Clarkson University

Unmanned Ground Vehicle Autonomous Navigation Research

Summer 2025

Rowan Wysocki

Dr.Chen Liu

Table of Contents

[Abstract 5](#_Toc204173093)

[Agile X Scout Mini (Real Space Navigation) 5](#_Toc204173094)

[Hardware and Software Composition 5](#_Toc204173095)

[Intel NUC 5](#_Toc204173096)

[Scout Mini Robot (Scout Base) 6](#_Toc204173097)

[Ros2 (Robot Operating System V2) 6](#_Toc204173098)

[Intel Real Sense RGB-Depth Camera 7](#_Toc204173099)

[Livox HAP LiDAR SDK and Drivers 9](#_Toc204173100)

[Cloud Fuser Node 10](#_Toc204173101)

[Pointcloud to Laserscan 11](#_Toc204173102)

[Robot Localization 11](#_Toc204173103)

[Slam Toolbox 12](#_Toc204173104)

[Nav2 13](#_Toc204173105)

[Collision Monitor 13](#_Toc204173106)

[Explore Lite 14](#_Toc204173107)

[Data Collection Using Livox HAP LiDAR 14](#_Toc204173108)

[Overview 14](#_Toc204173109)

[Launch Livox HAP LiDAR 15](#_Toc204173110)

[Record Rosbag 15](#_Toc204173111)

[Generate PCD Files 15](#_Toc204173112)

[Play and Review Recording (Optional) 15](#_Toc204173113)

[Scout Mini Autonomous Navigation (Livox & Real Space) 16](#_Toc204173114)

[Overview 16](#_Toc204173115)

[Shortcut commands: 16](#_Toc204173116)

[Launch Livox HAP LiDAR 17](#_Toc204173117)

[Open Can Port for Scout Base Communication 17](#_Toc204173118)

[Activate and Connect Scout Base Communication 17](#_Toc204173119)

[Launch Scout Mini Description 18](#_Toc204173120)

[Launch Point Cloud to Laser Scan Converter 18](#_Toc204173121)

[Run SLAM Toolbox 18](#_Toc204173122)

[Launch Nav2 18](#_Toc204173123)

[Run Collision monitor 19](#_Toc204173124)

[Use Rviz2 to Visualize Mapping 19](#_Toc204173125)

[Additional Information: 19](#_Toc204173126)

[Create, Save, and Navigate with a Custom Map 19](#_Toc204173127)

[Overview 19](#_Toc204173128)

[Initialize the Robot 20](#_Toc204173129)

[Start SLAM 20](#_Toc204173130)

[Build and Save the Map 20](#_Toc204173131)

[Load the Map and Navigate 20](#_Toc204173132)

[Multi-Sensor Point Cloud Data Fuser (Custom Node) 21](#_Toc204173133)

[Implementation Details 21](#_Toc204173134)

[Parameters 22](#_Toc204173135)

[Installation and Launch 22](#_Toc204173136)

[Runtime Behavior 22](#_Toc204173137)

[Example Use Case 23](#_Toc204173138)

[TurtleBot3 in GAZEBO Simulator 23](#_Toc204173139)

[Software Composition 23](#_Toc204173140)

[TurtleBot3 23](#_Toc204173141)

[GAZEBO Simulator 24](#_Toc204173142)

[GAZEBO Simulation Test 25](#_Toc204173143)

[Overview 25](#_Toc204173144)

[Launch Simulation 25](#_Toc204173145)

[Check Ros2 Topics 25](#_Toc204173146)

[Configure Ros2 26](#_Toc204173147)

[Create Bridge for a Topic 26](#_Toc204173148)

[Use Teleop Keyboard Control 26](#_Toc204173149)

[Visualize LiDAR 26](#_Toc204173150)

[TurtleBot3 GAZEBO SLAM 27](#_Toc204173151)

[Overview 27](#_Toc204173152)

[Launch Robot and Environment 27](#_Toc204173153)

[Start Keyboard Control 27](#_Toc204173154)

[Run SLAM Toolbox 28](#_Toc204173155)

[Save a Map 28](#_Toc204173156)

[Run Nav2 on a Saved Map 28](#_Toc204173157)

[Scout Mini in GAZEBO Simulator 28](#_Toc204173158)

[Software Composition 29](#_Toc204173159)

[Scout Mini Simulation Description 29](#_Toc204173160)

[Running the Scout Mini in an Empty GAZEBO Environment 29](#_Toc204173161)

[Overview 29](#_Toc204173162)

[Launch Scout Mini in GAZEBO 30](#_Toc204173163)

[Enable Keyboard Control 30](#_Toc204173164)

[Run SLAM Toolbox 30](#_Toc204173165)

[Launch Nav2 31](#_Toc204173166)

[Visualize in Rviz2 31](#_Toc204173167)

[Appendix 31](#_Toc204173168)

[Creating a Custom Bash Script 31](#_Toc204173169)

[Creating a Custom Python Launch File 32](#_Toc204173170)

# Abstract

Write abstract here

# Agile X Scout Mini (Real Space Navigation)

The Scout Mini is a compact unmanned ground vehicle (UGV) designed for indoor and outdoor robotic applications. We use the ROS 2 software framework, which provides modular, real-time capabilities for robotic control and perception. For autonomous navigation, we integrate **SLAM Toolbox**, which enables real-time mapping and localization, along with **Nav2**, which provides path planning and obstacle avoidance. The robot is equipped with an Intel NUC onboard computer and supports either a **Livox HAP LiDAR**, offering high-precision scanning at a lower cost, or a **RoboSense Helios-16 LiDAR**, which provides dense 360° point cloud data for enhanced environmental awareness.

## Hardware and Software Composition

### Intel NUC

The Scout Mini is powered by an **Intel NUC (Next Unit of Computing)**, a compact yet powerful mini-PC well-suited for embedded robotics applications. Despite its small form factor, the Intel NUC delivers strong computational performance, capable of handling real-time processing for SLAM, LiDAR data interpretation, and navigation planning. Its multiple USB and Ethernet ports support seamless integration with sensors such as LiDARs and IMUs, while its solid-state drive (SSD) and modern processors enable fast data access and low-latency communication. The NUC’s efficient thermal design and support for Linux make it an ideal onboard computer for mobile robots operating in dynamic environments.

