Ain Shams University
Faculty of Engineering
Computer Engineering and Software Systems

Graduation Project Thesis

Cloud-Connected Automated Guided Vehicle

Supervised By:

Prof. Dr. Watheq El-Kharashi - Dr. Mohamed AbdelSalam

Submitted By:

Sara Akmal Mohamed	17P6015
Mariam Salah Taha	17P6033
Fatema Hassan Aman	17P3060
Islam Ashraf Azab	17P6086
Yusuf Amr Nabil	17P6052
Omar Hassan ElKotb	17P6080

June 30, 2022

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance.

We would first like to thank our supervisor, Dr. Watheq El-Kharashi, whose expertise was invaluable in formulating the research questions and methodology.

We would also like to thank our tutor, Dr. Mohamed Abdelsalam, for his valuable guidance throughout our project. He provided us with the tools needed to choose the right direction.

In addition, we would also like to show gratitude to Siemens EDA Company for their cooperation and support.

Finally, we would also like to thank Eng. Mahmoud Selim who was always there for any questions and provided us with insightful comments and immense knowledge.

Abstract

The growing need of services offered by robots in our daily lives contributed in raising the different utilization for them. Our study was based on services offered Over the Air.

Previous research has primarily relied on static implementation of services such as Sensing, Actuating and Tensoring. These services are cloud-based; supported by AWS and are visualized using a car implemented by Raspberry-Pi Board.

In our work, we aim to automate the Cloud-Connect Infrastructure between any device and the docker'ized cloud. The automation comes from supporting variable number of sensors and services according to the user's configuration which is entered through a Qt GUI displaying the output on a dashboard running on the private cloud.

ROS Nodes (DUT) are coded to stimulate the launching of this infrastructure using an Ubuntu18.04 connected to a Jetson Board. However this Cloud-Connect Package is implemented to run on different boards due to generalization reached.

Contents

List of Figures

1 Introduction

Cloud-based vehicle systems is becoming more and more a booming topic and an interesting area of work for a lot of researchers. As a continuation to a previous thesis concerning this topic, we decided to improve the previously proposed methodology.

The previous work was aiming to provide cloud-based services through a robot. This system was based on two modules: the dashboard and ROS (Robotics Operating System). The dashboard component represents the user-interface, where the user can visualize the robot's sensed data and can monitor the robot's actions. The user can also control the robot's parameters, such as the speed, through this GUI. The ROS module is responsible for managing the robot and interfacing with its hardware. The communication between these two modules was done through sockets.

This approach was limited in terms of flexibility and reusability, since the user didn't have the option to freely select the number of the sensors, their type and the application desired. These configurations were already predefined and cannot be changed.

As a result, this thesis aims to automate the CloudConnect infrastructure, connecting the device to the cloud, to support variable number of sensors and services according to the user's configuration which is entered through a QT GUI. This is displayed through a dashboard running on a Docker'ized Private Cloud which contains SQL Database and OTA (Over-The-Air) services; sense, actuate, etc. Another improvement is using Jetson board because it has higher processing power than Raspberry-Pi. The CloudConnect Package will be implemented to run on different boards.

While working on our approach, we chose two related articles to our work. The first application presented a cloud-based tracking vehicle system, which was similar to our docker'ized cloud logic and also used close tools to the ones we're using, such as the way of handling the HTTP requests between the vehicle and the dashboard.

The second application presented a service robot platform that's able to map, localize itself and navigate in indoor environments. It was useful since we're aiming to develop a similar ROS module as a case study for our project.

Walking briefly into the data flow between the different components of our application, we present the following diagram. [1]

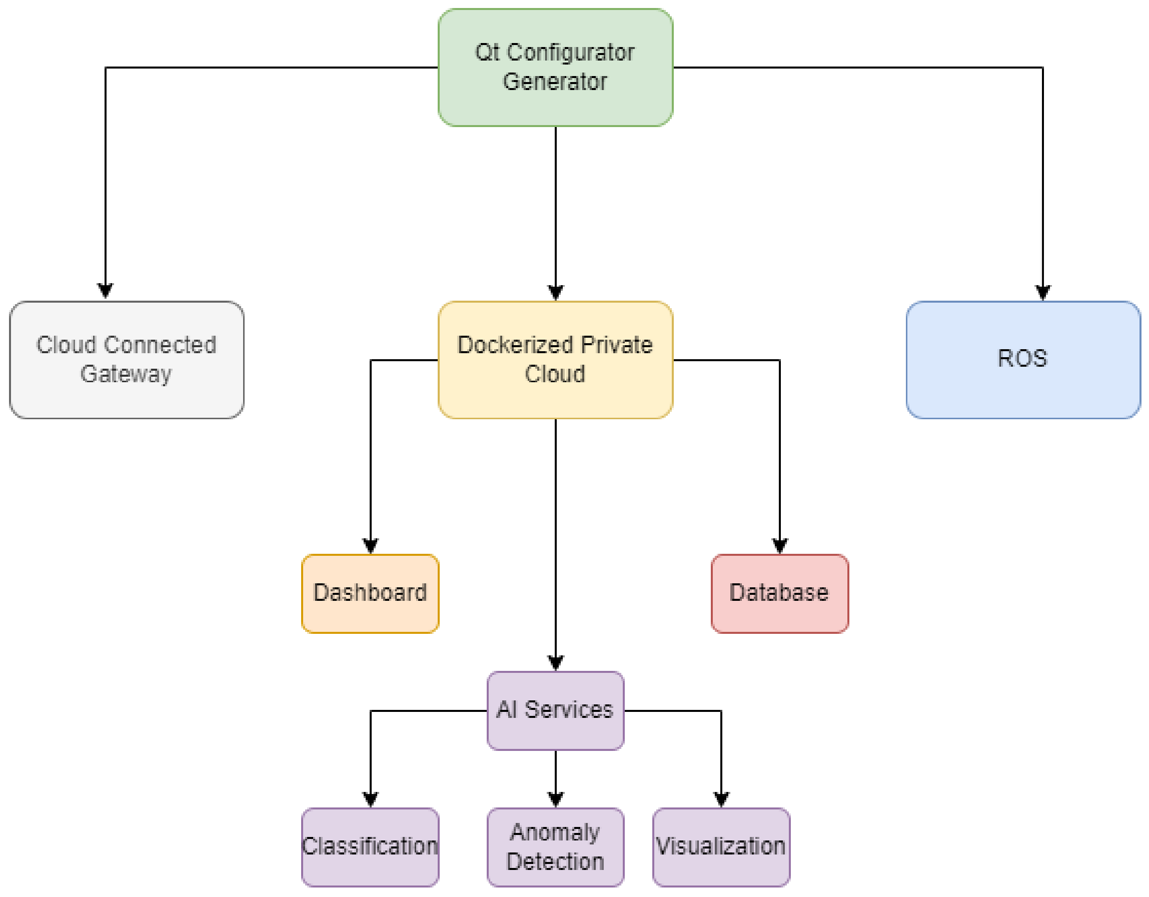


Figure 1.1: Abstract diagram

It starts with the user entering the sensors and their configuration using the QT, Then it generate Dockerized Private Cloud , Cloud Connected Gateway and ROS Subsystem. Dockerized Private Cloud consists of Dashboard, Database and AI Services.The configuration Data sent to the dashboard to create the database tables accordingly. Then the application keeps running as the dashboard and ROS exchange data through Cloud Connected Gateway. ROS module can update the database while sensing, while actuating from dashboard, dashboard sends data to ROS then ROS updates the Database.

In our thesis, we'll go into more details as we present the related papers, how we formulated the problem and talking more about the previous work. We'll then expand to our proposed methodology and the software requirements of our system. Finally, we'll discuss the future work that can be done to improve the approach.

2 Related Work

Cloud-Connected vehicles are subject to a lot of research papers nowadays as they provide variable services in a reliable way. While studying the topic, we chose two related applications' methodologies. The first paper presented a cloud-based tracking system and the second discussed a service robot platform.

In this section we'll be discussing both papers in further details, relating them to our proposed work.

The first paper is studying an cloud-based anti-theft vehicle tracking system that needs to transmit data through HTTP requests to the server to keep tracking and monitoring the vehicle in real-time.

There was a comparison between sending the data to a classic server with one administrator and sending it to the cloud which led to choosing the cloud option. This is because the cloud-based servers provide reliability, flexibility, scalability, and accessibility. It also facilitates the data sharing between different users and admins through an attached database. For this purpose, Mocha host web-services are used to store the needed data on the cloud. [4]

For this system to provide real-time services, the embedded system attached to the vehicle sends data every 10 seconds to be directly displayed to the user on the web application and saved to the database.

The server side consists of 3 different components: database, user-interface, and Google Maps.

The database used phpMyAdmin, which is based on SQL, to establish the connection and control the data flow. The user-interface components handles the user requests to the system such as requesting to retrieve vehicles position at a certain time by entering the start and end dates of the wanted duration. This is through the web application, which is based on HTML, CSS, JavaScript, and XML (Extensible Mark-up Language) for the data to be sent to the server side. It also uses AJAX to handle HTTP requests. Lastly, Google Maps is the component displaying the location coordinates based on the database and the data received from the vehicle. It auto-updates the coordinates shown every 10 seconds if new location details are received, otherwise it displayed the last received location. [4]

This sums up the first topic related to the cloud-based tracking system and the tools used to build it. Projecting on our paper, since our focus is developing a similar system but with a generalized purpose, we can see that the core logic is almost the same. We can also note the similarity in the tools used with the difference in the hardware component. The paper was using a simple Arduino board with minimal configurations, whereas we are planning on developing a complete ROS system managing the vehicle. For this reason, we chose the topic of the second paper, which discusses a ROS-based robot platform.

This paper proposes a methodology to develop a robot capable of mapping, localizing itself and navigating in an indoor environment which is best suited for structured environments with static obstacles.

This approach uses two components that are reflected in our work; ROS and Gazebo.

Starting with ROS, which is an open source Linux based framework for operating robots in which executable programs, referred to as nodes, communicate with each other via publishing or subscribing messages to topics. [5] This means that when a node needs data it subscribes to the relevant topic and the nodes sending the data publish messages to the topic. One of the main advantages of ROS is that it doesn't have a single point of failure making it more reliable when developing real-time systems. So, even if one of ROS components fails, it doesn't stop the entire robot from functioning. [3]

The second component, Gazebo, is an open-source 3D robotics simulator and a ROS package. It's a very powerful tool since it's very flexible in terms of simulating most of the robotics scenarios. It's considered an important pre-production phase to test and verify the application in a way that doesn't risk harming the actual robot hardware.

These two tools were the core of the case study explained in the paper. The robot is aimed to study its surrounding environment and map it to be able to localize itself and move around. This is done through a graph based Simultaneous Localization and Mapping approach, called RTAB-Map, using a ROS package called rtabmap. This also solves the kidnapping problem using muti-session mapping, meaning that it gives the robot the capability to be able to map common locations if placed in a different environment. The robot, using this map, will be able to navigate its way through the area. [3]

This is briefly what the second paper was discussing, which can be mapped to the ROS module of our work. Since we plan to develop a similar case study to test our cloud services. Ending up with ROS's main edges, which is code reusability giving us the option to reuse previous work easily, and facilitating the generalization process in our thesis.

3 Problem Formulation

3.1 Software Architecture Models

A software architecture is the system's structure on a high level. Architecture models and code architectures play an important role to represent the communication between components in large software systems. Therefore, to enhance and maintain the software system, the architectural model should be decided according to the use of the system. We are going to mention two models for comparison and an attempt to solve our problem.

3.1.1 Static Software Architecture

First model is the static software architecture; where the system structure is fixed and can't be configured. In addition, after initialization it is not equipped to have new connections between the system components and new components. Moreover, existing connections can not be destroyed [6].

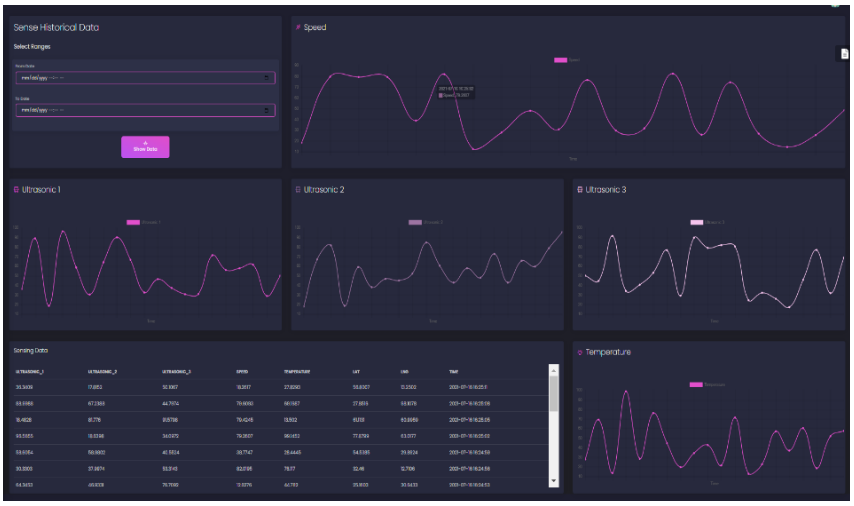
3.1.2 Dynamic Software Architecture

Second model is the dynamic architecture; a structure that is developed for evolution during runtime. The system's components can be generated or destroyed according to the design rules. These kinds of systems are subject to constant change by a certain trigger event or user request. Dynamic architectures aim to synchronize between several elements of the current configuration so all components are connected and operating together [6].

3.2 Problem Context

The application consists of two main infrastructures; the dashboard and the ROS component. In the previous work the dashboard is defined in a static manner according to the ROS component where it visualizes data for the ROS Nodes in real time or may be used later to perform some analytics on the data. There are different services the ROS structure has, sensing which takes real time sensor measuring to display on time series line graph, AI Services where the dashboard receives a livestream from ROS to display and perform image recognition on some objects and lastly the actuate service where the user can control the speed and angle control from the Dashboard and change their values affecting the ROS subsystem.

For example, in the following diagram displaying the dashboard, three ultrasonic graphs, one speed chart and one temperature are defined for analytics therefore the dashboard isn't flexible for any number or types of sensors it receives and must be implemented in the project while it's being created. Here the problem arises which is a fixed dashboard that isn't compatible with different types of ROS applications.



Sensing Data

ULTRASONIC_1	ULTRASONIC_2	ULTRASONIC_3	SPEED	TEMPERATURE	LAT	LNG	TIME
36.3409	17.8152	50.1067	18.2617	27.8293	58.8007	13.2502	2021-07-16 18:25:11
83.5968	67.2383	44.7974	78.6093	68.1567	27.8516	68.1078	2021-07-16 18:25:08
18.4828	81.776	81.5798	79.4245	13.902	61.131	63.8969	2021-07-16 18:25:05
93.5855	18.8398	34.0972	79.2607	89.1452	77.8799	63.3177	2021-07-16 18:25:02
53.6054	58.8802	40.5524	38.7747	28.4445	54.5385	29.8924	2021-07-16 18:24:58
30.3303	37.9874	53.3143	82.0195	78.77	32.46	12.7106	2021-07-16 18:24:56
64.3453	46.9308	76.7095	12.8376	44.783	25.1800	30.5433	2021-07-16 18:24:53

Figure 3.1: Old Historical Data

This problem might affect new users that need this dashboard for their specific ROS application. Each new application would require its own type and number of sensors to display, in addition to applying the services that are available in the dashboard.

Not only the visualizing of data is fixed but the database structure would also need to be flexible according to the new ROS applications. The current database has one table configured for these defined sensors while new applications will require different number of tables to store the data in.

Due to the project size and its complexity we need to use a dynamic software architecture and decompose the system into smaller units for the sake of reusability and make sure to operate for different ROS design alternatives and variations of the old application.

3.3 Objective

To achieve a dynamic architecture, we should generalize the structure of the current dashboard and database. Generalization in software is the process of extracting shared characteristics from two or more classes and combining them into a superclass, so members of the specialization classes will share some the super class properties yet will also have some differences. This approach aims to frequently reuse these classes without the need to write the same properties every time we need a new component. Similarly, we will use this approach to refine the code structure of the dashboard and database to be dynamically compatible with different sensors and subsequently, new users will be able to use the dashboard easily without configuring the code to their needs and make it compatible with ROS sensors.

4 Previous Work

We aim to refine and construct a ready to use solution for production of the previously developed project over the air remote driving. To make full use of the solution we need to compare the previous work to our method to develop a more dynamic structure. In this section we will define over the air (OTA) remote services, then we'll talk about the technologies used for the application and how each component communicates with the others.

4.1 OTA Services

Over-the-air services provide ways to transfer/receive data wirelessly using the cloud, globally or locally. The previous work provided multiple services using the cloud (AWS), including sensing, actuating and tensoring which will be discussed in the following subsections.

4.1.1 Sensing

Capturing sensor data from different sensors such as ultrasonic, temperature, speed sensors and camera that are mounted on the vehicle used, which was a car based on Raspberry-Pi microcontroller. This data is subsequently sent over the cloud to future analysis and streaming.

4.1.2 Actuating

Providing the user with the option to control the vehicle based on previously sensed data. Controlling could be in the form of changing the car's speed, direction or the steering angle for the example.

4.1.3 AI Services

Using the cloud to provide AI services such as object or lane detection based on the collected sensed data by running different machine learning models.

4.2 Technologies

The application consists of two infrastructures, each is working separately. ROS subsystem running on Raspberry pi and the Dashboard running on a local network.

4.2.1 Web Frameworks

Django - Backend

The dashboard is implemented by using the Django framework. The framework follows the Model-View-Template Software Design Pattern and it eases web development process by saving a lot of time with the use of only one framework. The dashboard contains three services; sensing, actuate and displaying historical data. Sensing displays the real time data received from ROS sensors, actuate displays some controls for the user if they want to change the angle control, direction or speed. It also represents a live tracking map and live stream from ROS. Historical data displays the measured data according to specified date from the user.

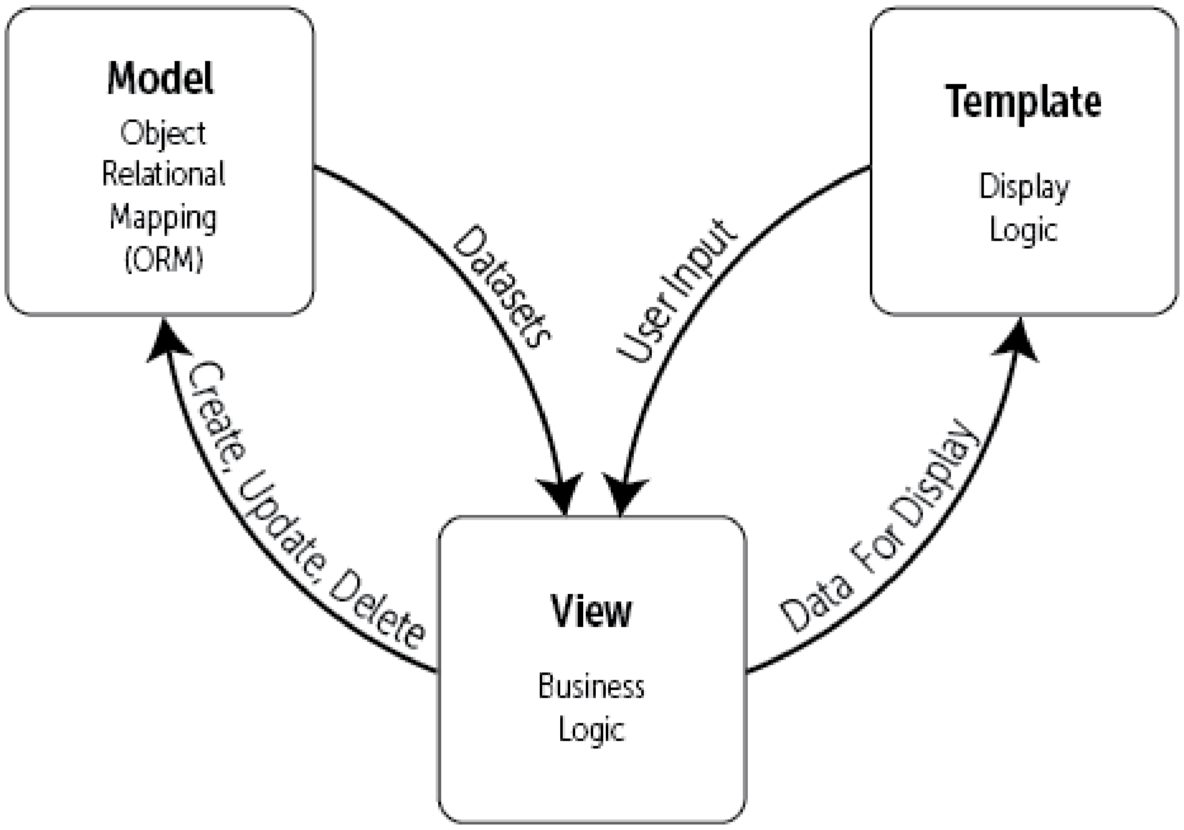


Figure 4.1: Django MVT

In the Django project we would like to refactor the frontend section of the three main pages; sense, actuate and historical data to represent any number of sensors the user requires. In addition, we need to refactor the backend for the database configuration to be consistent with our new dynamic frontend pages.

Frontend

Frontend is based on HTML, CSS and JavaScript. The web-pages designs is mainly based on Bootstrap functionalities. jQuery is the main library used in the JavaScript code in order to use AJAX functions aiming to communicate data between frontend and backend using HTTP requests.

4.2.2 SocketIO

SocketIO supports multiple communication between different applications by using the Socket.IO protocol which defines the namespace and events of the socket. In the application context, it's used to communicate between ROS Nodes and the dashboard by sending real time data to the dashboard and the database for storage.

