books?id=jlAovAEACAAJ (cit. on p. 1).

[2] Prem Rajak, Abhratanu Ganguly, Satadal Adhikary, and Suchandra Bhattacharya. *Applications of integrated IoT and smart sensors for precision farming*. [Online; accessed May 10, 2024]. 2023. URL: https://commons.wikimedia.org/wiki/File:Applications_of_integrated_IoT_and_smart_sensors_for_precision_farming.jpg (cit. on p. 1).

[3] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005. ISBN: 9780262201629 (cit. on pp. 4, 5).

[4] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011. ISBN: 9780262015356 (cit. on pp. 4, 5).

[5] J. Kober, J. A. Bagnell, and J. Peters. «Reinforcement Learning in Robotics: A Survey». In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. URL: https://journals.sagepub.com/doi/10.1177/0278364913495721 (cit. on pp. 5, 6).

[6] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. «ORB-SLAM: A Versatile and Accurate Monocular SLAM System». In: *IEEE Transactions on Robotics*. Vol. 31. 5. 2015, pp. 1147–1163. URL: https://ieeexplore.ieee.org/document/7219438 (cit. on p. 6).

[7] J. Engel, T. Schöps, and D. Cremers. «LSD-SLAM: Large-Scale Direct Monocular SLAM». In: *European Conference on Computer Vision*. 2014, pp. 834–849. URL: https://link.springer.com/chapter/10.1007/978-3-319-10605-2_54 (cit. on p. 6).

[8] M. König, C. Wiesen, C. Heining, and W. W. Stürzl. «Real-time 3D SLAM with a Hand-held RGB-D Camera». In: *International Conference on Intelligent Robots and Systems*. 2011, pp. 3862–3867. URL: https://ieeexplore.ieee.org/document/6094807 (cit. on p. 6).

[9] W. Hess, D. Kohler, H. Rapp, and D. Andor. «Real-time Loop Closure in 2D LIDAR SLAM». In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278. URL: https://ieeexplore.ieee.org/document/7487258 (cit. on p. 6).

[10] S. Macenski, F. Martín, R. White, and J. Clavero. «Marathon 2: A Navigation System». In: *arXiv preprint arXiv:2003.00368* (2020). URL: https://arxiv.org/abs/2003.00368 (cit. on p. 7).

[11] M. Quigley, B. Gerkey, W. D. Smart, F. Martín, and S. Macenski. *ROS 2 Design*. 2014. URL: https://design.ros2.org/ (cit. on pp. 7–9).

[12] RTI Connext DDS. *Data Distribution Service (DDS) Overview*. 2021. URL: https://www.rti.com/products/dds (cit. on pp. 8, 9).

[13] ROS 2 Documentation. *Managed nodes (lifecycle) in ROS 2*. 2021. URL: https://design.ros2.org/articles/node_lifecycle.html (cit. on p. 8).

[14] G. Biggs and J. Karlsson. *An overview of the ROS 2 security architecture*. 2018. URL: https://design.ros2.org/articles/ros2_security.html (cit. on p. 9).

[15] Colcon Documentation. *Colcon - ROS 2's build tool*. 2021. URL: https://colcon.readthedocs.io/ (cit. on p. 9).

[16] ROS 2 Documentation. *Launch system in ROS 2*. 2021. URL: https://docs.ros2.org/latest/api/launch/html/ (cit. on p. 10).

[17] Open Robotics. *Gazebo Simulation*. 2021. URL: http://gazebosim.org/ (cit. on p. 10).

[18] Open Robotics. *Ignition Robotics*. 2021. URL: https://ignitionrobotics.org/ (cit. on p. 10).

[19] J. Zhang and S. Singh. «LOAM: Lidar Odometry and Mapping in Real-time». In: *Robotics: Science and Systems Conference*. 2014. URL: http://www.roboticsproceedings.org/rss10/p18.pdf (cit. on p. 10).

[20] J. Henrich and H. Wörn. *Robot Manipulation for Industrial Automation*. Springer, 1999 (cit. on p. 10).