This onboard computer is Linux-based, running **Ubuntu 22.04 (Jammy Jellyfish)**, a long-term support (LTS) distribution that ensures stability and security for robotics applications. This environment provides compatibility with ROS 2 Humble Hawksbill, allowing seamless integration of navigation, perception, and control modules. The use of Ubuntu LTS also ensures access to a wide range of development tools, libraries, and community support essential for robust robotic operation.

To interact with the onboard Intel NUC remotely, we use **NoMachine**, a remote desktop application that provides a graphical interface for accessing the NUC over a network. Once the NUC is powered on and connected to the same network, we connect from a host computer using the NoMachine client, which allows us to launch terminals, monitor ROS nodes, visualize data in RViz, and manage files as if we were directly interfacing with the device. This setup simplifies development and testing, especially when the robot is mobile or operating in hard-to-reach locations, by eliminating the need for a dedicated display or peripherals attached to the NUC.

1. **Install NoMachine** on both your host computer and the target device (e.g., the Intel NUC).
2. **Ensure both devices are on the same network** and that NoMachine is running on the NUC.
3. Open **NoMachine** on your host computer.
4. Click **"Add"** or **"New"** to create a new connection.
5. Choose **"NX" protocol** and click **Continue**.
6. Enter the **IP address or hostname** of the NUC and click **Continue**.

* IP: 128.153.165.95

1. Leave default settings or adjust if needed, then click **Done**.
2. Double-click the new connection and **log in with the NUC's username and password**.

* Username: scoutmini2
* Password: newmini2

### Scout Mini Robot (Scout Base)

The Scout Mini requires specialized software to operate effectively. A **description package** is needed to define the robot’s physical structure, including its dimensions, sensor placements, and joint configurations, which are essential for simulation and control. For motor control, the robot can be operated using a handheld remote controller for direct manual input, or through **keyboard teleoperation** via the onboard computer using ROS 2-compatible tools.

It has a dedicated workspace that houses the following packages:

* scout\_description

### Ros2 (Robot Operating System V2)

**ROS 2 (Robot Operating System 2)** is an open-source middleware framework that integrates various robotics components such as perception, control, and planning into a unified system. It acts as a **software architecture** that ties together individual packages, allowing them to communicate through a publish/subscribe messaging system, service calls, and action interfaces. This modular approach enables developers to build complex robotic behaviors from interchangeable and reusable components, while ensuring real-time performance, scalability, and cross-platform compatibility. ROS 2 is especially well-suited for multi-robot systems, edge computing, and modern industrial or research applications due to its support for DDS (Data Distribution Service) and robust lifecycle management.

The core **ROS 2 installation**, including built-in tools and commonly used packages like slam\_toolbox and nav2, is located in:

**/opt/ros/humble/**

with package-specific resources stored under:

**/opt/ros/humble/share/**

These packages are installed system-wide via a package manager (e.g., apt) and are maintained separately from custom or third-party code. o install the full ROS 2 desktop environment along with slam\_toolbox and nav2, we ran the following commands:

* sudo apt update
* sudo apt install ros-humble-desktop

This method provides a stable, pre-built environment for development and ensures that core functionalities are readily available without the need for building from source.

In contrast, other packages such as scout\_description or livox\_ros\_driver are typically cloned from source into user-defined workspaces like ~/scout\_ws or ~/ws\_livox. These custom packages are built locally with colcon and sourced manually, allowing for more flexible development while keeping the system installation stable and clean.

### Intel Real Sense RGB-Depth Camera

An **RGB-D camera**, like the Intel RealSense D435, captures both color (RGB) images and depth (D) data, making it a powerful sensor for robotics, AR/VR, and spatial computing. The depth component allows a system to perceive the 3D structure of the environment, enabling tasks like object detection, scene reconstruction, obstacle avoidance, and simultaneous localization and mapping (SLAM). In **ROS 2**, RGB-D cameras integrate seamlessly to provide spatial context to robots, which is essential for navigation, perception, and manipulation. With ROS 2’s real-time capabilities and modular design, RGB-D data can be published over topics and used in nodes for applications like mapping, path planning, and human-robot interaction.

We installed the Intel RealSense SDK 2.0 to enable the D435 camera on Ubuntu 22.04. First, we updated the system and installed the necessary dependencies:

* sudo apt update && sudo apt upgrade
* sudo apt install git wget cmake build-essential libssl-dev libusb-1.0-0-dev libudev-dev pkg-config libgtk-3-dev libglfw3-dev libgl1-mesa-dev libglu1-mesa-dev v4l-utils

Then we cloned the SDK from GitHub and set up device permissions:

* git clone https://github.com/IntelRealSense/librealsense.git
* cd librealsense
* sudo ./scripts/setup\_udev\_rules.sh
* mkdir build && cd build
* cmake ../ -DBUILD\_EXAMPLES=true
* make -j$(nproc)
* sudo make install

With the camera plugged in, we verified it using:

* rs-enumerate-devices
* realsense-viewer

These confirmed that both RGB and depth streams were available. We chose not to patch the kernel (yet) since our camera worked without it, despite running on Linux kernel 6.8, which is not officially supported. The system now streams RGB-D data and is ready for integration into ROS 2 using the RealSense ROS wrapper.

To integrate the RealSense D435 with ROS 2, we installed the official **RealSense ROS 2 wrapper**, which publishes camera data as ROS topics for use in perception, SLAM, and navigation stacks. First, we ensured we had a working ROS 2 workspace (e.g., ros2\_ws) and navigated to its src directory. We then cloned the ROS 2 wrapper repository using:

* cd ~/ros2\_ws/src
* git clone -b ros2-development https://github.com/IntelRealSense/realsense-ros.git

After cloning, we installed dependencies and built the workspace:

* cd ~/ros2\_ws
* rosdep install --from-paths src --ignore-src -r -y
* colcon build --symlink-install

Finally, we sourced the environment and launched the camera node:

* source install/setup.bash
* ros2 launch realsense2\_camera rs\_launch.py

This started streaming color, depth, and infrared data on ROS 2 topics like /camera/color/image\_raw and /camera/depth/image\_rect\_raw, making the camera fully accessible to other ROS 2 nodes and packages for further processing.