4.2.3 Docker

In the previous project the web application had two options to serve different users, one choice was to AWS if they prefer an industrial solution for visual analysis and amazon cloud for logging and monitoring. The other solution is to use the docker'ized private cloud for users that prefer to provide more data security by using their own visualization and analysis application. Since our project aims for an industrial solution but at the same time, we want our own implementation of logging data and a general dashboard that accepts any types of sensors, we will choose to use docker for our web application and tune it to our needs.

4.2.4 ROS

Last year's project ROS system was intended to be applied for a specific hardware, not generalized like we are intending to do. To elaborate, here is the description for the whole ROS system of the project. Generally, some nodes were running on Raspberry Pi connected with AWS cloud platform, other nodes running on laptop connected with docker container.
Closely, the ROS master which is the main and initial node operates on the local laptop. Other nodes are relatively tiny and independent programs that could be running simultaneously on several systems.
Nodes running on the Raspberry Pi:

- actuateReceiver: adjusts the actuate and steering angle of the car
- sonar array: sends the values of the three ultrasonic
- temp: sends the values of the temp sensor
- speed: sends the values of the speed sensors
- gps: sends the location of the gps module
- usb cam: sends the stream of the camera module
- AWS_CloudMetrics: creates a message of ros_monitoring_msgs/MetricList

- cloudwatch_metrics_collector: publishes your robot's metrics to the cloud
- h264_video_encoder: encode a stream of images into an H264 video stream
- kinesis_video_streamer: enables robots to stream video to the cloud for analytics, playback, and archival use

Nodes running on the laptop:

- Actuate and Sensing: Gets actuate and steering values from the dashboard and sends the value of sensors to the dashboard
- Streaming Services: Receives streaming form the raspberry and sends it to the dashboard

Topics:

- speedanddirection: steering and actuate data from dashboard to the ROS node
- metric data: send metric's name and value to the docker
- metrics: send all needed data to the AWS cloud

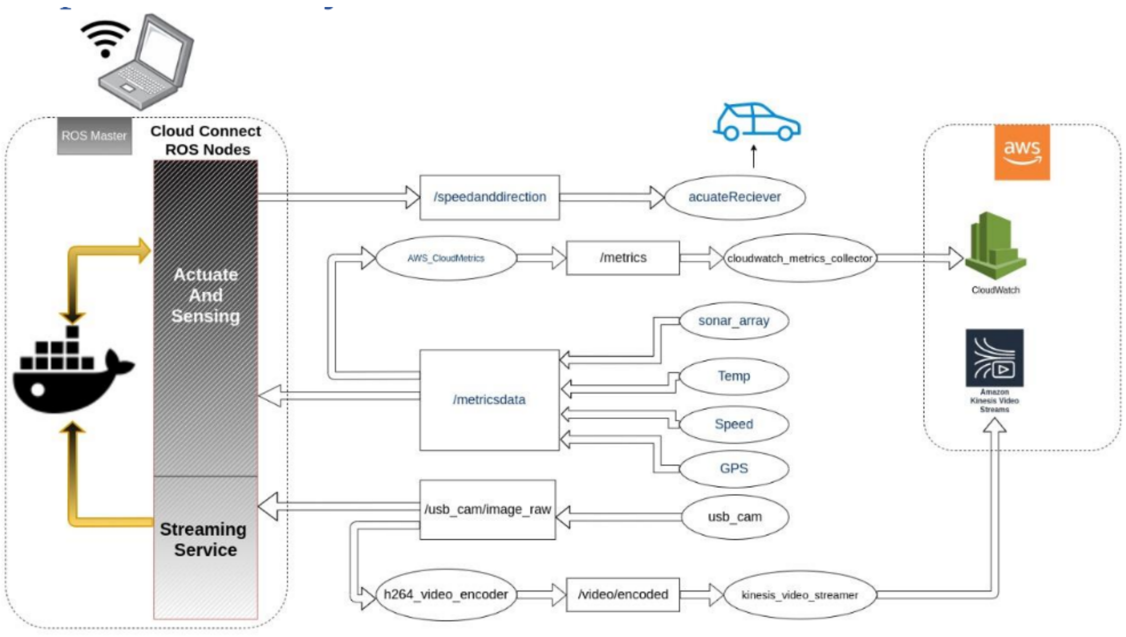


Figure 4.2: Previous ROS System

5 Proposed Method

5.1 Database

5.1.1 DB Overview

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data.

5.1.2 DBMS

Database Management Systems (DBMS) are software systems used to store, retrieve, and run queries on data. A DBMS serves as an interface between an end-user and a database, allowing users to create, read, update, and delete data in the database.

DBMS manage the data, the database engine, and the database schema, allowing for data to be manipulated or extracted by users and other programs. This helps provide data security, data integrity, concurrency, and uniform data administration procedures.

DBMS optimizes the organization of data by following a database schema design technique called normalization, which splits a large table into smaller tables when any of its attributes have redundancy in values. DBMS offer many benefits over traditional file systems, including flexibility and a more complex backup system.

Database management systems can be classified based on a variety of criteria such as the data model, the database distribution, or user numbers. The most widely used types of DBMS software are relational, distributed, hierarchical, object-oriented, and network.

5.1.3 MySQL

We choose to use MySQL to be our DBMS.MySQL is the world's most popular open source database. With its proven performance, reliability and ease-of-use, MySQL has become the leading database choice for web-based applications, covering the entire range from personal projects and websites, via e-commerce and information services, all the way to high profile web properties including Facebook, Twitter, YouTube, Yahoo! and many more.

5.1.4 Database Schema

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It's the database designers who design the schema to help programmers understand the database and make it useful.

5.1.5 Database Implementation

We make our data base dynamic, we have basic table which is created to each sensor, Dashboard takes number of sensors established by user in his project and type of each sensor, then it creates a table for each sensor which consist of sensor name and value of sensor.

After creation the data base, each sensor assign it's value from ROS nodes, Dashboard reads database and Show them either in actuate or in sense page.

5.2 Docker

Docker is an open-source platform that enables developers to containerize their application into standard executable components. Containers simplify the process of running distributed applications; developers will not go through the inconveniences of configuring the dependencies required to run the code in different environments because the container already has all the requirements ready to deploy the application. Containers logical mechanism is to abstract the application from the environment. This makes developers predict how their application may be utilized on different environments from their own as the application is independent of the environments it is deployed on.

5.2.1 Docker Benefits:

Docker would assist us in our project in different approaches.

- Docker containers decrease the deployment time
- It will automatically build a container from the application source code
- Regardless where the application is deployed everything would remain consistent and this leads to less time debugging and being sure that our application will perform exactly as we tested it.
- Docker images are not constrained with environmental limitations making it scalable.
- Docker containers provides us with better performance since they are much faster to create than using virtual machines

5.2.2 Docker Tools:

Docker-File: Docker starts with a text file including all the instructions to build docker images by automating the process of creation and defining the application's environment

Docker-compose: It has the services which are needed for the application to be deployed so it could run in the created environment. Each service is running on a different port on the local host inside one container for maintainability. Services included in our project are the following:

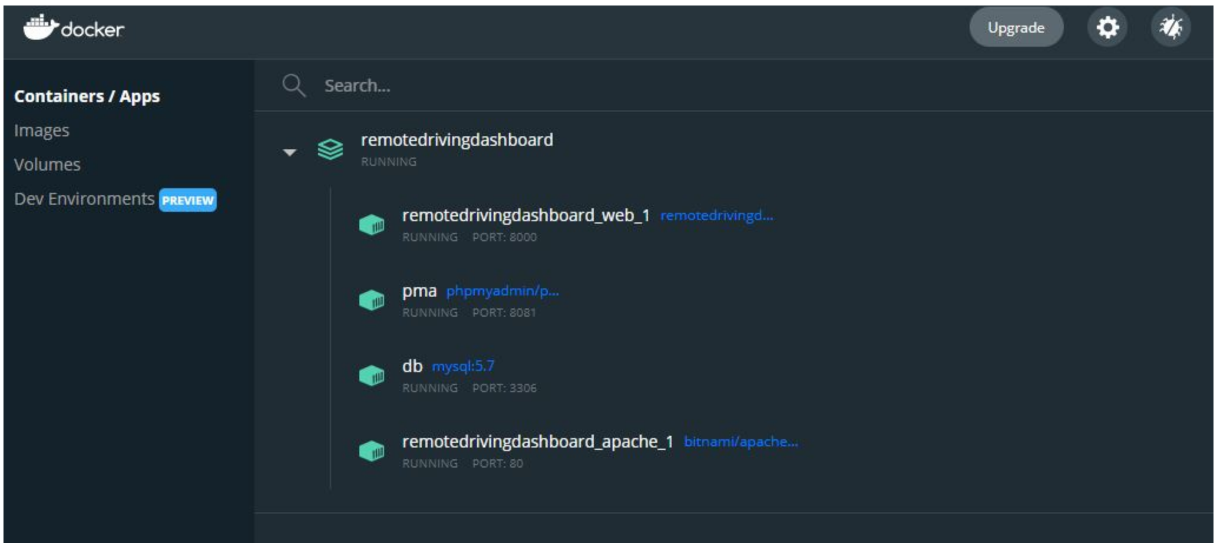
- db: MySQL performance, reliability and ease-of-use is one of the best choices for the web-based application to store our data in.
- phpMyAdmin: It provides handling of MySQL over the web with a friendly user interface to browse, drop databases, tables, views, fields and indexes.
- Apache: phpMyAdmin interface is based entirely in our browser so we will need a web server such as Apache to install phpMyAdmin's files into.
- web: The image where the dashboard will be deployed on

5.2.3 Images:

The process of creating images should be easily done for the developers and does not take any time, thus the process is automated since all the services are already defined in the docker-compose.yml file. We will only need two commands to start the container and four images

1. Start by creating the container with docker-compose build command
2. Then for building the images docker-compose up command is used

All requirements will be downloaded automatically as dependencies are already stored in the requirements.txt file then images will be created from the information in the docker-compose.yml file. Each image has a defined port in the text file. Finally, when all the images are deployed and running, any service can be used from the browser.



```
remotedrivingdashboard
RUNNING
    remotedrivingdashboard_web_1  remotedrivingd...
    RUNNING  PORT: 8000
    pma  phpmyadmin/p...
    RUNNING  PORT: 8081
    db  mysql:5.7
    RUNNING  PORT: 3306
    remotedrivingdashboard_apache_1  bitnami/apache...
    RUNNING  PORT: 80
```

Figure 5.1: Docker

5.3 SocketIO

Socket.IO is a transport protocol that enables real time communication between server and client based on various events. In our application we use Socket.IO in each separate component to enable full communication between our application. We define event handlers in each component to invoke callback functions to acknowledge that the other side has processed the event. In the following sections we are going to define the methods we need to communicate between different modules in our application.

5.3.1 Namespaces

A namespace is a communication channel that splits the logic of one application to receive different requests from other connections so it can handle each request separately calling the methods it needs, this is called multiplexing. Each namespace has its own event handlers, rooms and middle wares. For example, if we want to check if connection is established between our server and a client, we will use on connect event handler with the name space we choose so an alert will notify us if the connection is initiated.

5.3.2 Event Handlers

There are several methods to handle events that are transmitted between the clients and server. On the client side the socket instance uses the component-emitter library to emit an event using the emit() method. The single argument the method takes is the data passed to the server, this data can be dict, str or list. There can be a callback function when an emit method is called to handle some event.

5.3.3 Server

The dashboard which is implemented by the Django framework acts as the server to handle different requests coming from ROS nodes and the QT application. Our main concern is to create a dynamic dashboard upon the user request, this means we will need some information from the desktop application. The user will enter his required number of sensors to display on the dashboard and this information will be sent from the desktop application to the dashboard by a socket with a logical connection using the same namespace and event name. After the server receives the data, it will be handled by the frontend side to display the new charts.

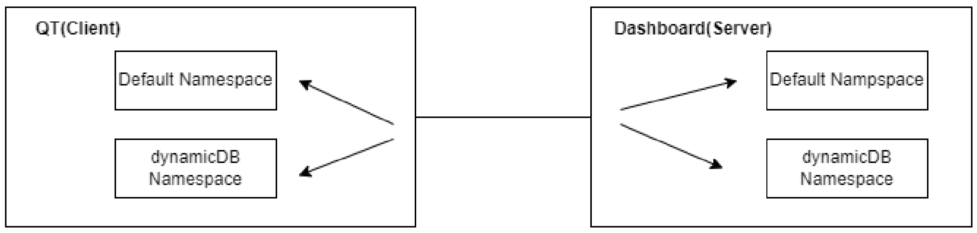


Figure 5.2: Multiplexing

5.3.4 Client

The application contains two main clients; a desktop application for the users to enter their configurations and the ROS subsystem that sends real time data over the sockets to the dashboard. There are different types of data that are sent over the sockets, there is real-time sensor data which can be displayed on the charts, a live-stream from the subsystem camera and the current location using GPS.

QT

QT application is a graphical user interface that takes input from the users. A submit button is set so the user can click on it when they enter all the required data. When this submit button is entered, an emit() method is invoked and a dictionary with the information the user entered is sent over to the dashboard using the socket with the namespace of dynamicDB.

ROS

ROS is an open-source, meta-operating system for your robot. [2] We are going to get into depth in the next chapter. The connection of the whole system can be described as a sequence of nodes communicating with each other.

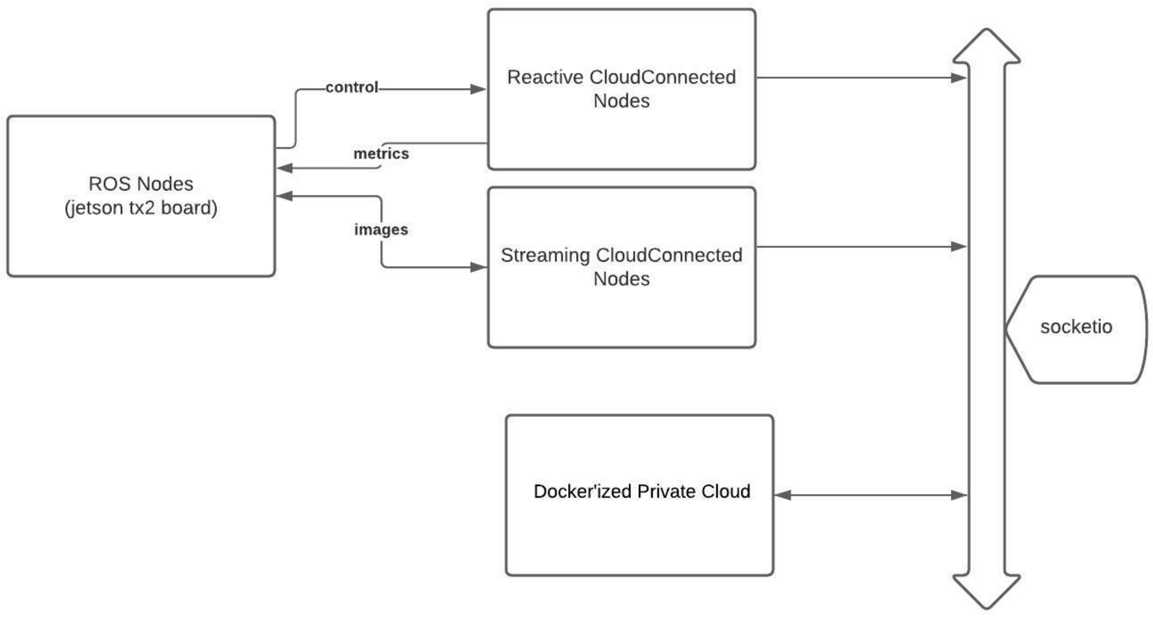

ROS Nodes
(jetson tx2 board)
control
metrics
images
Reactive CloudConnected Nodes
Streaming CloudConnected Nodes
Docker'ized Private Cloud
socketio

Figure 5.3: ROS Overview Modules Communication

5.4 ROS

[2, 5] ROS (Robot Operating System) is a software development kit which is open source used for robotics applications. It provides a standard platform for developers that help them reach the deployment and production step by saving time spent on research and prototyping. In other words, don't reinvent the component. By building ROS, you can have the availability to create new or faster things otherthan building it from scratch. ROS offers vast tools, libraries, as well as capabilities that are needed to develop your robot, by letting you spend more time on the development and innovation that is essential for your project. Due to the fact that it is open-source, it offers the flexibility to take a decision on the way and the parts you are welling to use ROS. Additionally, the total freedom to adjust it for your demands. Furthermore, ROS isn't, as many expecting, exclusive, you are not supposed to opt between ROS or some other software; ROS slightly merge with the existing software. Many robotics applications are already uses the open-software development kit and may appear in several types of industries for example indoor and outdoor, underwater and space, both home and automotive. ”distros” stands for distributions, similarly called a package of releases gathered together. ROS 2 supports working on Linux, Windows, macOS, and other embedded platforms by micro-ROS. During our project, we are using melodic.

5.4.1 Nodes

Primary ROS modules are the ROS Nodes and one of the main function of ROS is to make the communication between this modules easier. In short, these nodes contains the code that is executable, nodes can be totally residing on one computer, or distributed between several robots and computers.

```
rosnode list
```

5.4.2 ROS Master

Metaphorically, we can imagine the master as the main function in any programming language, as the similarity in the start of the ROS program always by launching this ROS master and it controls the communication between other nodes. It keep tracking to publishers and subscribers of each specific topic. One more job of the ROS master, it to enable ROS nodes to allocate on another. By using TCP/IP protocols which are called TCPROS in ROS, nodes have the ability to communicate with each other P2P. Example here: [5]

At first the ROS node camera wants to publish to the topic images.

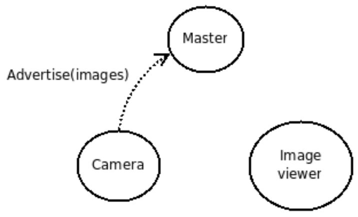


Figure 5.4: ROS Master 1

Afterwards, the image viewer node requests the images (subscribe to the image topic) to represent them.

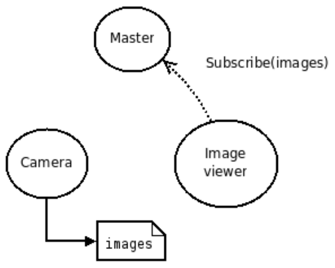


Figure 5.5: ROS Master 2

Now topic image has both publisher and subscriber, the master node tells the two nodes about the existence of each other and have the ability the transfer images between each other.

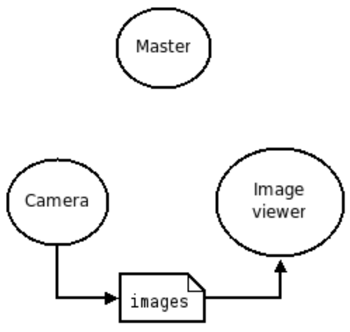


Figure 5.6: ROS Master 3

5.4.3 ROS Message

Like datatypes in any programming languages, in order for the message to be specified for a predetermined number of bytes stored.

5.4.4 ROS Topics

The core component that is responsible for the communication between the nodes. As mentioned before, the function of ROS is to ensure the communication between its nodes. Topic is the sort of information. Each topic is assigned to a type(s) of message(s). One of the core reasons to use topics is to have a unidirectional communication, streaming in other words, for receiving and sending communication, you should go for the service.

To list all topics:

```
rostopic list
```

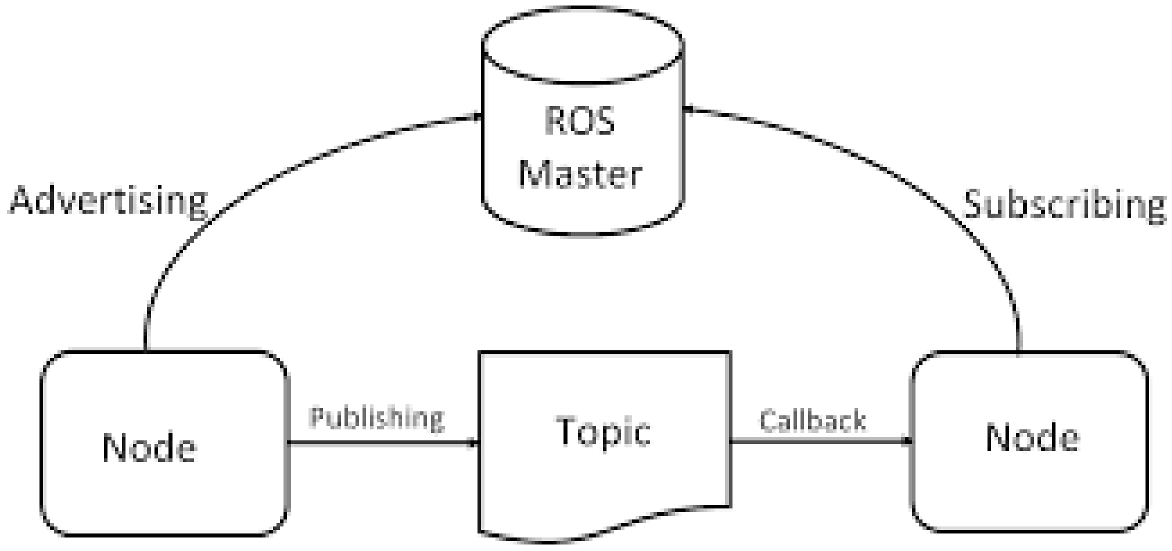


Figure 5.7: ROS Overview Components

5.5 UML System Diagrams

5.5.1 Use Case Diagram

To present the dynamic behaviour of the system and its functionality, and to summarize the relationships between modules and how actors interact with them, we constructed the following use case diagram.