[21] A. M. Okamura, L. N. Verner, R. H. Taylor, C. Simone, and P. Kazanzides. «Teleoperation: An Enabling Technology for Healthcare, Manufacturing, and Space Exploration». In: *Technology* 5.3 (2005), pp. 7–13 (cit. on p. 10).

[22] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer, 2016 (cit. on pp. 10, 11).

[23] Peter Mell and Timothy Grance. «The NIST definition of cloud computing». In: *NIST special publication* 800.145 (2011), p. 7 (cit. on pp. 12–15).

[24] Michael Armbrust et al. «A view of cloud computing». In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on pp. 15, 17, 20).

[25] Robert Baker. *AWS: The Complete Guide from Beginners to Advanced for Amazon Web Services*. CreateSpace Independent Publishing Platform, 2018 (cit. on pp. 17, 18).

[26] itim2101. *Icon by https://https//dribbble.com/itim2101 on freeicons.io*. URL: https://freeicons.io/robotics-icon-set-5/robot-icon-268754 (cit. on p. 23).

[27] Husarion. *ROSbot XL github repository*. This work is licensed under the following licenses: Apache-2.0: https://github.com/husarion/rosbot_xl_ros/blob/e48acb6d3efe7c7a5e1dddaec43b640ab4870c26/LICENSE_APACHE2.txt MIT: https://github.com/husarion/rosbot_xl_ros/blob/e48acb6d3efe7c7a5e1dddaec43b640ab4870c26/LICENSE_MIT.txt. URL: https://github.com/husarion/rosbot_xl_ros.git (cit. on p. 28).

[28] ROS Navigation. *ROS navigation - navigation2*. This work is licensed under the following licenses: Apache-2.0: https://github.com/husarion/rosbot_xl_ros/blob/e48acb6d3efe7c7a5e1dddaec43b640ab4870c26/LICENSE_APACHE2.txt BSD-3-Clause: https://opensource.org/license/bsd-3-clause LGPL-2.1-or-later: https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html. URL: https://github.com/ros-navigation/navigation2.git (cit. on p. 30).

[29] AWS. *NICE DCV. High-Performance Remote Display Protocol*. 2023. URL: https://aws.amazon.com/hpc/dcv/ (cit. on p. 32).

[30] AWS. *AWS Security Best Practices*. 2023. URL: https://aws.amazon.com/architecture/security-identity-compliance/ (cit. on p. 32).

[31] AWS. *NICE DCV Supported Platforms*. 2023. URL: https://docs.aws.amazon.com/dcv/latest/userguide/what-is-dcv.html (cit. on p. 32).

[32] AWS. *AWS Scalability Benefits*. 2023. URL: https://aws.amazon.com/autoscaling/ (cit. on p. 32).

[33] AWS. *NICE DCV Collaboration Features*. 2023. URL: https://docs.aws.amazon.com/dcv/latest/userguide/working-with-shared-sessions.html (cit. on p. 32).

[34] AWS. *AWS Cost Management*. 2023. URL: https://aws.amazon.com/pricing/ (cit. on p. 33).

[35] AWS. *NICE DCV User Guide*. 2023. URL: https://docs.aws.amazon.com/dcv/latest/userguide/ (cit. on p. 33).

[36] Charles River Analytics. *Robot Localization*. This work is licensed under the following licenses: https://github.com/cra-ros-pkg/robot_localization/blob/f2b36f761ffd6f7b07c98b6c748c4d75660098b9/LICENSE. URL: https://github.com/cra-ros-pkg/robot_localization.git (cit. on p. 37).

[37] ROS Navigation. *Nav2 GPS Waypoint Follower Demo*. This work is licensed under the following licenses: MIT: https://mit-license.org. URL: https://github.com/ros-navigation/navigation2_tutorials/tree/cb800b76d40d18229f1355807351a138b1123ec5/nav2_gps_waypoint_follower_demo (cit. on p. 37).