To run the camera, run the following command:

* ros2 launch realsense2\_camera rs\_launch.py enable\_rgbd:=true enable\_sync:=true align\_depth.enable:=true enable\_color:=true enable\_depth:=true pointcloud.enable:=true

### Livox HAP LiDAR SDK and Drivers

To interface the Livox HAP LiDAR with ROS 2, we use two essential components: **Livox-SDK2** and **livox\_ros\_driver2**.

**Livox-SDK2** is the official C++ software development kit provided by Livox for interacting with their LiDAR sensors at a low level. It handles communication protocols, data packet decoding, and device configuration, providing a foundation for building custom LiDAR applications or middleware. This SDK is necessary to interpret raw data streams from the sensor and expose usable point cloud data.

**livox\_ros\_driver2** is the ROS 2 driver built on top of Livox-SDK2. It publishes the LiDAR data to standard ROS topics such as /livox/lidar as sensor\_msgs/PointCloud2, making it compatible with common perception tools like RViz, SLAM Toolbox, and Nav2. The driver also includes launch files for initializing the sensor and configuring RViz for visualization.

In our setup, we installed the **Livox HAP LiDAR drivers** by cloning both the **livox\_ros\_driver2** package and its core dependency, **Livox-SDK2**, into a dedicated ROS 2 workspace (e.g., ~/ws\_livox). This source-based installation provides full compatibility with our hardware and allows us to modify configuration files or frame settings as needed for integration with other ROS 2 tools.

We began by creating the workspace and cloning the repositories:

* mkdir -p ~/ws\_livox/src
* cd ~/ws\_livox/src
* git clone https://github.com/Livox-SDK/Livox-SDK2.git
* git clone https://github.com/Livox-SDK/livox\_ros2\_driver.git -b humble

After cloning, we ensured all dependencies were installed and then built the workspace:

* cd ~/ws\_livox
* rosdep install --from-paths src --ignore-src -r -y
* colcon build

Once the build completed successfully, we sourced the workspace to overlay the environment:

* source install/setup.bash

We use the config file **HAP\_config.json**

### Cloud Fuser Node

To unify perception from both RGB-D and LiDAR sensors, we developed a custom ROS 2 node named cloud\_fuser, implemented in C++ and located in the scout\_ws workspace. The node synchronizes point clouds from the Intel RealSense D435 and Livox HAP LiDAR, transforms them into a common frame (base\_link), and merges them into a single sensor\_msgs/PointCloud2 message published on the /fused\_points topic.

The fusion process involves the following key steps:

1. **TF-based Frame Alignment**: The node uses tf2\_ros to lookup transforms from each sensor's frame to base\_link, then converts them to Eigen::Matrix4f for use in PCL.
2. **Point Cloud Conversion and Transformation**: It converts ROS messages to PCL format (pcl::PointCloud<pcl::PointXYZ>), applies the appropriate transform to each cloud, and aligns them in the robot's base frame.
3. **Filtering**: A pcl::PassThrough filter removes unwanted data outside a configured z-range (e.g., 0.05–0.4 m). Parameters like z\_min, z\_max, and voxel\_size are configurable at runtime and dynamically updateable through ROS 2 parameter callbacks.
4. **Merging and Publishing**: After filtering, the two clouds are concatenated into one and optionally passed through a voxel filter or outlier removal filter (commented out in the current version). The fused cloud is published with the appropriate timestamp and frame ID.

The node uses message\_filters::ApproximateTime to synchronize incoming RGB-D and LiDAR clouds:

* /camera/camera/depth/color/points (RealSense)
* /livox/lidar (Livox HAP)

Launch file:

* cloud\_fuser/launch/cloud\_fuser\_launch.py

To run the node:

* ros2 launch cloud\_fuser cloud\_fuser\_launch.py

This fused point cloud can then be passed to the pointcloud\_to\_laserscan node to enable standard 2D navigation and SLAM algorithms using combined 3D spatial information from multiple sensors.

### Pointcloud to Laserscan

**PointCloud to LaserScan** is a ROS 2 utility that converts 3D LiDAR data (point clouds) into 2D laser scan data, simulating the output of a traditional 2D laser scanner. This is useful for algorithms or packages such as SLAM or obstacle avoidance that expect 2D LaserScan messages as input. The conversion process slices the 3D point cloud horizontally at a defined height and projects the points onto a 2D plane, effectively creating a virtual 2D scan from a 3D sensor. This allows 3D LiDARs like the Livox HAP or RoboSense to be used with existing 2D navigation and mapping tools without modification.

For our setup, **PointCloud to LaserScan was installed using APT**, as it is included in the ROS 2 ecosystem. We used the command:

* sudo apt install ros-humble-pointcloud-to-laserscan

to install the package and its dependencies. Once installed, the node is available under the ROS 2 environment in:

/opt/ros/humble/share/pointcloud\_to\_laserscan/

This tool can be launched directly or included in launch files to convert real-time LiDAR point cloud data into LaserScan messages, enabling compatibility with 2D-based algorithms like SLAM Toolbox and certain navigation plugins.

Depending on Livox only or fused data, we have different launch files:

* Livox only:
  + **sample\_pointcloud\_to\_laserscan\_livox\_launch.py**
* Fused data:
  + **sample\_pointcloud\_to\_laserscan\_ALL\_launch.py**

### Robot Localization

**Robot Localizatio**n is a ROS 2 package that provides state estimation through sensor fusion. In our setup, it takes **IMU** data from the **Livox LiDAR** and fuses it with **odometry** data from the base to produce a more accurate estimate of the robot’s position and orientation over time. This fused estimate is essential for reliable navigation, especially when using SLAM Toolbox and Nav2. We installed the robot\_localization package using the system package manager for ROS 2 Humble:

* sudo apt update
* sudo apt install ros-humble-robot-localization

This installs the necessary nodes (ekf\_node, ukf\_node, etc.) and places them in the ROS 2 environment under:

* /opt/ros/humble/share/robot\_localization/

Once installed, we configured and launched the **ekf\_node** to publish a continuous transform between **odom** and **base\_link**, enabling downstream packages like Nav2 and RViz to track the robot’s motion accurately in real time.