The diagram represents the functionalities that the user should use in order to use the system efficiently. He should enter configurations to Qt configurator generator then it launches the ROS module that will activate his desired application, Dockerized Private Cloud which contains AI Services, Database and Dashboard.

The diagram also shows how different components interact with each other. It points out that the dashboard creates the database tables, that the Qt sends the sensors information directly to the dashboard, that the Render Historical Page retrieves data from the database, and that the ROS nodes can send the sensed data to be saved in the database. [1]

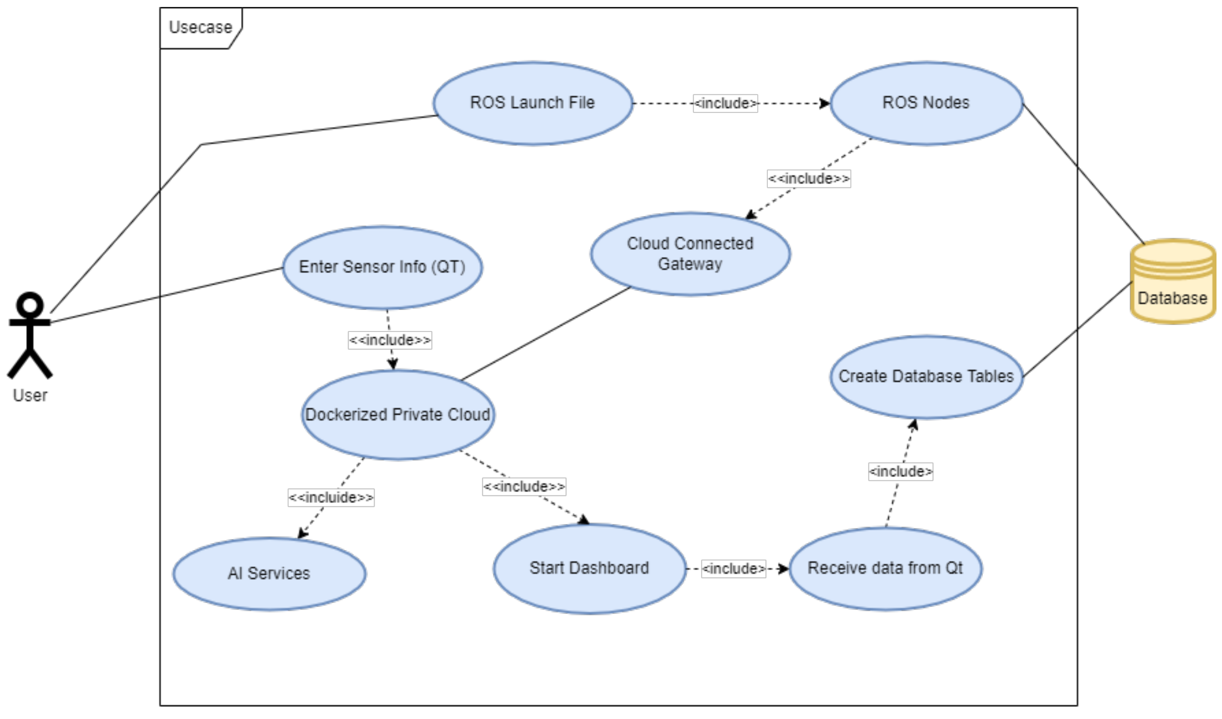


Figure 5.8: Use Case Diagram

The web pages of the dashboard are also represented as 3 different components the user can navigate them.Real-Time page which displays the sensed data, Actuate page where the user can send instructions to the vehicle to manipulate vehicle parameters and finally the Render Historical Page which retrieves sensed data from the database upon the user's specified dates.

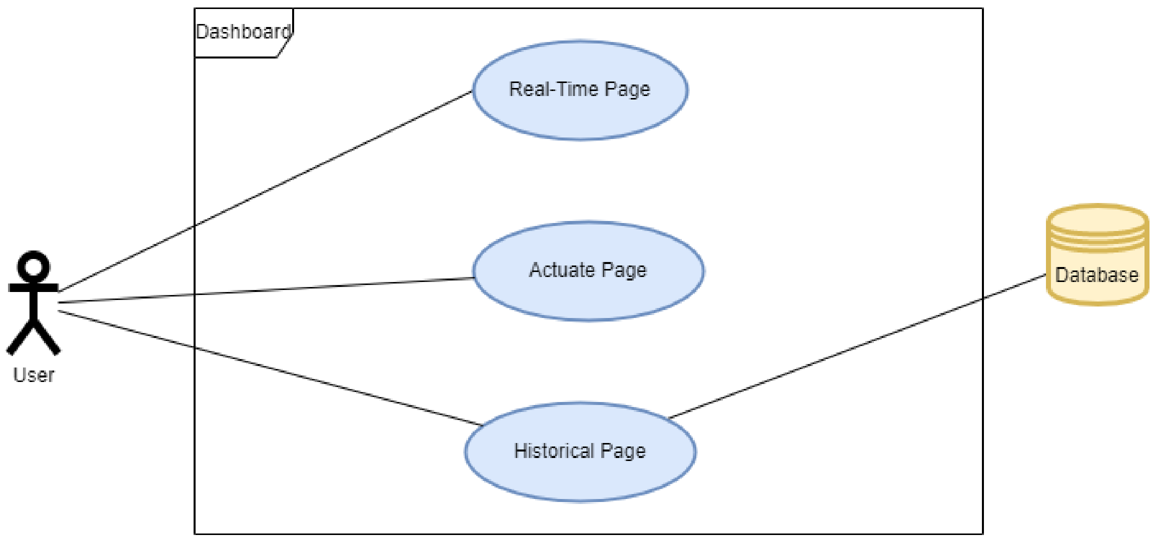


Figure 5.9: Use Case Diagram 2

5.5.2 Sequence Diagrams

In order to visualize the different modules of the project, we constructed the following sequence diagrams. There are mainly 3 modules to be discussed:

- Initializing configurations Diagram
- Actuate data module, including the live stream
- Historical data module
- Real-time module.

Initializing Configurations Diagram

This diagram represents how the dashboard is initialized. The user enters the sensors configurations through the Qt which then sends this data to the dashboard, which finally creates the database tables based on these sensors. [1]

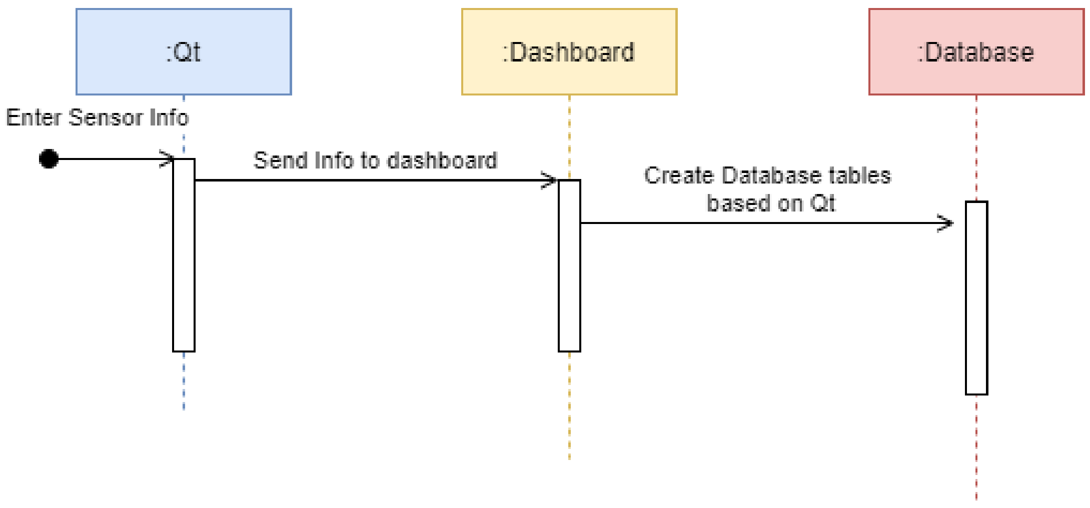


Figure 5.10: Initializing Configurations

Actuate Data Module

This diagram represents the actuate web page where the user can manipulate the vehicle parameters; speed, direction and steering angle. The sequence is as follows: the user requests the changes from the front-end which are directly sent to the back-end using an HTTP request. This request is subsequently sent through Socketio to the vehicle using ROS which handles the change requests and sends them to the corresponding topics. These changes are immediately displayed on both actuate and real-time web pages. [1]

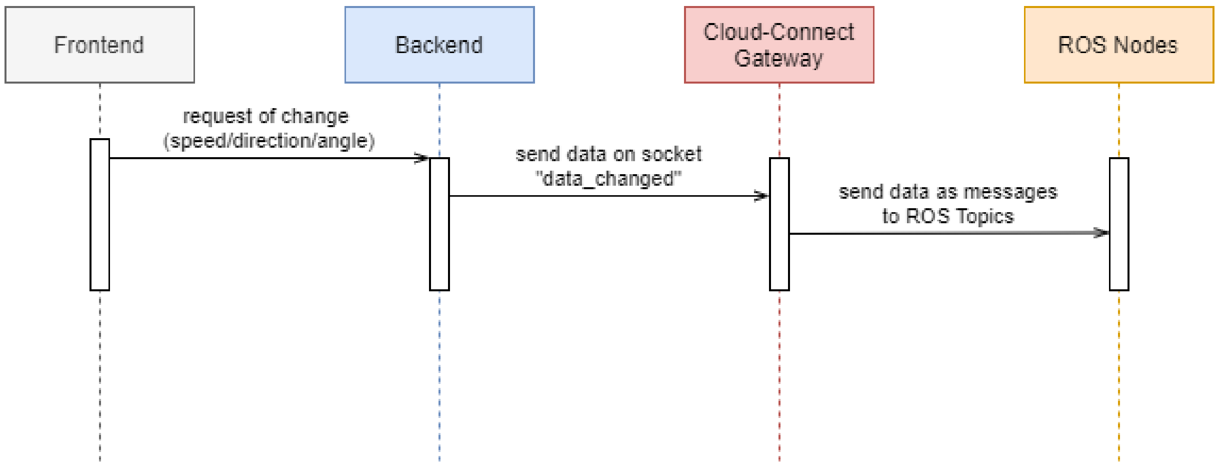

Frontend
Backend
Cloud-Connect Gateway
ROS Nodes
request of change (speed/direction/angle)
send data on socket "data_changed"
send data as messages to ROS Topics

Figure 5.11: Actuate Sequence Diagram

Live-stream

On the same web page, the camera live-stream is displayed, as shown in the following diagram. The front-end simply keeps listening on the server2web socket until any data is sent. The same scenario goes for the back-end which keeps listening on the cv2server socket until receiving any data is sent from ROS. Once the camera detects any input, the data is sent from ROS on the cv2server socket and then immediately sent from the back-end on the server2web to finally be displayed on the front-end. [1]

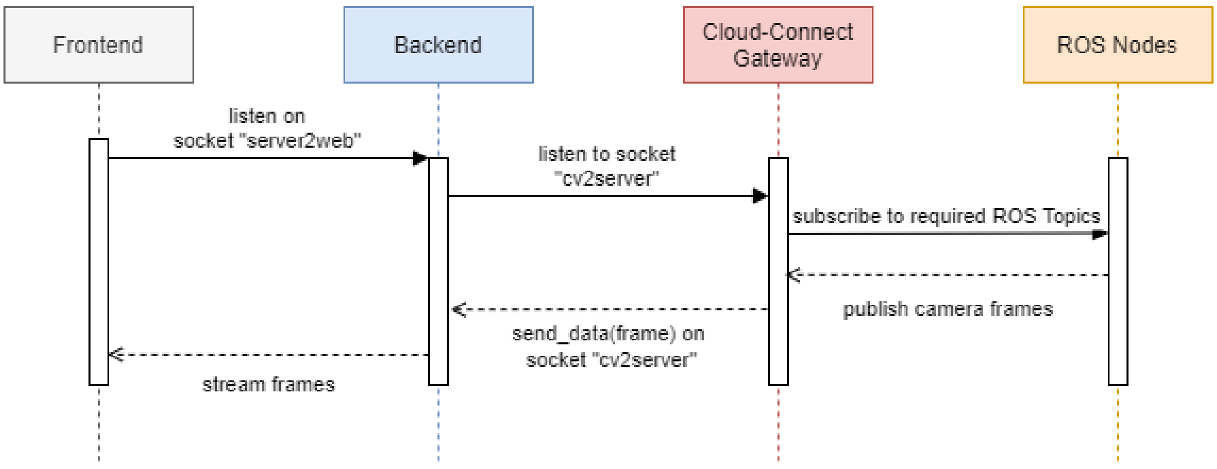

Frontend
Backend
Cloud-Connect Gateway
ROS Nodes
listen on socket "server2web"
listen to socket "cv2server"
subscribe to required ROS Topics
publish camera frames
send_data(frame) on socket "cv2server"
stream frames

Figure 5.12: Livestream Sequence Diagram

Real-Time Data

The real-time web page is where all the sensed data is visualized and displayed. The logic behind the page is simply based on opening a socketio (sensedData) as a port to receive real-time data from ROS. The front-end/back-end keeps listening on the socket until data is sensed and sent from ROS on the socket, it's then directly displayed by the front-end. [1]

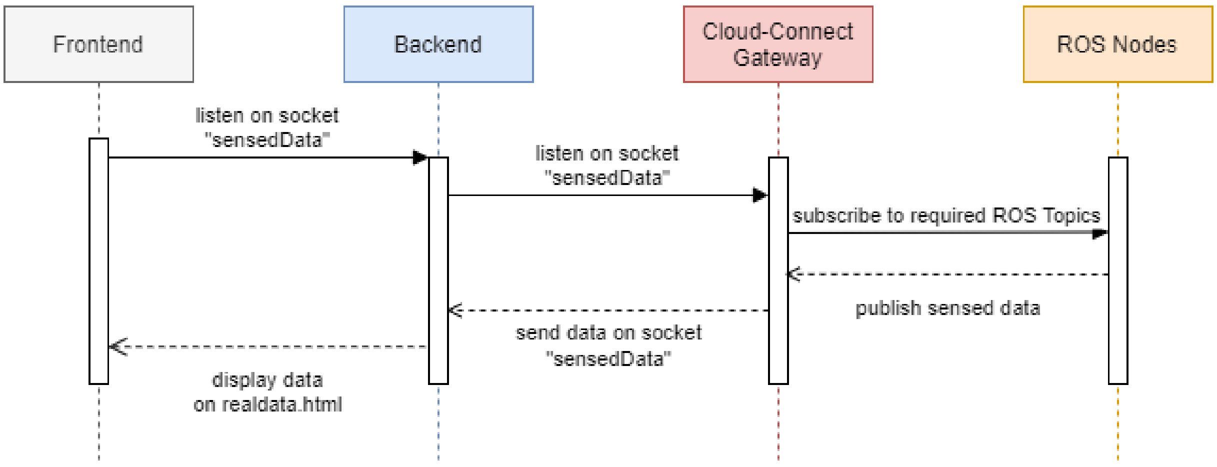

Frontend
Backend
Cloud-Connect Gateway
ROS Nodes
listen on socket "sensedData"
listen on socket "sensedData"
subscribe to required ROS Topics
publish sensed data
send data on socket "sensedData"
display data on realdata.html

Figure 5.13: Realtime Sequence Diagram

Historical Data

The historical web page is where the user requests to see old data by specifying the start and end dates required to be fetched from the database. The sequence goes as follows: the user enters the start and end dates which are processed as HTTP POST request to be sent to the back-end which translates the request to SQL queries directly sent to the database. The fetched data is returned to the back-end and finally sent back to the front-end to be displayed. [1]

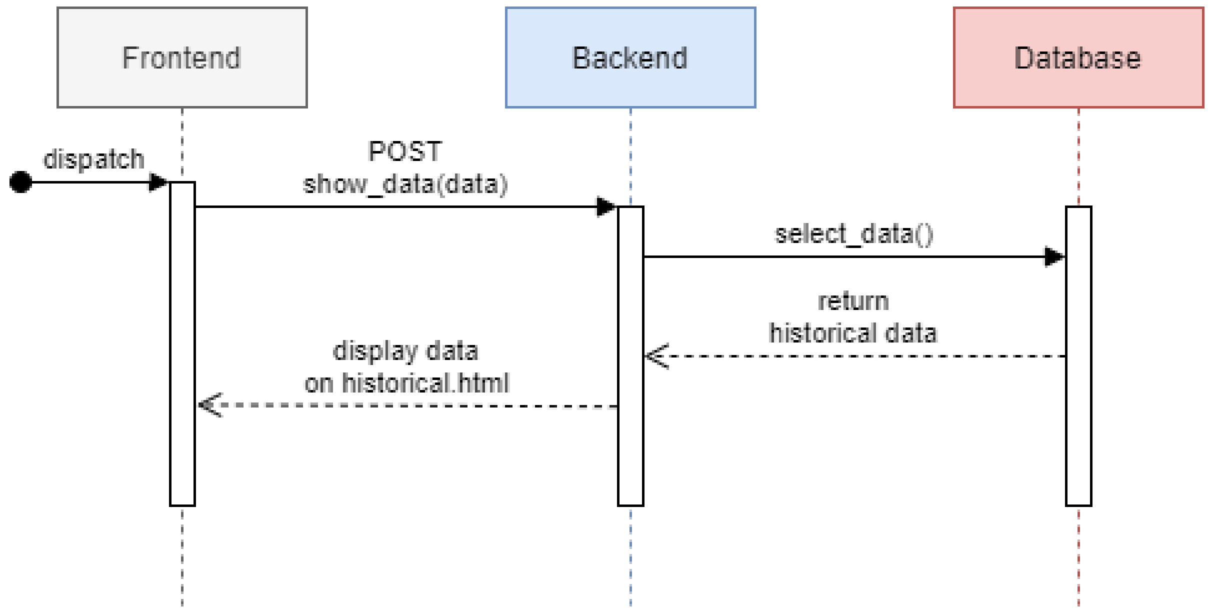


Figure 5.14: Historical Sequence Diagram

6 Software Requirements

Software Requirements Specification is always an important milestone to start any project. It establishes what the application should do, the constraints applied on it and all features that should be included.

Software Requirements are divided into two main partitions; Functional and Nonfunctional Requirements.

6.1 Functional Requirements

Functional Requirements represent the requirements which the end user specifies and requests to have in the system. They are represented in the form of the input that should be given to the system, the function operated and the output expected.

6.1.1 Functional Requirements

What should be included

Functional requirements should cover the following points:

- Functions conducted in every screen.
- How data input is handled.
- It should describe the expected output.
- The workflows performed by the system.
- It should clearly define access control over the data in the system.

Benefits of Functional Requirements

The advantages of writing functional requirements include:

- Providing a way to check whether the application is covering all the functionalities that were listed in the system's functional requirement of that application.
- A functional requirement document defines the functionality of a system or one of its subsystems.
- Errors caught in the Functional requirement gathering stage are of low cost.

System's Functional Requirements

- The system should allow the user to select the number and type of the sensors.
- The system should create the database tables dynamically depending on the sensors entered.
- The system should display the real-time sensed data charts upon user request.
- The system should display the historical sensed data charts upon user request and depending on the start and end dates chosen.
- The system should give the user the option to change the vehicle's parameters upon user request to actuate.
- The system should handle any missing data required by the user.
- The system should allow communication between the dashboard and database if there's any required data.
- The system should allow communication between the dashboard and ROS (the vehicle) through sockets.
- The dashboard should handle and be able to display real-time streaming coming from the vehicle through sockets.
- The dashboard should use the Google Maps API to display the location of the vehicle.
- The dashboard and the cloud-connected infrastructure should be generic enough to handle any ROS system connected by the user.
- The system should provide the user with all access controls since the application is aimed to help the user control his vehicle over the cloud.

We should note that there are no specific requirements or constraints on the ROS system to be connected, since it depends only on the case study chosen by the user.

6.2 Non-functional Requirements

While the functional requirements specify *what* the system should do, the non-functional requirements specify *how* the system performs its functionalities. Non-functional Requirements represent the constraints applied on the system which as well should be met for the project to be accepted. They cover alot of aspects such as: Portability, Maintainability, Reliability, Performance, Reusability, and Security.

6.2.1 System's Non-functional Requirements

What should be included

Non-functional requirements should describe how the system handles quality constraints, including but not limited to:

- Security and privacy
- Capacity and storage
- Compatibility; where the hardware components and the supported versions of operating systems are specified.
- Availability of the system.
- Usability; describing the user experience of the system.

Benefits of Non-functional Requirements

The advantages of writing non-functional requirements include:

- Specifying the quality attributes of the software.
- Ensuring the reliability, availability, performance, and scalability of the software system.
- Participating in constructing the security policy of the software system.
- Ensuring good user experience, ease of operating the software, and minimize the cost.

System's Non-Functional Requirements

- The system should be secure enough allowing only the user to have control over his application.
- The system should be able to perform real-time, without any noticeable delays.
- The storage should depend on the user's application size and running time.
- The system should be working on an Ubuntu 18.04 operating system supporting Docker and ROS-1.
- The system should be able to work with different boards, not limited to Jetson board.
- The system should be able to keep running until the user decides to shut it down.
- The user should have an easy experience selecting the sensors and their configurations from the QT user interface.
- The user should have an easy experience dealing with the dashboard; displaying sensed data, actuating data and navigating through different screens.
- The system should keep running normally without crashing in case of any unexpected actions by the user.