We use a custom config file called **ekf\_custom.yaml**.

### Slam Toolbox

**SLAM Toolbox** is a powerful ROS 2-compatible package that enables real-time **Simultaneous Localization and Mapping (SLAM)** for mobile robots. It provides both online and offline mapping capabilities using 2D LiDAR data, allowing a robot to build a map of an unknown environment while simultaneously tracking its own position within it. Designed with flexibility and performance in mind, SLAM Toolbox supports pose graph optimization, loop closure, and serialization of maps for reuse across sessions. It integrates seamlessly with other ROS 2 packages like Nav2 and is ideal for use in navigation, exploration, and autonomous mapping tasks in both simulated and real-world environments.

For our setup, **SLAM Toolbox** was installed from source inside our **ROS 2** workspace, **scout\_ws**. This approach provides flexibility for development and ensures compatibility with our customized environment. To install SLAM Toolbox from source, we followed these steps:

* cd ~/scout\_ws/src
* git clone https://github.com/SteveMacenski/slam\_toolbox.git -b humble
* cd ~/scout\_ws
* rosdep install --from-paths src --ignore-src -r -y
* colcon build --symlink-install
* source install/setup.bash

After building, SLAM Toolbox is available within our workspace and can be launched and integrated with other nodes in the navigation stack as part of our custom configuration.

We use the following parameter files:

* Livox only:
  + **mapper\_params\_online\_async.yaml**
* Fused Data:
  + **mapper\_params\_online\_async\_fused.yaml**

### Nav2

**Nav2 (Navigation 2)** is the official navigation stack for ROS 2, enabling mobile robots to autonomously move through their environment. It provides a complete set of tools and behaviors for navigation, including global and local path planning, obstacle avoidance, recovery behaviors, and goal handling. Nav2 works by consuming data from sensors such as LiDAR and IMUs, using that input to build maps (in conjunction with SLAM tools), plan safe paths, and control the robot’s motion in real time. Highly modular and extensible, Nav2 integrates tightly with other ROS 2 components and supports both simulation (e.g., Gazebo) and real-world deployment, making it a critical component in autonomous robotic systems.

In our configuration, Nav2 was installed from source into our ROS 2 workspace, scout\_ws, to allow for customization, better debugging, and access to the latest features and bug fixes. This approach gives us full control over the build process and component versions. We installed Nav2 by cloning the official repositories and their dependencies into the workspace and building them using colcon. The steps were:

* cd ~/scout\_ws/src
* git clone https://github.com/ros-planning/navigation2.git -b humble
* vcs import < navigation2/tools/repo\_files/ros2.repos
* cd ~/scout\_ws
* rosdep install --from-paths src --ignore-src -r -y
* colcon build --symlink-install
* source install/setup.bash

Once built and sourced, all core Nav2 components—including planners, controllers, behavior trees, and navigation utilities—are available within the workspace. This setup enables seamless integration with SLAM Toolbox and other ROS 2 nodes, supporting autonomous navigation in both simulation and real-world deployments.

We use the following parameter files:

* Livox:
  + **nav2\_Livox\_params.yaml**

### Collision Monitor

The Collision Monitor is a safety node in the Nav2 stack that listens to sensor data (e.g., LiDAR, LaserScan, or point clouds) and monitors for nearby obstacles in real time. If a potential collision is detected within a defined region around the robot, it can take automatic action, such as stopping or reducing speed. This is especially useful in dynamic environments where unforeseen objects may suddenly appear in the robot’s path. The Collision Monitor is part of the official Nav2 stack and is included when you install navigation2 via the ROS 2 APT package.

We use the parameter file **collision\_monitor\_params.yaml**

### Explore Lite

**Explore Lite** is a lightweight ROS 2 package that enables **frontier-based exploration**, allowing a robot to autonomously map unknown environments without prior knowledge of the layout. Frontier exploration works by identifying the boundaries or **"frontiers"** between known and unknown areas on a map generated by SLAM. The robot continuously selects and navigates to these frontiers, gradually expanding its understanding of the environment. Explore Lite operates on top of SLAM Toolbox and Nav2, issuing navigation goals as new frontiers are discovered. It’s especially useful for autonomous mapping tasks in complex indoor environments and is commonly used in both simulation and real-world robotic applications.

For our implementation, **Explore Lite was installed from source**, as it is not available through the default ROS 2 APT repositories. To install it, we first cloned the package into our workspace source directory using:

* cd ~/scout\_ws/src
* git clone https://github.com/SteveMacenski/explore\_lite.git -b humble

Next, we built the workspace using:

* cd ~/scout\_ws
* colcon build

After building, we sourced the workspace to make the package available:

* source install/setup.bash

## Data Collection Using Livox HAP LiDAR

### Overview

Data collection in this context refers to capturing 3D environmental information using the Livox HAP LiDAR sensor. This process is essential for tasks like mapping, perception testing, and validating SLAM and navigation algorithms. By recording LiDAR data in real-time, developers can analyze sensor performance, visualize environments, and generate accurate point cloud representations for further processing.

Steps:

1. Launch Livox HAP LiDAR
2. Record Rosbag
3. Generate PCD Files
4. Play and review Recording (Optional)

### Launch Livox HAP LiDAR

The first step is to power on the LiDAR and run the Livox driver. This allows ROS 2 to begin receiving real-time data from the sensor.Command:

* ros2 launch livox\_ros\_driver2 rviz\_HAP\_launch.py

This command starts the Livox driver and launches RViz with a predefined configuration to visualize LiDAR data.

### Record Rosbag

A **rosbag** is a file format used by ROS 2 to log and store message data published on ROS topics. It allows you to record sensor data and replay it later for analysis, testing, or visualization.

Steps:

* Start the recording:
  + ros2 bag record -o <file\_name> -a
* -a records **all topics** being published at the time
* -t /cmd\_vel records only velocity command topics
* -t /livox/lidar records only LiDAR point cloud data, which is typically the topic of interest for mapping

### Generate PCD Files

A **PCD (Point Cloud Data) file** is a standardized file format used to store 3D point cloud data. We generate PCD files from recorded rosbags to allow visualization, editing, or processing in external tools like CloudCompare or PCL (Point Cloud Library).

Steps:

* Navigate to the configuration directory:
  + cd /home/scoutmini2/ws\_rosbag/src/ros2\_bag\_exporter/config
* Open exporter\_config.yaml
* Edit bag\_path to reflect rosbag’s saved <file\_name>
* Save .yaml
* In a new terminal, build the exporter:
  + colcon build
  + source install/setup.bash
* To run conversion tool:
  + ros2 run ros2\_bag\_exporter bag\_exporter

### Play and Review Recording (Optional)

You can replay the recorded rosbag to visualize the data in RViz and ensure proper data collection.

Steps:

* Play the recording:
  + ros2 bag play <file\_name>
* In rviz, add a new display:
  + Set **Display Type** to PointCloud2
  + Set the topic to /livox/lidar
* Make sure the **Fixed Frame** is set to livox\_frame
* Ensure the playback speed is greater than zero for data to appear

## Scout Mini Autonomous Navigation (Livox & Real Space)

### Overview

Full autonomous navigation in our project involves integrating perception, localization, and motion planning into a unified system that enables the Scout Mini to navigate an unknown environment without human intervention. Using a combination of the Livox HAP LiDAR for 3D sensing, SLAM Toolbox for mapping and localization, and Nav2 for path planning and control, the robot can explore, build a map, and return to previously visited locations. The following steps outline the complete initialization process required to enable autonomous navigation.

Steps:

1. Launch Livox HAP LiDAR
2. Open Can Port for Scout Base Communication
3. Activate and Connect to Scout Base
4. Launch Scout Mini Description
5. Lauch Point Cloud to Laser Scan converter
6. Run SLAM Toolbox
7. Launch Nav2
8. Run Collision Monitor
9. Use Rviz2 to Visualize Mapping

### Shortcut commands:

We have shortcut commands for each of these process that each call a custom launch file that runs these in sequence:

* Slam Livox:
  + Slam
* Slam fused data:
  + SlamALL
* Nav2 Livox:
  + Nav2
* Nav2 fused data:
  + Nav2ALL
* Nav2 Map Livox:
  + Nav2Map
* Nav2 Map fused data:
  + Nav2MalALL

Each of these commands runs a different version of this same process but launching additional necessary packages for each depending on desired performance and sensor usage.

### Launch Livox HAP LiDAR

The first step is to start the Livox HAP LiDAR using livox\_ros\_driver2. This ROS 2 node publishes 3D point cloud data to the /livox/lidar topic, providing real-time perception of the surrounding environment. This point cloud data forms the foundation for both SLAM and obstacle avoidance, allowing the robot to “see” and understand its surroundings.

Steps:

* Run the command:
  + ros2 launch livox\_ros\_driver2 viz\_HAP\_launch.py
* Minimize the Rviz tab, it can be used for point cloud visualization any time

### Open Can Port for Scout Base Communication

Before the robot can move, the CAN (Controller Area Network) port must be initialized. This step configures the hardware interface that allows the onboard computer (Intel NUC) to communicate with the Scout Mini’s motor controllers. Opening the CAN port enables ROS 2 to send drive commands and receive odometry data from the robot base.

Steps:

* Run the command:
  + sudo ip link set can0 up type can bitrate 500000
* Input the sudo password: newmini2

### Activate and Connect Scout Base Communication

Once the CAN port is open, we activate and connect the Scout base using ROS 2 nodes or utilities provided in the ugv\_sdk. CAN is used as a robust and reliable communication protocol between the robot base and the onboard PC, handling messages related to velocity control, feedback from wheel encoders, and system diagnostics. This step ensures two-way communication is active and responsive.

Steps:

* Run the command:
  + ros2 launch scout\_base scout\_base.launch.py

### Launch Scout Mini Description

The Scout Mini’s **description package** is launched to load its URDF (Unified Robot Description Format) model into the ROS 2 environment. This includes details about the robot’s geometry, frames, sensors, and joints, which are necessary for visualization in RViz, sensor alignment, and frame transformations (TF). This step also ensures that SLAM, Nav2, and RViz are aware of the robot’s physical configuration.

Steps:

* Run the command:
  + ros2 launch scout\_description scout\_base\_description.launch.py

### Launch Point Cloud to Laser Scan Converter

To make the 3D LiDAR data compatible with 2D SLAM algorithms, we run the **PointCloud to LaserScan** converter. This node slices the point cloud horizontally and publishes a simulated 2D laser scan on the /scan topic. SLAM Toolbox and certain navigation plugins rely on LaserScan messages for localization and mapping, so this conversion is a critical compatibility step.

Steps:

* Run the command:
  + ros2 launch pointcloud\_to\_laserscan sample\_pointcloud\_to\_laserscan\_launch.py

### Run SLAM Toolbox

With sensor data and robot geometry available, we start **SLAM Toolbox**, which performs real-time Simultaneous Localization and Mapping. It uses the /scan topic to build a 2D occupancy map while also tracking the robot’s position within that map. This is essential for both exploration and navigation, allowing the robot to understand where it is and where it has been.

Steps:

* Run the command:
  + Ros2 launch slam\_toolbox online\_async\_launch.py use\_sim\_time:=false

### Launch Nav2

We then launch the **Nav2 (Navigation 2)** stack, which provides the robot with path planning, obstacle avoidance, and goal management capabilities. Nav2 uses the live map from SLAM Toolbox and odometry from the base to compute paths to goal positions, issue movement commands, and recover from navigation failures. It enables the robot to operate fully autonomously within the mapped environment.

Steps:

* Run the command:
  + ros2 launch nav2\_bringup navigation\_launch.py use\_sim\_time:=false

### Run Collision monitor

By running the Collision Monitor, we enhance the overall safety and robustness of the robot’s navigation behavior during autonomous operation.

Steps:

* Run the command:
  + ros2 launch nav2\_collision\_monitor collision\_monitor\_node.launch.py

### Use Rviz2 to Visualize Mapping

Finally, we use **RViz2**, a 3D visualization tool, to monitor the robot’s state, sensor data, and environment in real time. RViz displays the point cloud, 2D map, robot model, and active paths, helping us verify system performance and debug any issues. Viewing the SLAM and Nav2 outputs in RViz confirms successful integration and operation of the full autonomous navigation stack.