7 Implementation

In this section, we are going to mention the implementation we have reached for each component of the system. As mentioned, the system is composed of three main components, the configurator which is the application the user starts to interact with to customize the dashboard according to their ROS subsystem, we are going to highlight how a friendly user interface is important for efficiency and how it's implemented. The second component is our dashboard, the old version of the dashboard was a static application that couldn't be changed according to different ROS subsystems, in this section we are going to describe how it can be customized and configured with different services and sensors charts with the information the user submits through the qt configurator. Lastly, we are going to explain the cloud-connected nodes that is our main component to connect between the ROS subsystem and the dashboard.

7.1 Configurator

7.1.1 Initial Implementation of UI

As mentioned in QT section, this was the first implementation for the configurator. It was implemented using PyQt5 plain python code. We found it to be less user-friendly and we needed to add more feel and like to the UI.

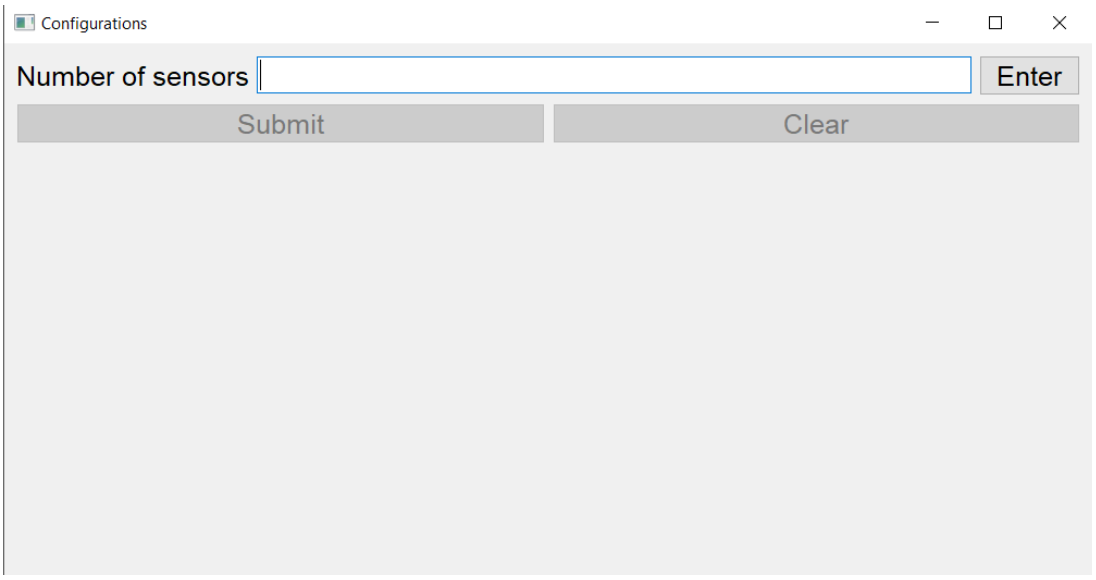
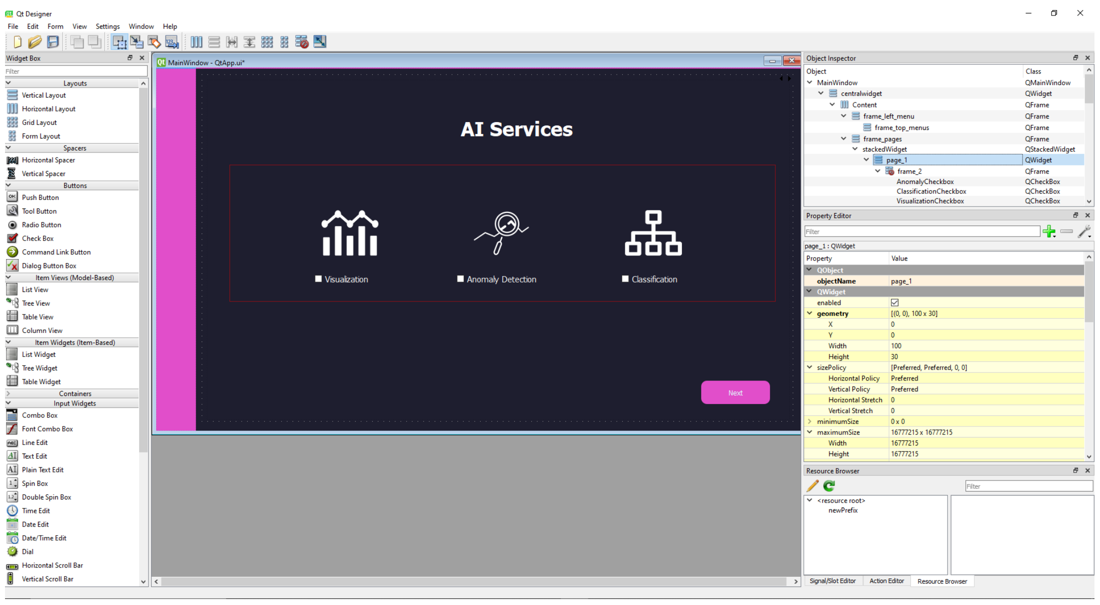


Figure 7.1: Initial UI

As an enhancement, we wanted a more consistent UI with the Dashboard theme. Therefore, we needed an assisting toolkit to help in developing and improves productivity.

7.1.2 Qt Designer

Qt Designer is a Qt tool used in building GUIs using Qt Widgets that are ready-to-use. It is based on the what-you-see-is-what-you-get concept and is based on dragging and dropping of widgets according to the need. It supports different styles and resolutions. Widgets are then integrated with the programmed code that can assign behavior to graphical elements.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
Content	QFrame
frame_left_menu	QFrame
frame_top_menus	QFrame
frame_pages	QFrame
stackedWidget	QStackedWidget
page_1	QWidget
frame_2	QFrame
AnomalyCheckbox	QCheckBox
ClassificationCheckbox	QCheckBox
VisualizationCheckbox	QCheckBox

page_1 : QWidget

Property	Value
QObject	
objectName	page_1
QWidget	
enabled	☑
geometry	[(0, 0), 100 x 30]
X	0
Y	0
Width	100
Height	30
sizePolicy	[Preferred, Preferred, 0, 0]
Horizontal Policy	Preferred
Vertical Policy	Preferred
Horizontal Stretch	0
Vertical Stretch	0
minimumSize	0 x 0
maximumSize	16777215 x 16777215
Width	16777215
Height	16777215

<resource root>
newPrefix

Figure 7.2: Configurator using Qt Designer

At Design completion, a .ui extension file is saved that can be converted to a python-programmed code file -.py extension file- using the command mentioned below.

```
pyuic5 -x QtApp.ui -o QtGUI.py
```

Scene Created

The first scene introduces the AI Services supported by our Dashboard which are Anomaly Detection, and Classification. Note that Visualization is added as a default feature. And on pressing Next, the second scene is displayed.

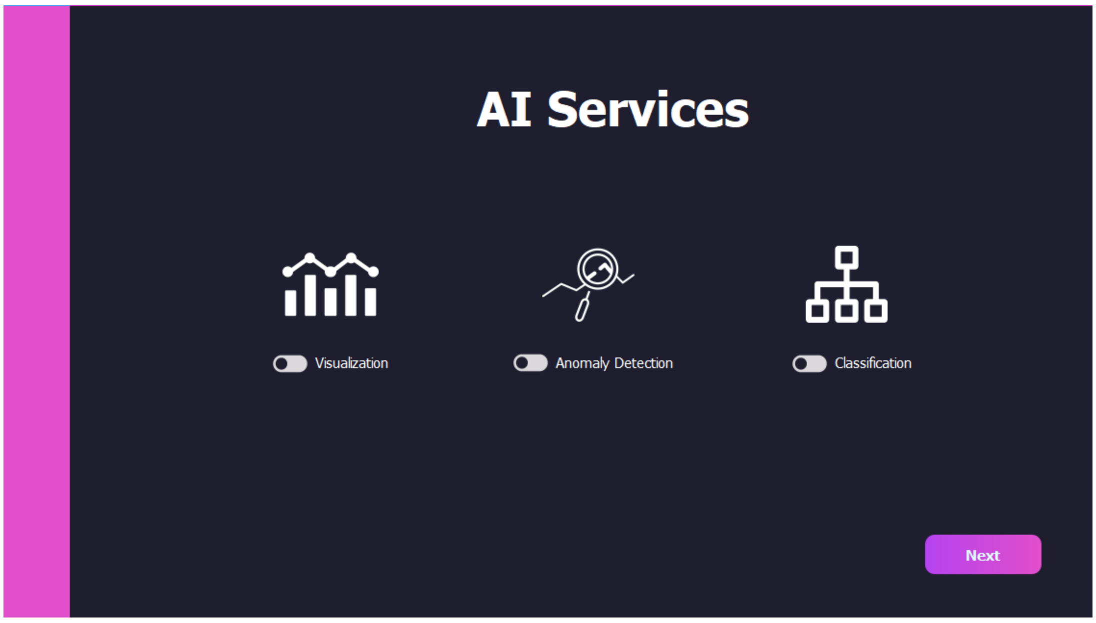


Figure 7.3: AI Services Scene

The second scene represents the sensors selection phase as well as the services required to be displayed on our dashboard on creation. The services offered are Actuate, RGB Camera and GPS.

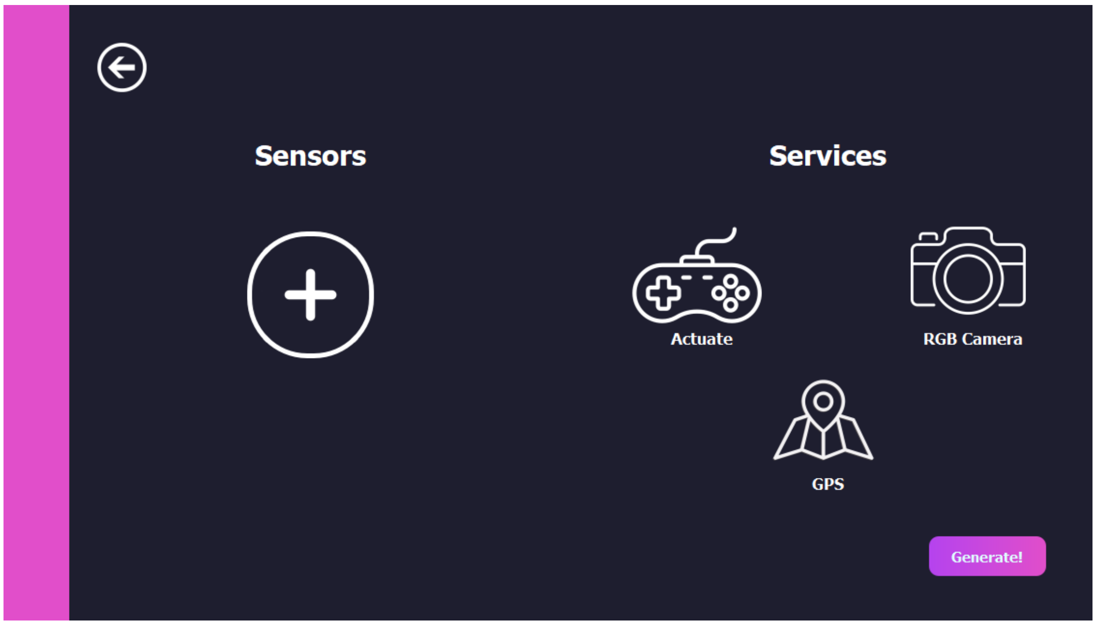


Figure 7.4: Services Scene Overview

On pressing the plus button, the sensors selection option is displayed for the user to choose according to their ROS subsystem configuration.

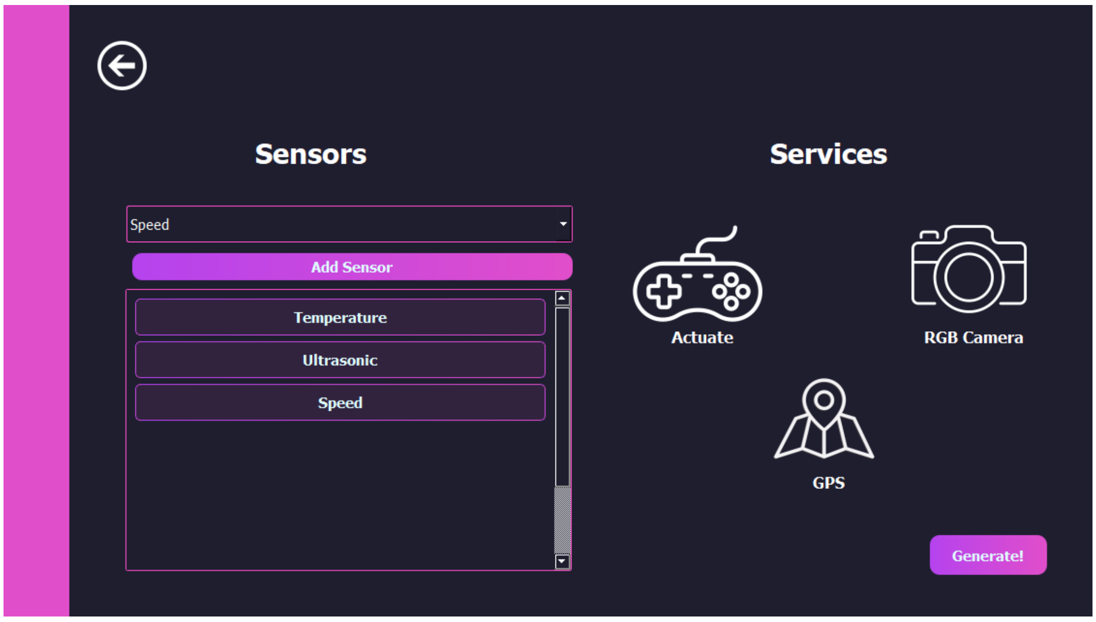


Figure 7.5: Sensors Option and Services Scene

User Interaction with the Configurator

Pointing to the first scene above, AI Services, the user selects their preferred AI Services to be supported on the dashboard.

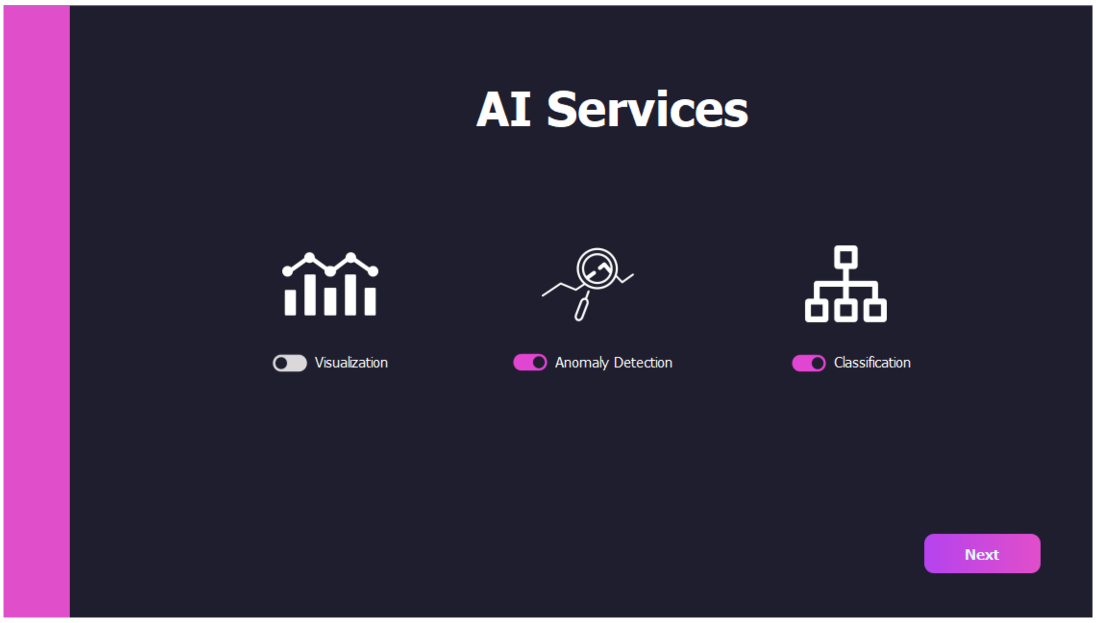


Figure 7.6: Toggled AI Services

In the Sensors and Services Scene, the user gets to select the sensors he requires from the combo box, and if the sensor is not present then the user can actually adds it using the Others option where he can type its name.

The user can also delete a sensor from the displayed list that he no longer requires by pressing on it.

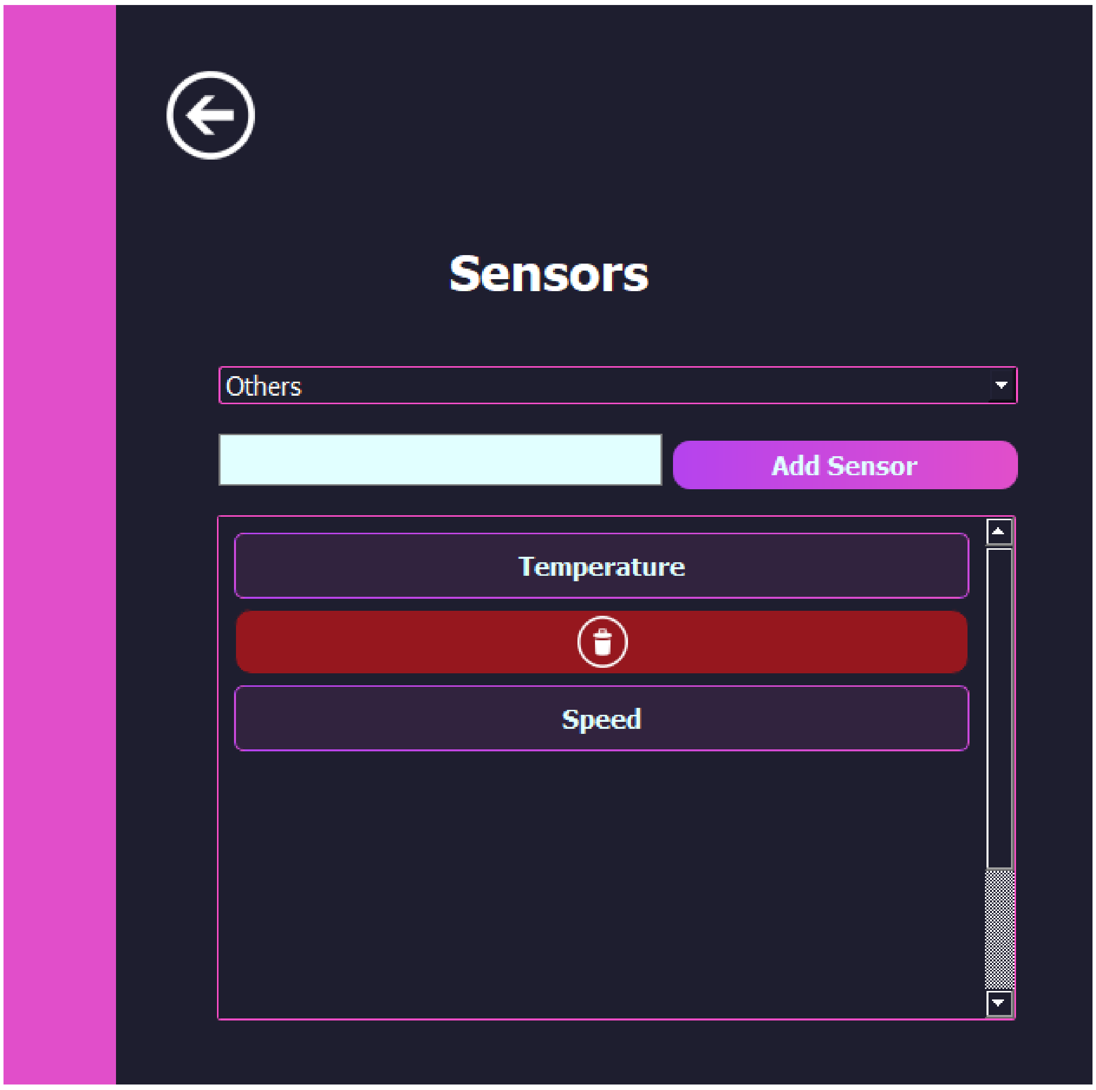


Figure 7.7: Addition and Deletion of Sensors

The Services part allows the user to select one or more service by pressing on the icon.

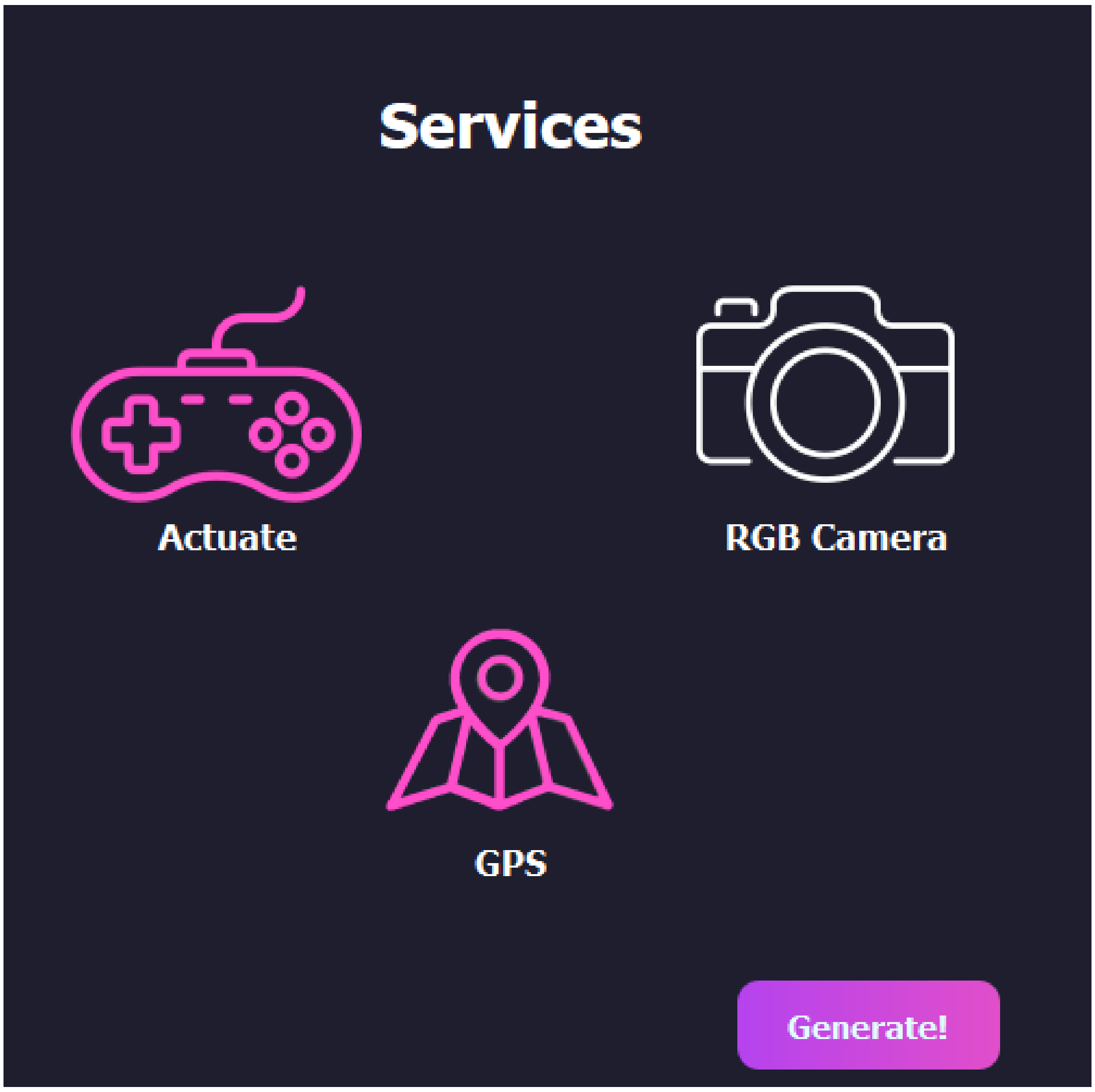


Figure 7.8: Selection of Services

7.1.3 Automation

After the user's configuration selection, the user presses "Generate Button" which runs the Docker Images found in our container. Docker container gets started and Images starts running when the configurator trigger the following command:

```
os.system("docker-compose -f " + dir_path + " up -d")
```

"dir_path" refers to the static directory location where the "docker-compose.yml" file exists.

The configurator also runs the following commands to run the cloud-connect scripts in the background. These scripts are located with the user's ROS directory

```
os.system("nohup python3" + reactive_path + "> output.log &")
os.system("nohup python3" + stream_path + "output.log &")
os.system("nohup python3" + teleop_path + "output.log &")
```

"reactive_path" refers to the "reactive.py" file that is responsible for connecting the dashboard and ROS sockets to send the sensors information. "stream_path" refers to the "stream.py" file that connects that streaming sockets between the dashboard and ROS for the camera. Lastly the "teleop_path" refers to the "teleop.py" file that sends the actuate data between ROS and the dashboard

The Configurator connects to the Dashboard using SocketIO through the namespace "dynamicDB". The data gets emitted to render the Dashboard as the user expects. And finally the browser automatically launches the Dashboard on:

```
http://localhost:8000
```

7.2 Dashboard

7.2.1 Frontend

The Dashboard's Frontend is mainly divided into 3 web pages: Historical Data and Realtime Data under the Sense header, and Actuate Data under the Actuate header.

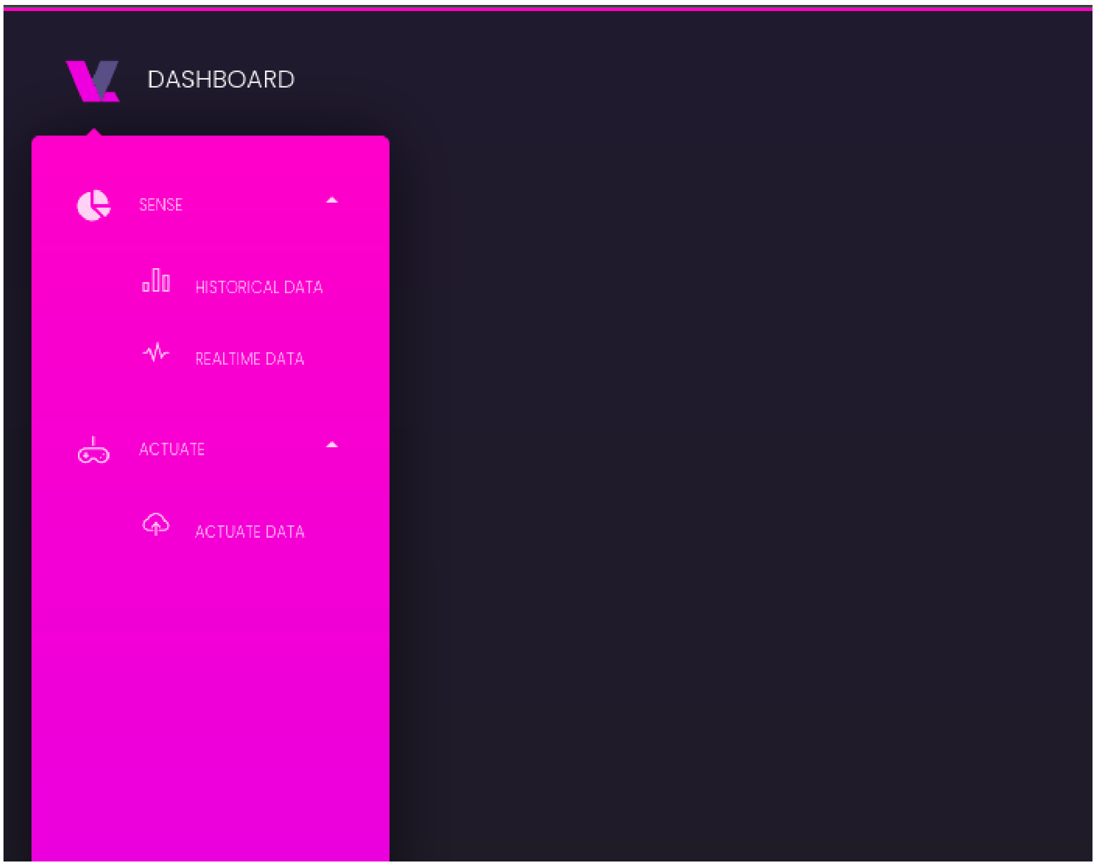
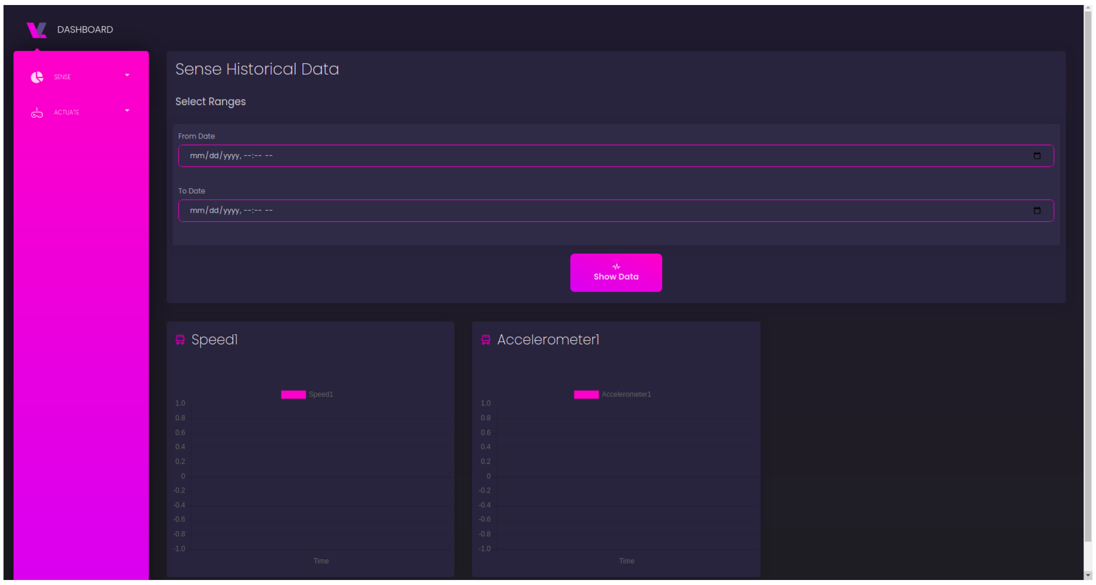


Figure 7.9: Dashboard

Historical Data

After receiving the sensors names from the QT configurator, the historical page displays the sensors graphs accordingly. In the figure below, the user chose to enter two sensors, Speed and Accolerometer, hence these are the ones displayed.

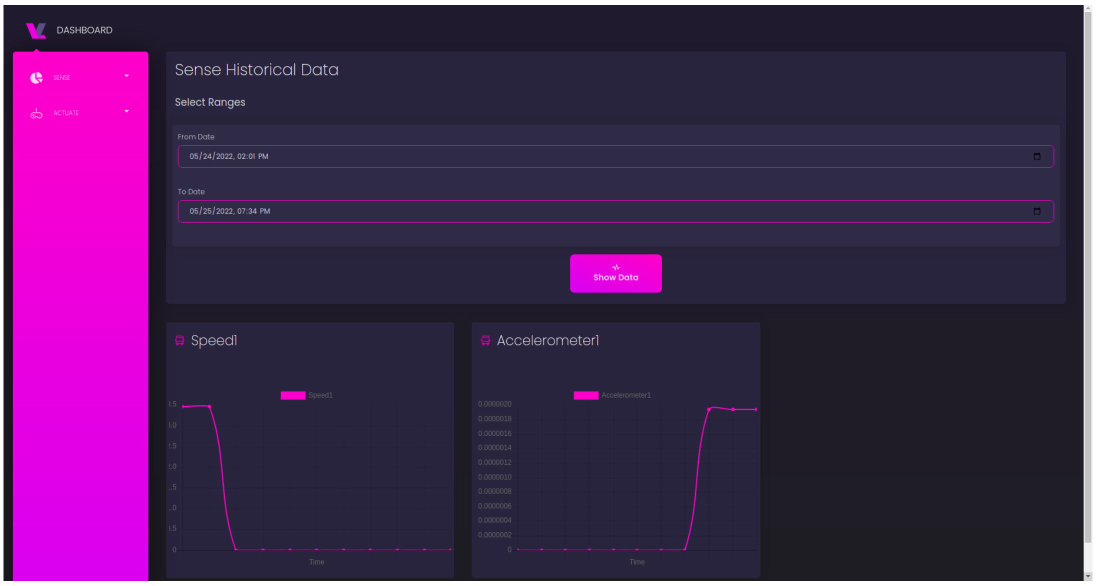


From Date: mm/dd/yyyy, --:-- --

To Date: mm/dd/yyyy, --:-- --

Figure 7.10: Historical Data Page

This page mainly aims to provide the user the functionality to enter two dates (from and to dates) that correspond to the data he wants to display. This is done by providing the user with these two enteries and shown and when he clicks show data, the needed data is fetched from the database to be finally displayed to the user, as presented below.



From Date: 05/24/2022, 02:01 PM

To Date: 05/25/2022, 07:34 PM

Figure 7.11: Historical Data Page With Displayed Data

This is done, as previously discussed, by sending these dates and a dictionary to the backend using AJAX, and after some processing, the backend fetches this data by accessing the database and returns it to the fronted.

Realtime Data

After receiving configurations from QT that is entered by user, The Realtime page starts showing graphs which display live readings from the sensors. In the figure below, the user choose to enter two sensors; Speed and Accolerometer, hence these are the ones displayed.

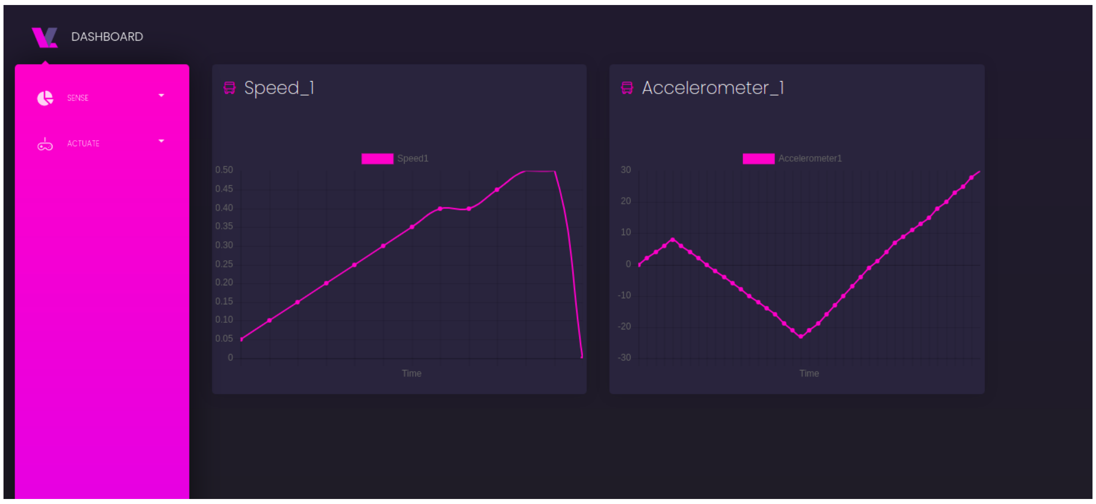


Figure 7.12: Realtime Data Page With Displayed Data

The readings are displayed to user by listening to 'sensedata' socket which receives data sent from the Accolerometer and Speed sensors in the ROS susystem.

Another feature that exists on this page, depending on the user's configuration, is the anomaly detection algorithm, which will be discussed in details in the following sections.

7.2.2 Actuate Data

The Actuate Data web page aims to provide the user control and monitoring over the robot, based on what he entered in the QT configuartor. There are 3 main Services that he can choose from, as previously explained: Actuate, GPS, RGB Camera.

- Actuate means he gets to move the robot through the controls provided on the pages.
- RGB camera means he gets to visualize what the robot's camera is catching at the moment.
- GPS helps him in tracking the robot's position at the time.

In our case study, we're mainly working with the actuate and RGB Camera services, which will be displayed on the web page as shown below.

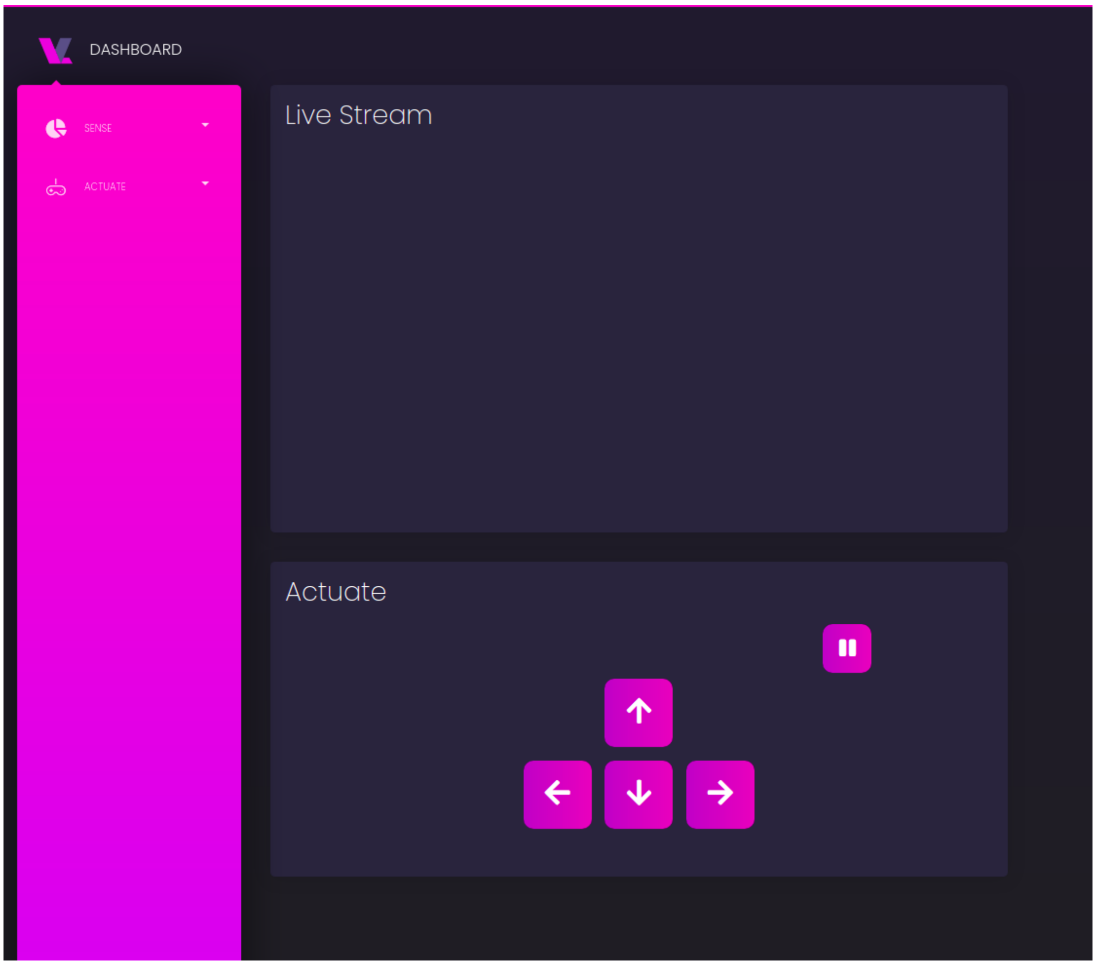


Figure 7.13: Actuate Data Page

Each of the two chosen services has its own flow of data. Starting with the Live Stream or RGB Camera, the Actuate page listens on a socket 'cv2server' waiting for images to come from the cloudConnect file. Once images are received they are directly displayed on the screen. A sample of what the camera displays can be shown in the figure below.

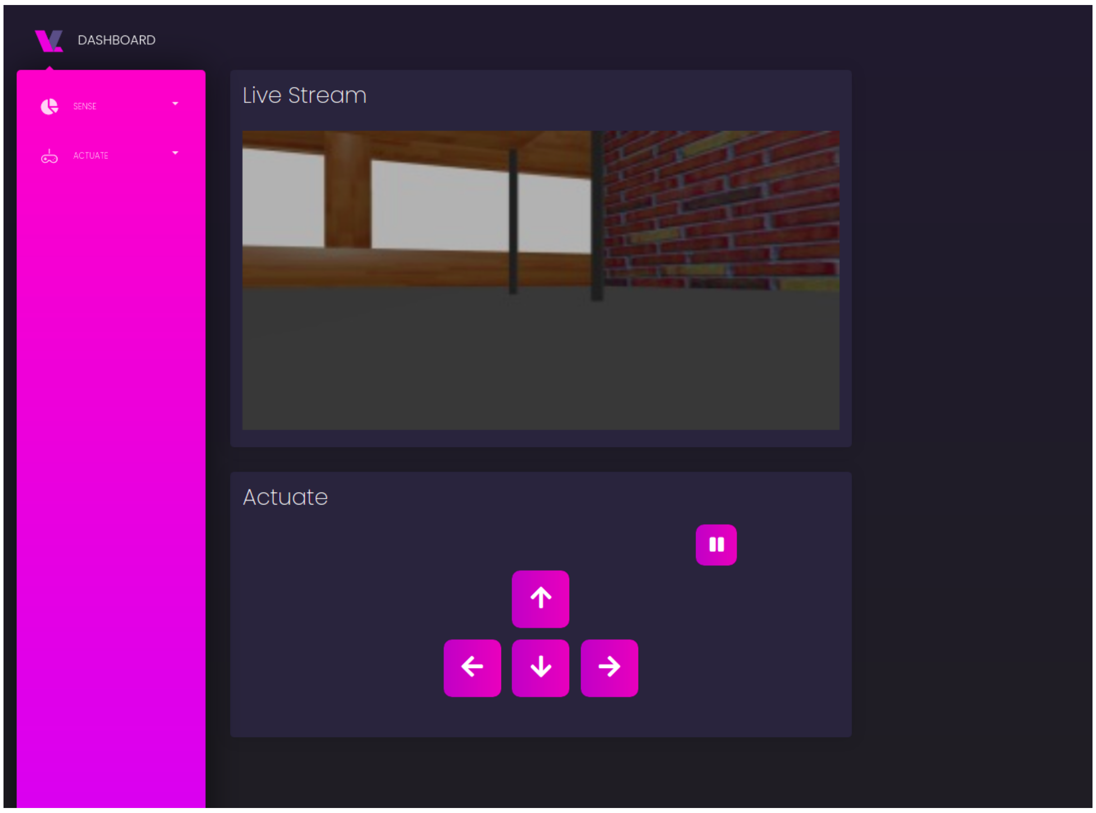


Figure 7.14: Actuate Data Page With Camera Images

The Actuate flow works the other way around, meaning that when the page senses that the buttons are being pressed, it sends corresponding signals to ROS, through a socket.

Another feature that exists on this page, is the classification/object detection algorithm, which will be discussed in details in the following sections.