Steps:

* Run the command:
  + rviz2 -d /opr/ros/humble/share/nav2\_bringup/rviz/nav2\_default\_view.rviz

### Additional Information:

We have also created a bash file to run all of these with a single command.

Steps:

* Run the command:
  + ./Nav2\_Bash.sh
* Then input the sudo password: newmini2

## Create, Save, and Navigate with a Custom Map

### Overview

Using SLAM (Simultaneous Localization and Mapping), we can create a 2D map of the robot’s environment either manually (teleop) or autonomously (e.g., explore mode). Once the map is built, it can be saved and reused for future localization and navigation, avoiding the need to remap each session.

Steps:

1. Initialize the Robot
2. Start SLAM
3. Build and Save the Map
4. Load the Map and Navigate

### Initialize the Robot

Start all essential robot components needed for sensing and control. This includes the base controller, sensor drivers (e.g., LiDAR), and robot URDF description. These components must be running before launching SLAM or navigation.

Steps:

* Run the basic launch commands for the robot, starting:
  + LiDAR
  + Robot Base
  + Robot Description
  + Point cloud to Laser scan

### Start SLAM

Launch the SLAM Toolbox in online mode to begin building a map in real-time as the robot moves through its environment.

Steps:

* Run the command
  + ros2 launch slam\_toolbox online\_async\_launch.py use\_sim\_time:=false
  + or use the bash script ./SLAM\_Bash.sh

### Build and Save the Map

Once the desired area has been mapped, save the map to disk. This creates two files: a .pgm (image) and .yaml (metadata) that define the occupancy grid used in navigation.

Steps:

* Run the command:
  + ros2 run map\_server map\_saver\_cli -f <file\_name>

### Load the Map and Navigate

Instead of recreating a map, load an existing one and localize the robot within it. After localization, Nav2 can be used to plan and execute paths in the known map.

Steps:

* Make sure the robot is running and initialized but not SLAM
* Run the commands in order:
  + ros2 launch nav2\_bringup localization\_launch.py use\_sim\_time:=false map:=<map\_file\_path.yaml>
    - Make sure the map topic has property: Transient local
  + ros2 launch nav2\_bringup navigation\_launch.py use\_sim\_time:=false map\_subscribe\_transient\_local:=true
  + rviz2 -d /opt/ros/humble/share/nav2\_bringup/rviz/nav2\_default\_view
* Initialize the current position
* Begin navigation

## Multi-Sensor Point Cloud Data Fuser (Custom Node)

To enrich our robot’s perception pipeline, we developed a **custom ROS 2 node** that fuses point cloud data from two different sources: the RGB-D camera and the Livox LiDAR. These sensors complement each other—RGB-D provides dense, color-rich short-range data, while LiDAR contributes sparse but far-reaching points. By combining them in a unified frame (base\_link), we produce a more complete and usable environmental point cloud for downstream processing such as visualization, mapping, and navigation.

This fusion improves SLAM performance and enhances situational awareness by giving a more robust and noise-filtered view of the world, suitable for both indoor and outdoor operation.

### Implementation Details

This node is written in Python and lives in the cloud\_fuser package inside the scout\_ws/src/ directory. Key features include:

* **Input**: Subscribes to /filtered\_depth\_cloud (RGB-D) and /livox/lidar (LiDAR)
* **TF Integration**: Uses the TF2 system to transform each point cloud into a shared frame (base\_link)
* **Filtering**: Applies a passthrough filter in the Z-direction (configurable via parameters) to reduce noise
* **Downsampling**: Uses a voxel grid filter to reduce redundant points and data size
* **Output**: Publishes the merged cloud on /fused\_points as a sensor\_msgs/PointCloud2 message

### Parameters

The node supports the following runtime parameters:

* z\_min: Minimum Z value for passthrough filter (default: 0.05 m)
* z\_max: Maximum Z value for passthrough filter (default: 0.4 m)
* voxel\_size: Size of voxels for downsampling (default: 0.05 m)

### Installation and Launch

The node is included in the scout\_ws workspace. To build it:

* cd ~/scout\_ws
* colcon build --symlink-install
* source install/setup.bash

To launch the node, we use a custom launch file located in

* cloud\_fuser/launch/cloud\_fuser.launch.py.

A typical launch command looks like:

* ros2 launch cloud\_fuser cloud\_fuser.launch.py

The launch file initializes the node with its parameters and ensures the correct topic remappings and TF availability.

### Runtime Behavior

At runtime:

1. The node synchronizes the two input point clouds using an **ApproximateTimeSynchronizer**.
2. Each cloud is transformed into the base\_link frame using tf2\_ros and tf2\_sensor\_msgs.
3. Points outside the specified Z-range are discarded.
4. The remaining points are voxel-filtered to remove redundant data.
5. The resulting fused cloud is published on /fused\_points.

This topic is available for visualization in RViz and for consumption by other ROS nodes. In our workflow, it can be visualized live or logged for post-processing.

### Example Use Case

In SLAM and mapping scenarios, especially in cluttered environments, having both near-field RGB-D data and far-field LiDAR data fused into a single cloud enhances robustness and map quality. This fused output is particularly valuable when operating in environments with varied structure—such as transitioning between rooms and open hallways.

# TurtleBot3 in GAZEBO Simulator

**TurtleBot3** is a widely used open-source robot platform that includes both physical and simulated versions for robotics research and education. It comes with full ROS 2 support and is natively integrated with the **Gazebo Simulator**, a physics-based simulation environment that models real-world dynamics such as gravity, friction, and collisions. In our setup, we use a **ROS 2-bridged version of Gazebo**, specifically **Gazebo Fortress**, to ensure compatibility with **ROS 2 Humble Hawksbill**. This configuration allows us to test SLAM, navigation, and sensor integration in a controlled virtual environment before deploying to real hardware. TurtleBot3 serves as a useful reference and comparison point for validating our Scout Mini simulation.Scout Mini in GAZEBO Simulator

## Software Composition

### TurtleBot3

The **TurtleBot3** platform offers three distinct robot models for simulation; **Burger**, **Waffle**, and **Waffle Pi**. Each varying in size, sensor configuration, and computational capabilities. The **Burger** model, which we primarily used, is the most compact and lightweight version, making it ideal for basic SLAM and navigation testing in simulation. TurtleBot3 also comes with a set of preconfigured **Gazebo worlds**, including a simple empty environment, a house-like indoor environment, and a maze world, which are designed to test mapping, path planning, and obstacle avoidance in progressively complex scenarios.