7.2.3 AI Services

The dashboard currently supports two kinds of AI services, Anomaly detection and Object classification. The user can configure the dashboard as they need for their ROS subsystem. They can choose to turn on or off the service from the configurator to make the dashboard functional according to their system.

Anomaly Detection

Detecting outlier points is one of the AI services that is implemented in our Dashboard. Normally, humans can easily detect unnatural behaviors in a set of visualized data, automating this process is the use of anomaly detection algorithms. Since anomaly detection is the process of finding out the outliers in any system it would require some domain knowledge to choose the right values in the data-set; but with constantly growing data-sets it becomes harder for humans to detect and analyze these anomalies so the algorithms are useful for faster and more accurate detection. Experts in domain usually set their own thresholds on the data but with AI algorithms, thresholds can be easily set and changed according to the available data without the need of human intervention setting the values.

AI-driven algorithms are self-learning systems that learn from the current and old data patterns while predicting with high accuracy and correlation to the domain values. In addition, automating the process of finding out anomalies eases the identification of the smallest anomalies that may go unnoticed by humans. The detection is a real time analysis solution and interprets data activity once the data is received.

Interquartile Range Method

The IQR is used as a statistical way to display the non-Gaussian distribution sample of data. IQR is calculated by getting the difference between the 75th and the 25th percentiles, this represents 50% of the data and is considered the body of the data.

It is used to detect the outliers by defining the upper and lower limits of the sample values, these limits are calculated by multiplying the IQR by a factor k, its value is normally 1.5. This would represent the cut-off value above the 75th percentile (upper limit) or below the 25th percentile (lower value). Values outside these limits are stated to be outliers.

In our implementation, we used the Interquartile Range Method to detect the outlier values in the data coming from the Robot Operating System while moving that is by using the built in NumPy percentile() function to calculate the IQR and the cut-off limits.

Data Flow

Each sensor data emitted from ROS is displayed on the dashboard in a labeled chart for that sensor. If the user requires to detect the outliers in the data received on the dashboard, presuming they already clicked on the anomaly detection service in the configurator in their initial steps, they click on the "Detect Outliers" button. The data is parsed into a dictionary with each chart and its' values are sent to the backend through AJAX POST request.

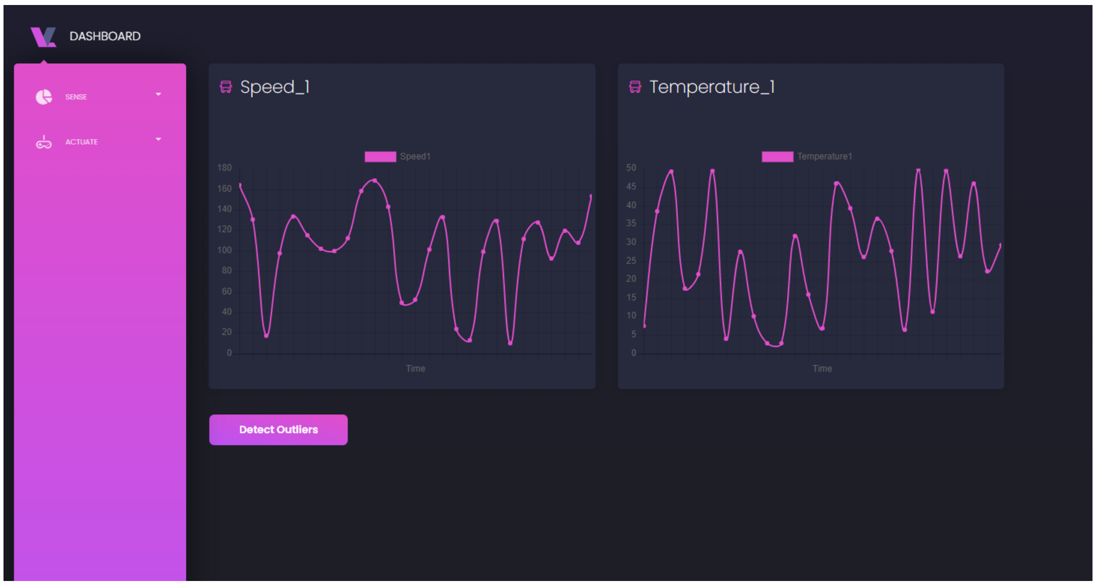


Figure 7.15: Graph Points

In the backend, we loop through each graph to get its data and run the anomaly detection on it, the algorithm can be changed in the future for enhancement but for now we are using the Inter Quartile Range method by multiplying the cutoff by a certain factor. The following diagram shows the navigation of data between the frontend and backend for the dashboard service

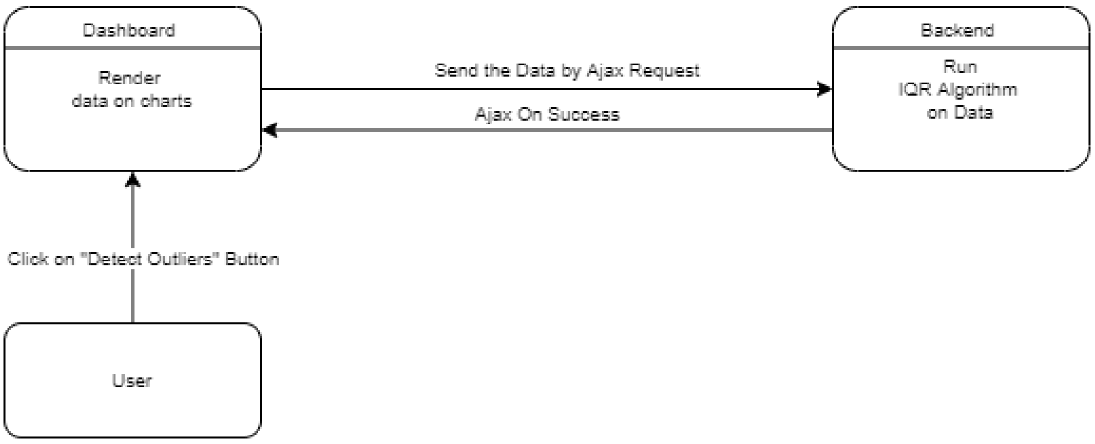

Dashboard
Render data on charts
Send the Data by Ajax Request
Ajax On Success
Backend
Run IQR Algorithm on Data
Click on "Detect Outliers" Button
User

Figure 7.16: Data Flow Diagram

Using AJAX on success callback function, the backend returns a dictionary of the indices of outliers and their values for each chart. Json object is parsed again to be displayed on the charts, the points which are anomalies are highlighted with green to stand out between the other normal points on the chart

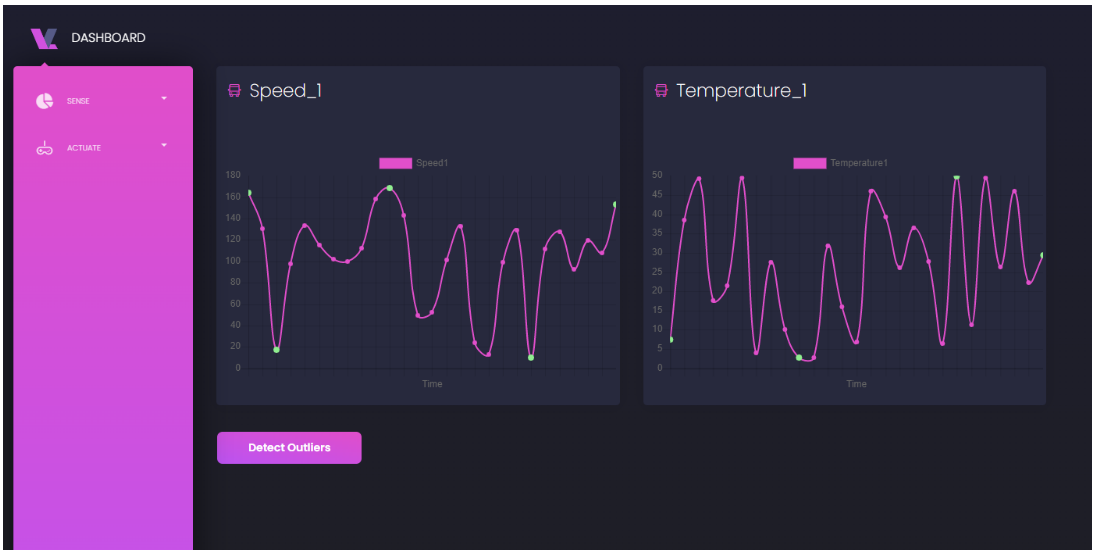


Figure 7.17: Outliers Highlighted

Classification

The second AI service offered by our system is Object Classification. Image classification is a supervised learning problem that involves predicting the class of one object in an image. By defining a set of target classes (objects to identify in images), and training a model to recognize them using labeled example photos(test data). The

The model used in our system is a readily trained model based on the COCO (Common Objects in Context) dataset, which is a large-scale object detection, segmentation, and captioning dataset. The model used is the COCO-SSD model embedded in the javascript code of the Dashboard.

Finally, for a better visualization, We surround our detection by a colorful rectangle to indicate it's type using the model specified.

We can find below some of the objects detected by this model.

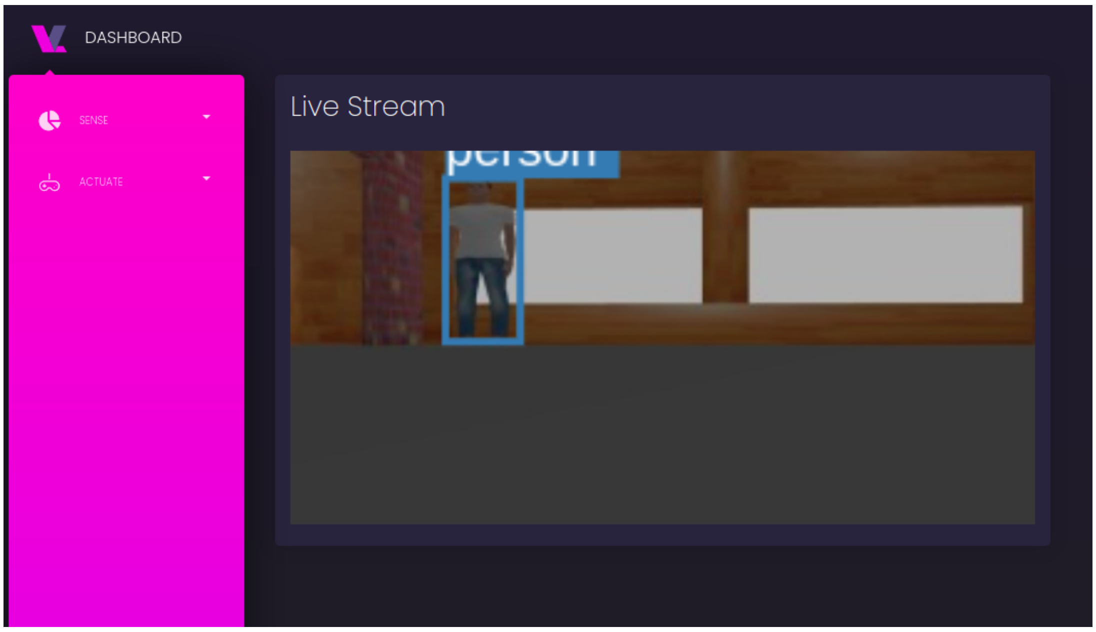


Figure 7.18: Person Detected in the Environment

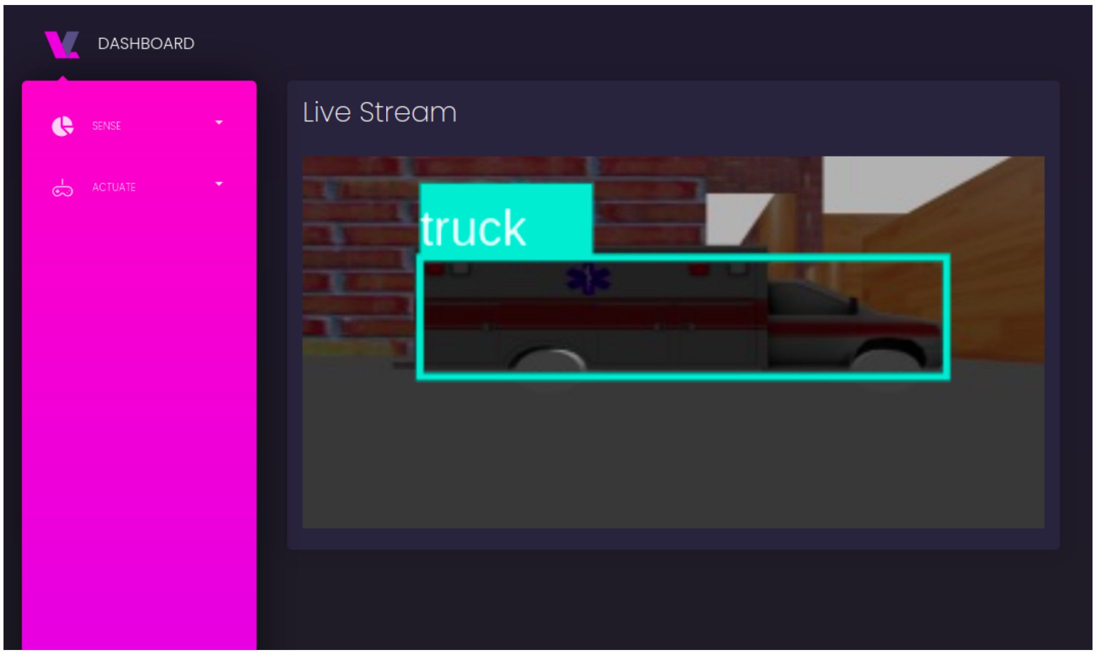


Figure 7.19: Person Detected in the Environment

7.3 Cloud-Connected Nodes

7.3.1 Overview

As we need the dashboard to communicate with ROS, we need to establish a way for this connection to happen, we implemented something like a gateway that ease the communication between them. The gateway named cloud-connected nodes which are basically three nodes that are responsible for the communication between the dashboard and any ROS subsystem.
The cloud-connected nodes will be executed once the user submits from the QT configurator generator. Thus, the nodes will be operating in the background through the lifetime of the program in order to be ready to operate correctly.
In the upcoming sections, we are going to explain more about these three nodes and how they work.

7.3.2 Cloud-Connect Diagram

In the above figure, we could see how the Cloud-Connect nodes work. The dashboard communicate with the nodes through socket.io package, while the ROS subsystem communicates with them through ROS topics.
Next, we will talk about each connection in more details.

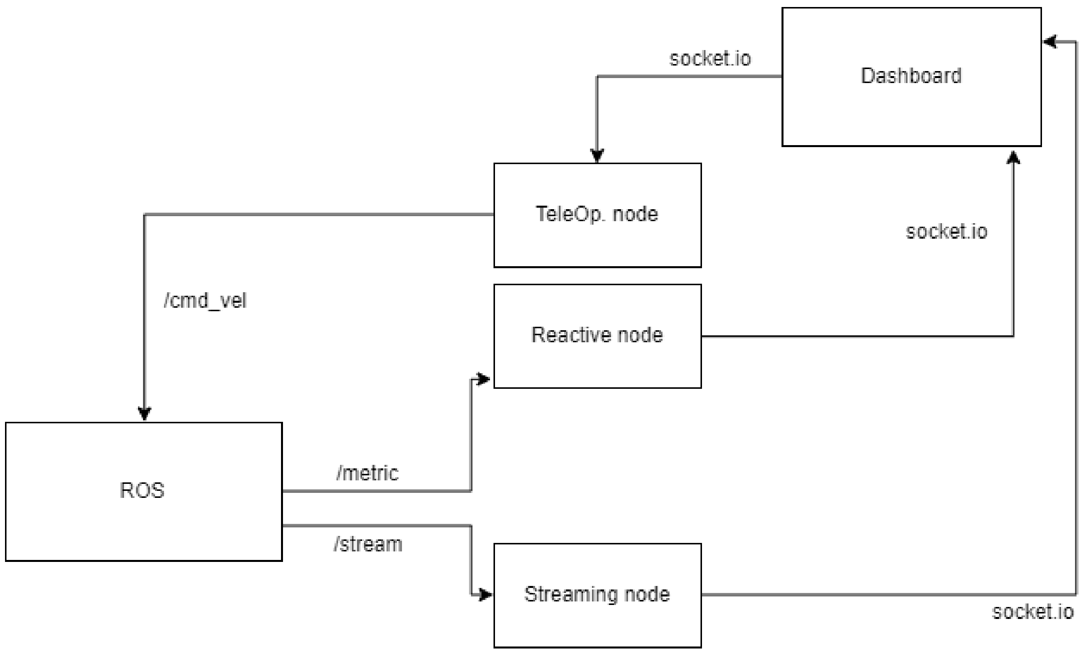


Figure 7.20: Cloud-Connect diagram

7.3.3 ROS Communication

As we said, ROS communicates with the CloudConnect nodes through ROS topics. There are three nodes, TeleOp., Reactive, and Streaming nodes, each node has its own functionality.

Reactive Node

The Reactive node works with ROS subsystem in the two directions where it sends to and receives from ROS. The communication is done through a topic named metric.
The main function of the node is that it collects the data needed from the sensors in the ROS subsystem, this is implemented by subscribing to the topic metric while ROS publishes to that topic the data of the sensors.

Streaming Node

This node is responsible for collecting the image data from the camera in ROS subsystem, this is implemented by subscribing to the topic stream while ROS publishes to the same topic the streaming of the camera. All the user has to do is just publish the camera images to this topic, and the node will handle anything else.

TeleOp. Node

TeleOp. node is handling the movement of the robot, where the user could controls the robot's movement easily, the node publishes the direction and the speed required from the user to topic control, on the other side, ROS subscribes to the same topic in order to get the data published

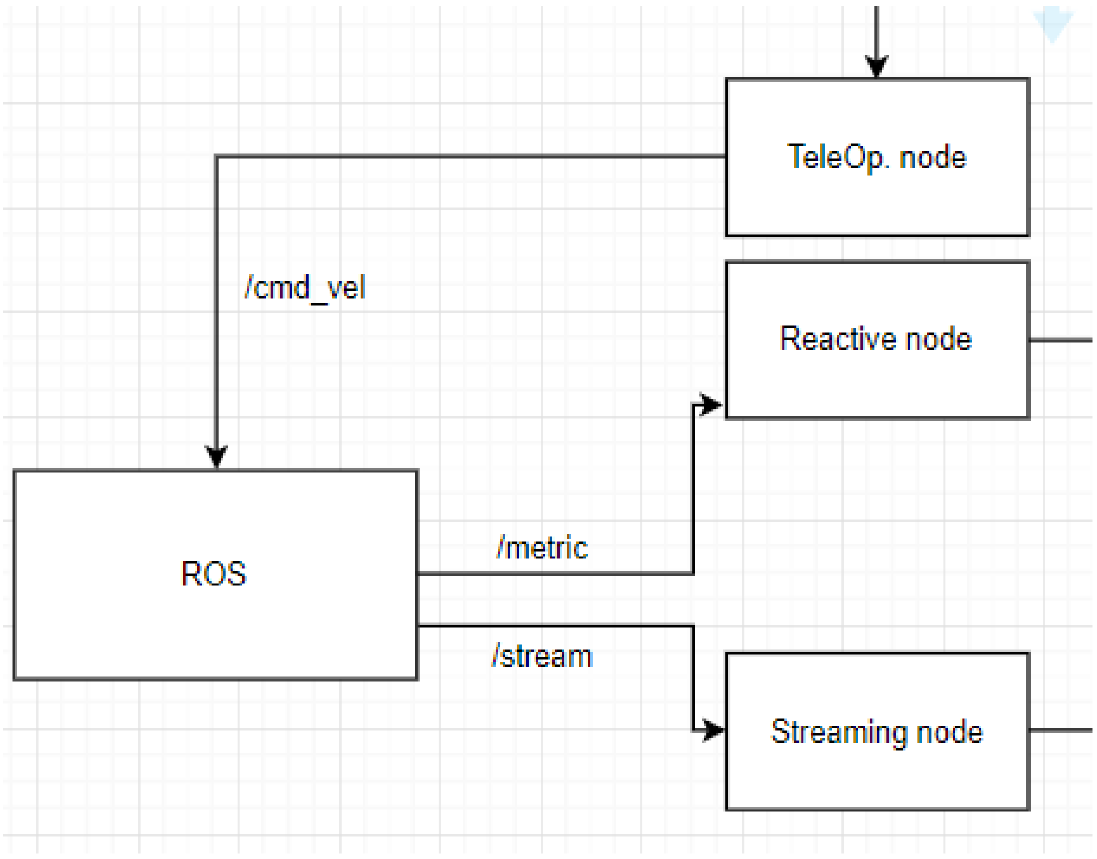


Figure 7.21: Dashboard and CloundConnect nodes

and use it. This topic should be the topic responsible for holding velocity's data in the ROS subsystem of the user, where the user is responsible for constructing that topic that holds the velocities.

7.3.4 Dashboard Communication

As mentioned above, the dashboard communicates with the CloudConnect nodes through socket . io. we will talk about the same three nodes we mentioned in the above section and how they work from the prespective of the dashboard; TeleOp., Reactive, and Streaming nodes.