These robot models are tightly integrated with ROS 2, and the simulation packages include full URDF descriptions, controller plugins, and sensor emulators (e.g., LiDAR, IMU, and wheel encoders). The simulated robot publishes the same topics and transforms you would expect from a real robot, which makes it an excellent tool for prototyping and validating navigation stacks like SLAM Toolbox and Nav2.

In our setup, we used **Gazebo Fortress** alongside **ROS 2 Humble** for maximum compatibility, taking advantage of the turtlebot3\_gazebo and turtlebot3\_simulations packages. This allowed us to run end-to-end navigation pipelines in a virtual environment before transitioning to the real Scout Mini hardware, ensuring our software components were properly configured and operational in a risk-free setting.

For our setup, **TurtleBot3 was installed from source** to ensure full compatibility with **ROS 2 Humble** and **Gazebo Fortress**. We began by creating a dedicated workspace using:

* mkdir -p ~/turtlebot3\_ws/src
* cd ~/turtlebot3\_ws/src

We then cloned the necessary repositories from the official **ROBOTIS GitHub**, making sure to check out the humble-devel branches:

* git clone -b humble-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
* git clone -b humble-devel <https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git>

After cloning, we installed dependencies with:

* cd ~/turtlebot3\_ws
* rosdep install --from-paths src --ignore-src -r -y

We built the workspace using:

* colcon build

Once the build was successful, we sourced the workspace to overlay the environment:

* source install/setup.bash

This process made the TurtleBot3 robot models and simulation environments available in ROS 2, enabling us to launch and test SLAM and navigation pipelines in Gazebo before deploying them on the Scout Mini.

### GAZEBO Simulator

**Gazebo Fortress** is a modern, open-source 3D robotics simulator that provides a highly realistic physics environment for testing and validating robot behavior in both indoor and outdoor scenarios. It simulates dynamics such as gravity, collisions, sensor noise, and actuation delays, making it an essential tool for safely developing and testing robotic systems before deploying them on real hardware.

Gazebo Fortress is part of the **Ignition Gazebo** family (now branded as **Gazebo Sim**) and is officially supported in ROS 2 Humble. It offers improved performance and modularity over older versions like Gazebo Classic. It integrates with ROS 2 through ros\_gz (formerly ros\_ign), which bridges topics, services, and transforms between Gazebo and ROS 2.

We installed **Gazebo Fortress** through the official binary packages to ensure compatibility with ROS 2 Humble and modern simulation features. First, we added the OSRF package repository by running:

* sudo sh -c 'echo "deb [arch=amd64] http://packages.osrfoundation.org/gazebo/ubuntu stable main" > /etc/apt/sources.list.d/gazebo-stable.list'

Next, we added the OSRF public key to authenticate packages:

* curl -sSL http://packages.osrfoundation.org/gazebo.key | sudo apt-key add –

After updating our package lists with sudo apt update, we installed Gazebo Fortress along with its ROS 2 integration bridge by running:

* sudo apt install ros-humble-ros-gz

This command installs the ros\_gz bridge, simulation plugins, and Gazebo Fortress itself. Once installed, Gazebo can be launched and used with TurtleBot3 or other ROS 2-based robots for high-fidelity, physics-based simulation environments.

## GAZEBO Simulation Test

### Overview

**GAZEBO** includes a tutorial launch file that demonstrates a basic simulation environment with a simple robot model. This can be used to verify the installation and understand how ROS 2 nodes interface with the simulator.

Steps:

1. Launch Simulation
2. Check Ros2 Topics
3. Configure Ros2
4. Create bridge for a Topic
5. Use Teleop Keyboard Control
6. Visualize LiDAR

### Launch Simulation

Start GAZEBO Fortress using a tutorial launch file. This initializes the simulator, loads a sample .sdf world, and enables ROS 2 integration.

Steps:

* Run the command:
  + ign gazebo -v 4 -r visualize\_lidar.sdf

### Check Ros2 Topics

Check that Gazebo and ROS 2 are communicating by listing topics in both environments.

Steps:

* Run the commands:
  + ign topic -l
  + ros2 topic list

### Configure Ros2

Install and prepare the ROS 2 <-> Gazebo bridge so the systems can exchange messages.

Steps:

* Run the command:
  + sudo apt-get install ros-humble-ros-ign-bridge

### Create Bridge for a Topic

Use the ros\_gz\_bridge to create a bridge between Gazebo and ROS 2 topics (e.g., velocity commands).

Steps:

* Run the commands:
  + source /opt/ros/humble/setup.bash
  + ros2 run ros\_gz\_bridge parameter\_bridge /model/vehicle\_blue/cmd\_vel@geometry\_msgs/msg/Twist]ignition.msgs.Twist

### Use Teleop Keyboard Control

Send keyboard commands to control the robot using the bridged /cmd\_vel topic.

Steps:

* Run the commands:
  + source /opt/ros/humble/setup.bash
  + ros2 run teleop\_twist\_keyboard teleop\_twist\_keyboardd –ros-args -r /cmd\_vel:=/model/vehicle\_blue/cmd\_vel

### Visualize LiDAR

Visualize simulated LiDAR data in RViz2 to confirm proper sensor bridging and ROS topic publishing.

Steps:

* Run the commands:
  + source /opt/ros/humble/setup.bash
  + ros2 run ros\_gz\_bridge parameter\_bridge [/lidar2@sensor\_msgs/msg/LaserScan[ignition.msgs.LaserScan](mailto:/lidar2@sensor_msgs/msg/LaserScan%5bignition.msgs.LaserScan) –ros-orgs -r /lidar2:=/laser\_scan
  + source /opt/ros/humble/setup.bash
  + rviz2
* Configure rviz by setting Fixed\_Frame to:
  + vehicle\_blue/lidar\_link/gpu\_lidar
* Add laserscan topic

## TurtleBot3 GAZEBO SLAM

### Overview

We can simulate the TurtleBot3 robot in a Gazebo environment to test SLAM and autonomous navigation without using physical hardware. This process allows us to map an unknown environment, save the resulting occupancy grid, and later use that map for navigation with the Nav2 stack

Steps:

1. Launch Robot and Environment
2. Start Keyboard Control
3. Run SLAM Toolbox
4. Save a Map
5. Run Nav2 on a Saved Map

### Launch Robot and Environment

This step starts the Gazebo simulator with the TurtleBot3 robot spawned into a virtual world. It ensures that the robot model, environment, and simulation time are properly initialized for later steps.