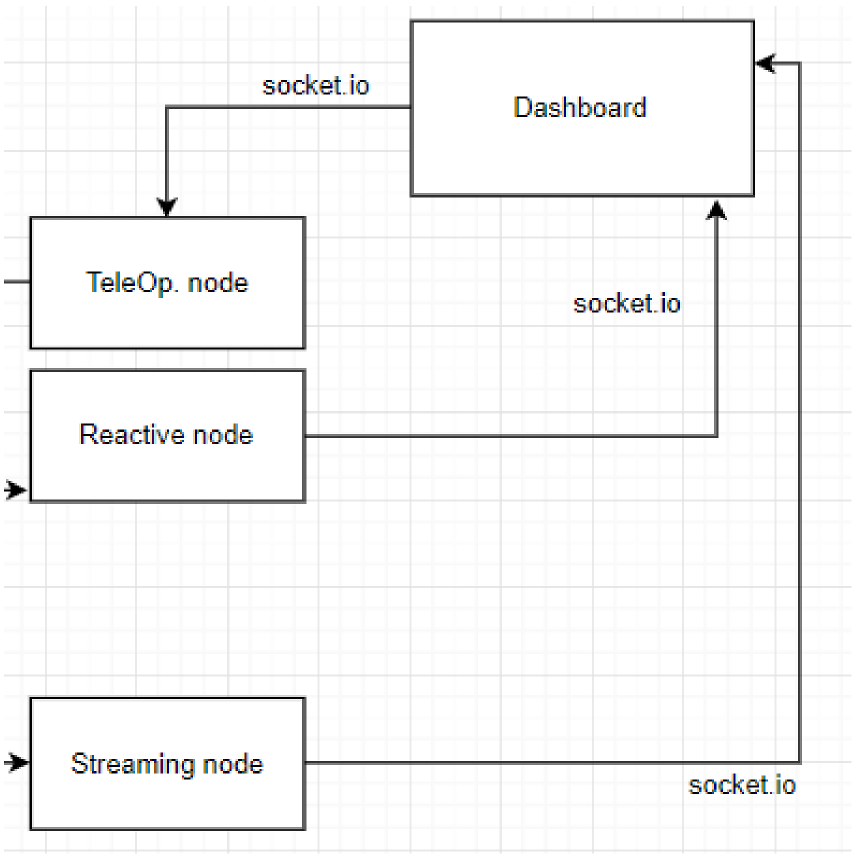


Figure 7.22: Dashboard and Cloud-Connect Nodes

TeleOp. Node

TeleOp. node works with the same functionality which is it sends the commands of controling the robot from the user. The communication between it and the dashboard works through socket.io where in the dashboard the user can control the robot using the keyboard or by clicking on the arrows itself appeared on the dashboard. These commands are sent to the node, then the node sends it to ROS.

Reactive Node

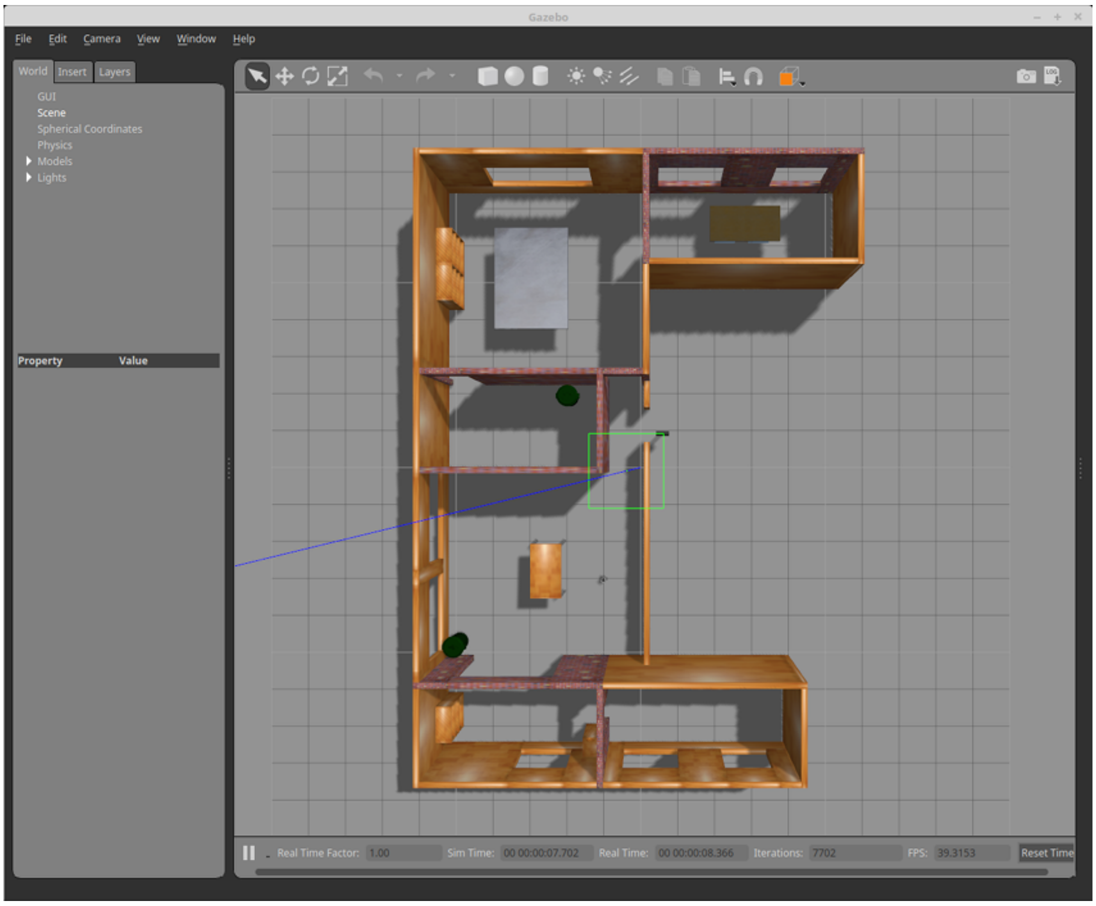
After receiving the sensors' data from ROS subsystem, the node sends these data to the dashboard through socket.io where the dashboard uses these data to display it to the user in the charts and also saves it in the database.

Streaming Node

Streaming node receives the images from ROS as mentioned above. After that, the node sends these images to the dashboard by socket.io so that the dashboard displays these images in the streaming page where the user could see the streaming of the camera in the robot and use it as tracking to what the robot sees.

7.4 ROS

Initially, the software requires ubuntu 18.04 as an operating system on your device due to the fact that we are using melodic ROS version. For the robot environment, you can install and use any environment you are intending to use therefore we are using turtlebot3 as our robot and turtlebot3 house as our environment as shown on the following picture.



Real Time Factor: 1.00 Sim Time: 00 00:00:07.702 Real Time: 00 00:00:08.366 Iterations: 7702 FPS: 39.3153

Figure 7.23: TurtleBot3 House Highlighted

7.4.1 TurtleBot3 Installation

The first step is to install the ROS on your operating system. Inside the catkin workspace folder inside the root folder (/catkin_ws), source folder should be created (/src)

```
cd ~/catkin_ws/src/
```

Afterwards, cloning the ROS melodic using the following command

```
git clone -b melodic-devel https://github.com/ROBOTIS-GIT/
turtlebot3_simulations.git
```

Finally, building the whole ROS project using catkin_make command

```
cd ~/catkin_ws && catkin_make
```

7.4.2 Customizing TurtleBot3 Gazebo

The file with the following path " /catkin_ws/src/ROS/turtlebot3_simulations/turtlebot3_description /urdf/turtlebot3_waffle_pi.gazebo.xacro" contains the design for the turtlebot3 house skeleton, the user have the opportunity to change the default values for the sensors, as well as the colors of your TurtleBot3 as shown in the following image (Spiderman TurtleBot3)

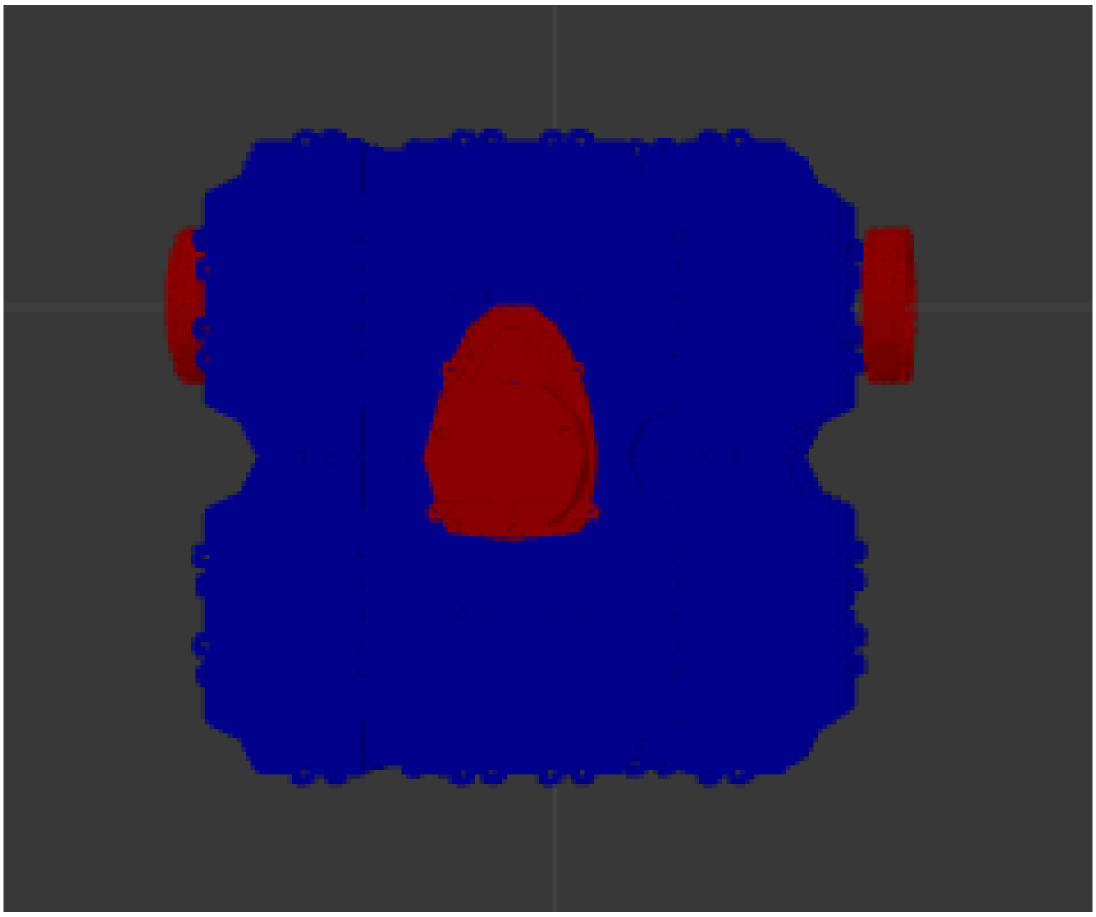
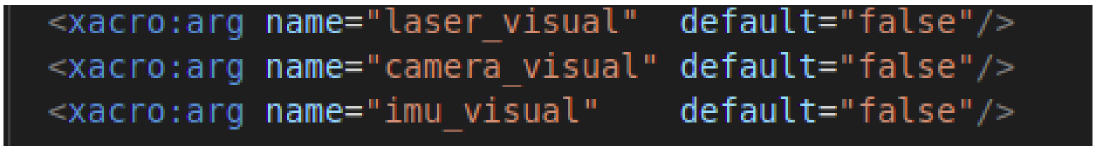


Figure 7.24: Customized TurtleBot3

To modify the sensors default values, whether to be running or to be idle until a topic subscribe to it using the following lines



```
<xacro:arg name="laser_visual"  default="false"/>
<xacro:arg name="camera_visual" default="false"/>
<xacro:arg name="imu_visual"    default="false"/>
```

Figure 7.25: TurtleBot3 Sensors

7.4.3 Add Models

For your environment you can insert more models to your built environment using gazebo simulator GUI from the following menu click insert button

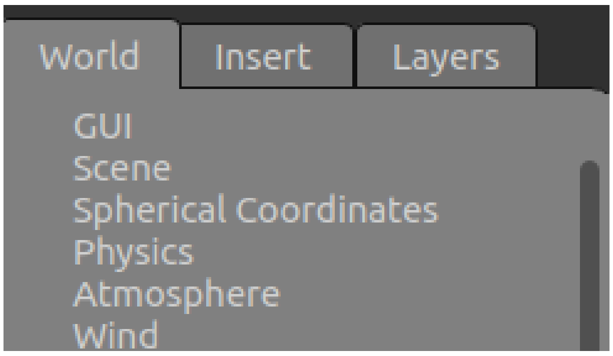


Figure 7.26: Gazebo Tabs

From the following menu, you can choose whatever model are interested to add. Drag and drop the model into your environment
Note: You can find all your local models on this path " /.gazebo/models"

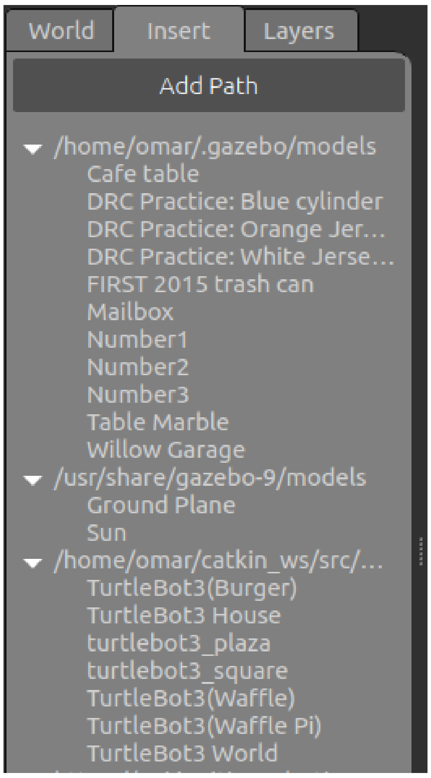


Figure 7.27: Installed Models

7.4.4 Run TurtleBot3 House

Firstly, you should open your terminal and be directed to catkin_ws folder. Fundamentally, execute the following command to source your command line for the environment variables and default values for the ROS

```
source /opt/ros/melodic/setup.bash
```

Then, exporting the environment variable TURTLEBOT3_MODEL to the model used which is waffle_pi in our case

```
export TURTLEBOT3_MODEL=waffle_pi
```

Ultimately, launching the master node of gazebo using the turtlebot3_house.launch file that contains the configurations and needed packages as well.

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

TeleOp TurtleBot3 Node

This teleop node is offered by the TurtleBot3 Gazebo Simulation in order to teleport and control your robot using the keys of the keyboard. Open a new terminal, then export the variable TURTLEBOT3_MODEL to the model used, then launching the node for teleop using this command

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Collision Avoidance Node

Collision Avoidance Node is published by the TurtleBot3 Gazebo Simulation in order to move your robot randomly. The feature offered by this node is that it avoid to collide with any item. Open a new terminal, then export the variable TURTLEBOT3_MODEL to the model used, then launching the node for turtlebot3_simulation using this command:

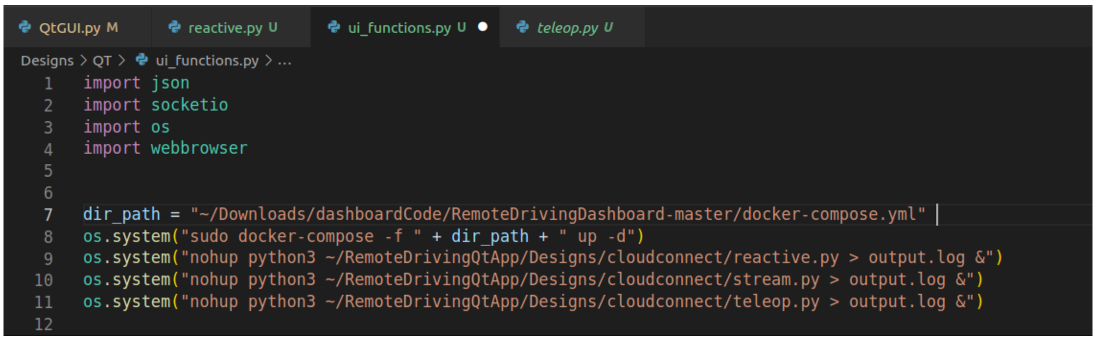
```
roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

8 User Manual

In this section we will present the user instructions to use our program. The application can be divided into two parts, the ROS part which belongs completely to the user and the Dashboard part for which the usage is generic. For the ROS part we'll show our under-study system as an example, and, as we go through the user guide, we'll highlight the what the Dashboard expects from the ROS in order to function correctly.

8.1 QT Configurator Dashboard

The first thing the user needs to do is to open the QT configurator. This can be done by opening the RemoteDrivingQtApp repository, and running the QtGUI.py script. Noting that the following lines should be adjusted according to the location of the files on the system:



```python
import json
import socketio
import os
import webbrowser


dir_path = "~/Downloads/dashboardCode/RemoteDrivingDashboard-master/docker-compose.yml"
os.system("sudo docker-compose -f " + dir_path + " up -d")
os.system("nohup python3 ~/RemoteDrivingQtApp/Designs/cloudconnect/reactive.py > output.log &")
os.system("nohup python3 ~/RemoteDrivingQtApp/Designs/cloudconnect/stream.py > output.log &")
os.system("nohup python3 ~/RemoteDrivingQtApp/Designs/cloudconnect/teleop.py > output.log &")

```

Figure 8.1: Paths To Adjust

The first line is the location of the docker-compose file, the other lines correspond to the cloud-connect scripts needed for the communication with the ROS system.

After that we can safely run the QtGUI.py script, and choose the following configurations:

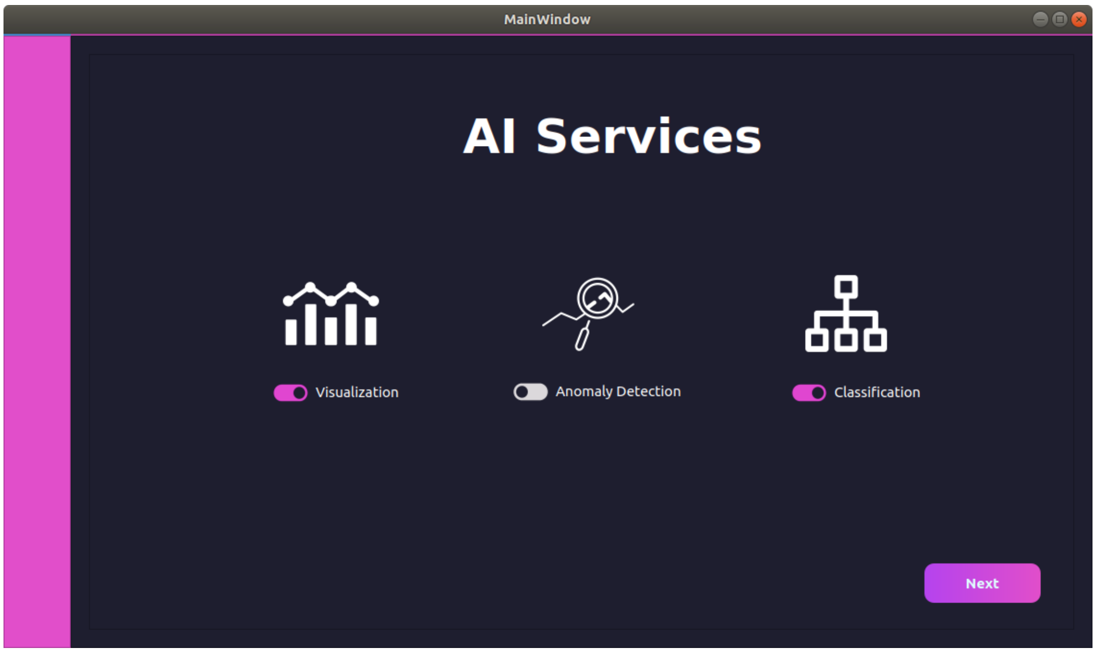


Figure 8.2: AI Services

Continue with pressing next and choosing the following configurations:

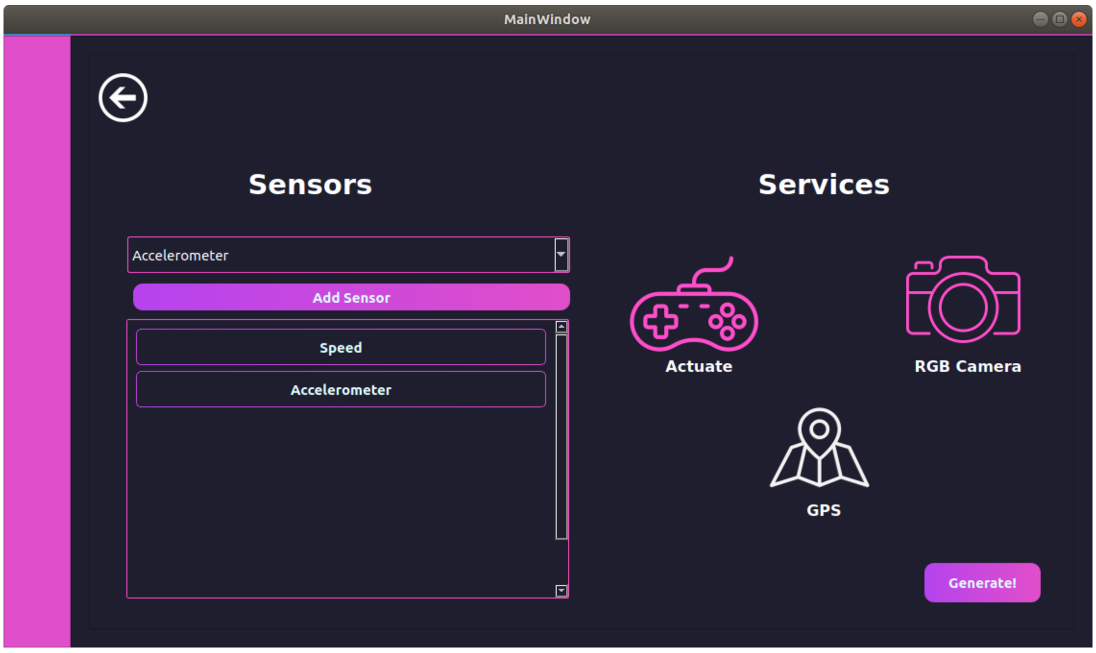


Figure 8.3: Services

Finally, press generate to automatically open the dashboard in the browser. As previously explained, the dashboard will be created according to the configurations entered in the QT, the below figure is an example.

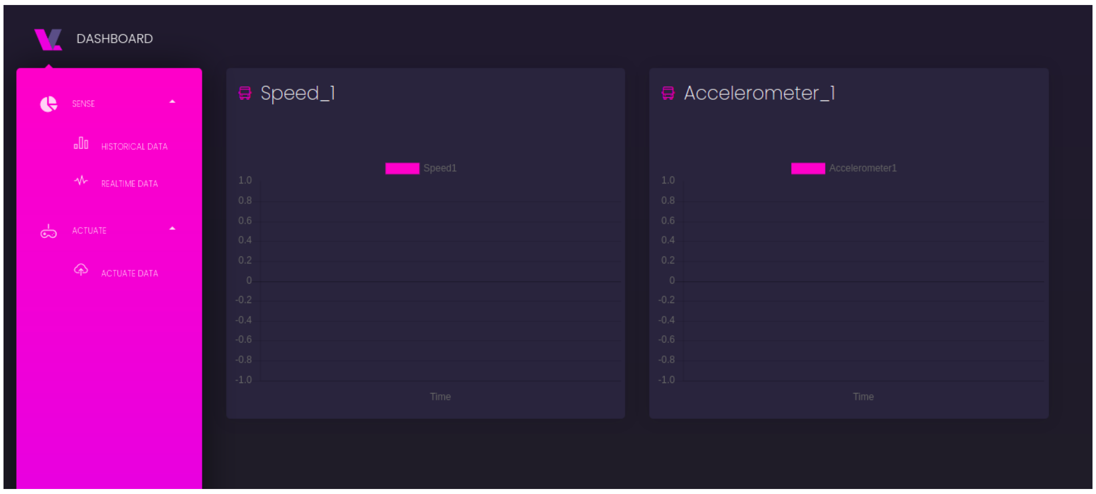
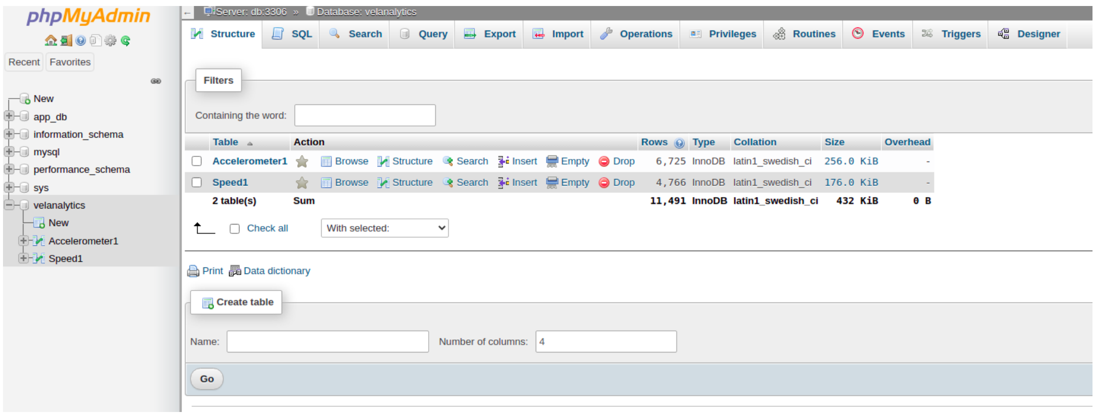


Figure 8.4: Snippet Of The Dashboard

The database is also created and can be accessed through phpmyadmin using the link: http://0.0.0.0:8081/ and selecting the velanalytics database.


phpMyAdmin

Recent Favorites

New
app_db
information_schema
mysql
performance_schema
sys
velanalytics
New
Accelerometer1
Speed1

Server: db:3306 » Database: velanalytics

Structure SQL Search Query Export Import Operations Privileges Routines Events Triggers Designer

Filters

Containing the word:

	Table	Action							Rows	Type	Collation	Size	Overhead
☐	Accelerometer1	☆	Browse	Structure	Search	Insert	Empty	Drop	6,725	InnoDB	latin1_swedish_ci	256.0 KiB	-
☐	Speed1	☆	Browse	Structure	Search	Insert	Empty	Drop	4,766	InnoDB	latin1_swedish_ci	176.0 KiB	-
	2 table(s)	Sum							11,491	InnoDB	latin1_swedish_ci	432 KiB	0 B

☐ Check all With selected:

Print Data dictionary

Create table

Name: Number of columns: 4

Go

Figure 8.5: Snippet Of The Database

8.2 ROS Case Study

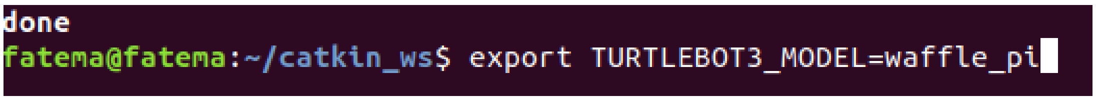
The communication between the ROS system and the Dashboard is done through the cloudconnect gateway, which is running automatically with the generation of the Dashboard.

The details of the ROS nodes and topics in our case study can be found in the ROS repository attached with the project.

In order to run the ROS subsystem, the following steps should be done.

1st Command to initialize the turtlebot environment:

```
export TURTLEBOT3_MODEL=waffle_pi
```

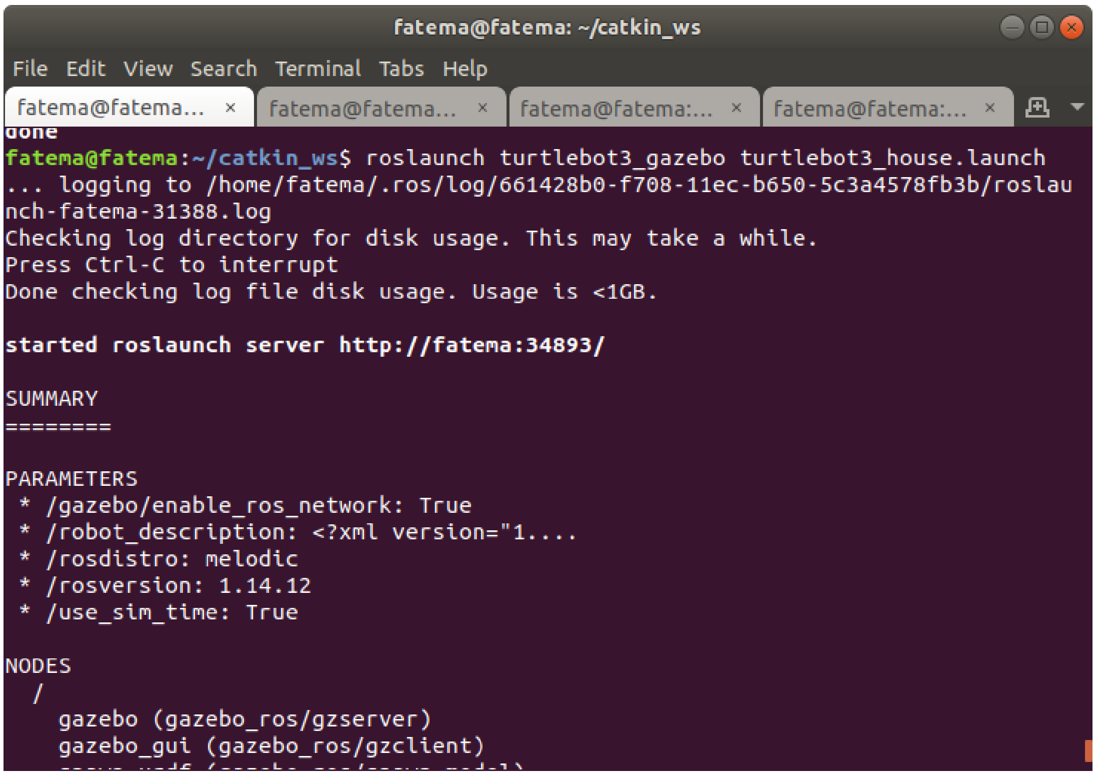


```
done
fatema@fatema:~/catkin_ws$ export TURTLEBOT3_MODEL=waffle_pi
```

Figure 8.6: 1st Command

2nd Command to start the gazebo simulator with the initialized environment and turtlebot:

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```



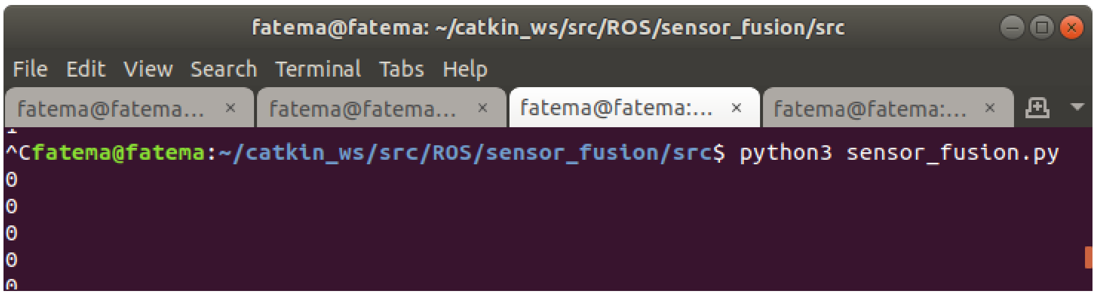
```
done
fatema@fatema:~/catkin_ws$ roslaunch turtlebot3_gazebo turtlebot3_house.launch
... logging to /home/fatema/.ros/log/661428b0-f708-11ec-b650-5c3a4578fb3b/roslau
nch-fatema-31388.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://fatema:34893/

SUMMARY
========

PARAMETERS
 * /gazebo/enable_ros_network: True
 * /robot_description: <?xml version="1....
 * /rosdistro: melodic
 * /rosversion: 1.14.12
 * /use_sim_time: True

NODES
  /
    gazebo (gazebo_ros/gzserver)
    gazebo_gui (gazebo_ros/gzclient)
```

Figure 8.7: 2nd Command

3rd Command to run the sensor$_fusion.pyscript$:

```
python3 sensor_fusion.py
```

This script publishes the sensors' data to the topic /metric, so the cloudconnect (reactive node) can subscribes to the same topic in order to get these data. It's located in the /ROS/`sensor_fusion`/src directory.



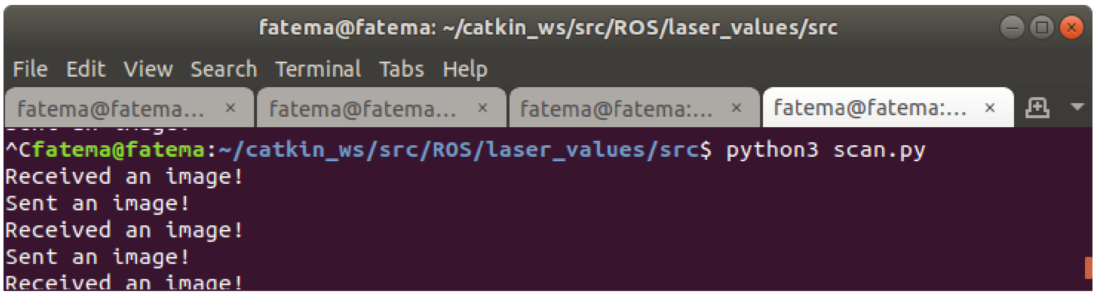
```
fatema@fatema: ~/catkin_ws/src/ROS/sensor_fusion/src
^Cfatema@fatema:~/catkin_ws/src/ROS/sensor_fusion/src$ python3 sensor_fusion.py
0
0
0
0
```

Figure 8.8: 3rd Command

4th Command to run the scan.py script:

```
python3 scan.py
```

This script publishes images captured from the camera to the /images topic where the cloudconnect node (Streaming node) subscribes to the same topic, so it get the images and sends them to the dashboard. It's located in the /ROS/`laser_values`/src directory.



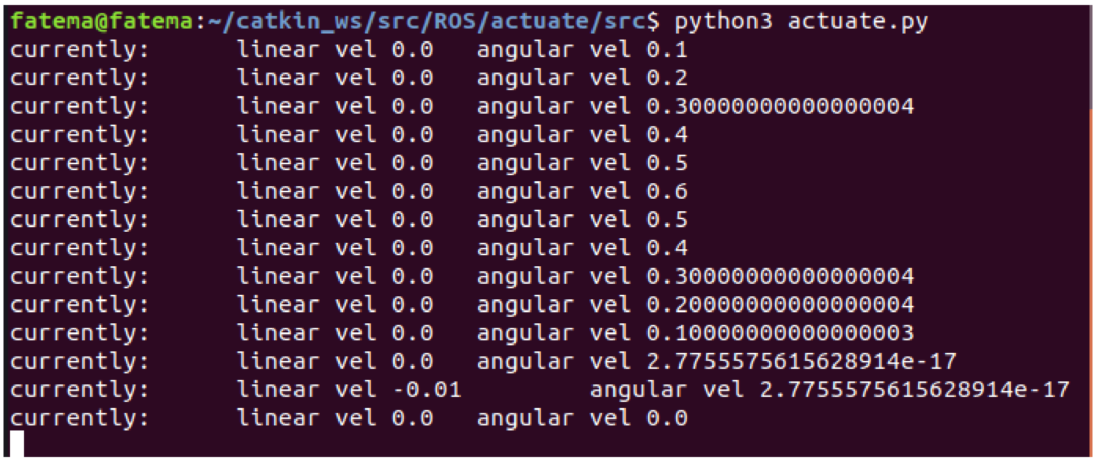
```
fatema@fatema: ~/catkin_ws/src/ROS/laser_values/src
^Cfatema@fatema:~/catkin_ws/src/ROS/laser_values/src$ python3 scan.py
Received an image!
Sent an image!
Received an image!
Sent an image!
Received an image!
```

Figure 8.9: 4th Command

5th Command to run the actuate.py script:

```
python3 actuate.py
```

This script subscribes from the /control topic that the cloudconnect node (TeleOp. node) publishes to it the actuation direction it got from the dashboard. It's located in /ROS/`actuate`/src directory.



```
fatema@fatema:~/catkin_ws/src/ROS/actuate/src$ python3 actuate.py
currently:      linear vel 0.0   angular vel 0.1
currently:      linear vel 0.0   angular vel 0.2
currently:      linear vel 0.0   angular vel 0.30000000000000004
currently:      linear vel 0.0   angular vel 0.4
currently:      linear vel 0.0   angular vel 0.5
currently:      linear vel 0.0   angular vel 0.6
currently:      linear vel 0.0   angular vel 0.5
currently:      linear vel 0.0   angular vel 0.4
currently:      linear vel 0.0   angular vel 0.30000000000000004
currently:      linear vel 0.0   angular vel 0.20000000000000004
currently:      linear vel 0.0   angular vel 0.10000000000000003
currently:      linear vel 0.0   angular vel 2.7755575615628914e-17
currently:      linear vel -0.01         angular vel 2.7755575615628914e-17
currently:      linear vel 0.0   angular vel 0.0
```

Figure 8.10: 5th Command

After these commands, the Dashboard is ready to be used. The user now can:

- Monitor live data from Real-time page.
- Display data from specific dates from historical page
- Watch live-stream images from actuate page
- Control the robot from the same page (actuate page) by increasing or decreasing its speed or changing its direction.

8.3 Cloud Connect Infrastructure

An important thing to be considered is the message format of the data published to the topics of the cloud connect; metric, images, and control topics.

Metric

For the topic /metric, the sensors' data are published to it by ROS subsystem, then the reactive cloud connect node subscribes to the same topic to get these data. The format of these data must be as follows:

```
{'name':<sensor_name_and_number>,'magnitude':<value>}
```

For example; 'name': 'speed1', 'magnitude': 1.4

Images

For the topic /images, the camera's images are published to it by ROS subsystem, then the streaming cloud connect node subscribes to the same topic to get the camera's images.
Note: the images sent to the topic must be of type sensor_msgs.msg.Image

Control

For the topic /control, the actuation direction is published to it by TeleOp. cloud connect node, then the ROS subsystem subscribes to the same topic to get this data.
Note: the data in this topic is just a character that specifices the direction of the actuation;
[w $\rightarrow$ up], [x $\rightarrow$ down], [a $\rightarrow$ left], [d $\rightarrow$ right], [s $\rightarrow$ stop]

So to sum it up for any other ROS subsystem, all the user has to do in his subsystem is:

- Publish sensors' data with the appropriate format to topic /metric
- Publish camera images with the appropriate format to topic /images
- Subscribes the actuation direction from topic /control

9 Conclusion

In this thesis, we have proposed the methodologies to develop a suitable software architecture model for the Cloud-Connected Vehicle application. We want to ensure consistency between two different infrastructures such as the ROS module and the Dashboard module. On account of the fact that the application is subject to change according to different ROS modules, the application must be always ready with a re-configurable system. To reach a solution to generalize our dashboard and provide certain services the user requires for his ROS module, we have used the following technologies and approaches:

- Docker container to deploy our dashboard and database on as it decreases the deployment time
- Qt Configurator which is a GUI application for the user to send the configuration of the ROS subsystem to the dashboard for a new dashboard that supports this ROS subsystem
- A dynamic frontend for the user that provides different services according to the user's specific ROS subsystem
- Cloud-connect nodes which are generalized for different ROS subsystems and used for ROS and dashboard communication
- SocketIO to facilitate the communication of the data between the two modules

Our application's entry point is the Qt Configurator where the user selects his required AI Services, Sensors, and Services. On Generate, the Docker Container images gets launched and is running as well as the Dashboard is opened automatically on the browser. The Dashboard visualizes the charts for the sensors used and the Services selected widgets. Also, the Database Tables are created dynamically according to the Configurator selections as well.

In summary, these configurations provide us with the requirements we need for the Cloud-Connected Vehicle software application but there are still some matters to be considered for future work that are going to be discussed in the next chapter.

10 Our Contribution

After detailing all the aspects of the project, we'll sum up in this section what we did so far in the project in the following points:

10.1 1st Semester

- A detailed study of the system, its aspect and problem formulation.
- Understanding and visualizing the system's architecture using UML diagrams.
- Creating the docker'ized cloud containing the dashboard and database images.
- Configuring the code so that the database tables gets created once and dynamically based on the user's input.
- Creating a prototype for the configurator to test out a generalized dashboard

10.2 2nd Semester

- Designing a user friendly interface for the Qt configurator
- Adding the Qt component as the design entry of the system.
- Automating the Docker'ized Private Cloud Generation.
- Generalizing the 3 web pages of the dashboard by making it customized according to the user input from the configurator.
- Historical Data page is configured according to the user input and connected to the database component.
- Adding the AI Services in the docker'ized cloud (Anomaly Detection and Object Classification).
- Generalizing the Services Feature (Actuation, GPS and Streaming) on the dashboard.
- Implemented the cloud-connect nodes that can be used with any ROS subsystem
- Finalizing and conducting our Case-study using ROS.

11 Future Work

5G Technology

Our project depends mainly on offering services over-the-cloud, where sending and receiving of data takes place between the vehicle and the dashboard and is needed to be fast enough to function properly and as expected. Since the robot works on 50Hz to process and send the data. Therefore, to control and communicate with a robot over a wireless network, it is needed for this network to be of an appropriate delay time relative to the processing time of the robot meaning to be at least equal or less than 20 msec. Researching the delay times of 4th and 5th Generation Wireless Networks; 4G's latency rate of sending and receiving information is 200 msec which is extremely greater than the allowed time. While the 5G came offering considerably less latency rate which is 1 msec giving a chance for further work to be accomplished placing the system over the cloud and controlling it through a 5G network rather than communicating over a local network which is what is used currently.

JavaScript Framework

Another suggested improvement for our project would be to rebuild the dashboard's front-end using a reliable web framework like Vue.JS, React, etc. This would be a better approach than using simple HTML, CSS and JavaScript codes, since it provides a consistent and standardized front-end methodology which makes it easier to track the dependencies between components and to extend the code efficiently if needed. It also assures a faster and nicer user experience by making application perform more like a native application while implementing features such as client-side routing which in return increases the website's performance.

Digital Twin of Automated Guided Vehicle

Fundamentally, a digital twin is a computer program that uses real world data to create simulations that can predict how a product or process will perform. These programs can integrate the internet of things, artificial intelligence and software analytics to enhance the output. The future work is going to contain a physical turtlebot which would act as a mirror for our gazebo turtlebot one. To elaborate, any actions happen to the gazebo simulation should be reflected to the physical turtlebot and vise versa. Such approach would be extremely beneficial in various fields, for instance in the remote controlling of robotics.

Bibliography

[1] Gaudenz Alder. Uml diagrams maker.

[2] Carol Fairchild and Thomas L. Harman. *ROS Robotics By Example*. Packt Publishing, 2016.

[3] Ruchik Mishra and Arshad Javed. Ros based service robot platform. pages 55–59, 2018.

[4] Ali Mustafa, Mohammed Aal-nouman, and Osama Awad. Cloud-based vehicle tracking system. *Iraqi Journal of Information Communications Technology*, 2:21–30, 02 2020.

[5] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[6] Sebastian Thöne. *Dynamic Software Architectures: A Style-Based Modeling and Refinement Technique with Graph Transformations*. 2005.