Steps:

* Run the command:
  + export TURTLEBOT3\_MODEL = burger
  + ros2 launch turtlebot3\_gazebo turtlebot3\_world.launch.py

### Start Keyboard Control

Keyboard teleoperation allows us to manually drive the robot within the simulation. This movement is necessary for SLAM to collect enough sensor data and generate a useful map.

Steps:

* Run the command:
  + ros2 run turtlebot3\_teleop teleop\_keyboard

### Run SLAM Toolbox

This launches the SLAM Toolbox in asynchronous online mode. It listens to the robot’s LiDAR and odometry data to build a 2D occupancy map in real time as the robot moves through the simulated environment.

Steps:

* Run the command:
  + ros2 launch slam\_toolbox online\_async\_launch.py use\_sim\_time:=true

### Save a Map

Once the map is sufficiently built, we save the occupancy grid and associated metadata to disk. This creates a .pgm and .yaml file pair that can later be used to initialize a static map for navigation.

Steps:

* Run the command:
  + ros2 run nav2\_map\_server map\_saver\_cli -f ~/turtlebot3\_ws/map

### Run Nav2 on a Saved Map

In this step, we use the saved map as input to the Nav2 stack. The robot localizes itself on the static map and can now perform autonomous navigation tasks like path planning and goal execution in the simulated environment.

Steps:

* Run the command:
  + ros2 launch turtlebot3\_navigation2 navigation2.launch.py use\_sim\_time:=true map:=$HOME/turtlebot3\_ws/map.yaml

# Scout Mini in GAZEBO Simulator

Running the Scout Mini in the GAZEBO simulator allows us to test navigation, mapping, perception, and control algorithms in a virtual environment that mimics real-world physics and sensor feedback. Even if we have a physical robot, simulation enables us to try out new code, experiment with risky maneuvers, and test in varied environments without hardware wear or safety risks.

Simulated testing also allows faster iteration by removing real-world constraints like battery life, sensor noise, and physical damage. Unlike other generic simulated robots, this setup uses the actual URDF and sensor configuration of our real Scout Mini—making results more transferable to the real platform. Compared to running on real hardware, simulation allows full introspection of all components, easier debugging, and controlled repeatability.

## Software Composition

### Scout Mini Simulation Description

The Scout Mini simulator mirrors the real-world robot in form, dimensions, sensor placement, and ROS 2 interface, making it ideal for testing software in a risk-free virtual environment. This ensures that algorithms developed in simulation—such as SLAM, navigation, and object avoidance—can be transferred to the physical robot with minimal adjustments.

To isolate the Scout Mini simulation and related packages, create a new ROS 2 workspace:

* mkdir -p ~/scout\_gazebo\_ws/src
* cd ~/scout\_gazebo\_ws/src

To run the Scout Mini in simulation under ROS 2 Humble, install the ugv\_gazebo\_sim package from the official GitHub repository. Be sure to use the humble branch to ensure compatibility:

* git clone -b humble <https://github.com/agilexrobotics/ugv_gazebo_sim.git>

After cloning, build the workspace with:

* colcon build

Source it before launching simulation environments.

* source install/setup.bash

This simulated setup allows safe experimentation with navigation stacks, behavior trees, sensor integration, and system debugging before deploying on hardware.

Something important to note here is that we already have a version of scout\_description in scout\_ws, so these commands will conflict. We changed all versions of the package scout\_description inside

## Running the Scout Mini in an Empty GAZEBO Environment

### Overview

This workflow runs the Scout Mini robot inside a minimal, empty GAZEBO world, ideal for testing core robot behaviors without environmental distractions. This setup allows us to validate SLAM, basic navigation, and keyboard teleoperation in simulation before introducing more complex world geometry or mission logic.  
Steps:

1. Launch Scout Mini in GAZEBO
2. Enable Keyboard Control
3. Run SLAM Toolbox
4. Launch Nav2
5. Visualize in Rviz2

### Launch Scout Mini in GAZEBO

This step launches the Scout Mini robot into an empty GAZEBO environment with simulated time enabled. It loads the URDF model, initializes robot controllers, and publishes the necessary transform frames and sensor topics.

Steps:

* Run the command:
  + ros2 launch scout\_gazebo\_sim scout\_mini.launch.py use\_sim\_time:=true

### Enable Keyboard Control

To manually drive the simulated robot, you can use a keyboard teleoperation node. This node publishes velocity commands (/cmd\_vel) to control the robot's movement using keyboard arrow keys or WASD input.

Steps:

* Run the command:
  + ros2 run teleop\_twist\_keyboard teleop\_twist\_keyboard

### Run SLAM Toolbox

The SLAM Toolbox node uses laser scan data to build a 2D map in real time as the robot moves. This version runs sync\_slam\_toolbox\_node, which is ideal for online SLAM in simulation environments.

Steps:

* Run the command:
  + ros2 run slam\_toolbox sync\_slam\_toolbox\_node --ros-args -p odom\_frame:=odom -p base\_frame:=base\_link -p map\_frame:=map -p scan\_topic:=/scan -p map\_update\_interval:=1.0 -p max\_laser\_range:=5.0 -p minimum\_travel\_distance:=0.1 -p minimum\_travel\_heading:=1.57 -p use\_scan\_matching:=true -p do\_loop\_closing:=true -p use\_sim\_time:=true
* NOTE: This command explicitly defines all required parameters for the SLAM node. When not using a launch file, ROS 2 parameters must be passed via --ros-args to ensure correct frame names, timing behavior, and SLAM tuning. If these parameters are not set, SLAM may fail silently or behave incorrectly.

### Launch Nav2

This step launches the full Nav2 stack, enabling path planning, behavior trees, map servers, and control. It requires a map and TF transforms to function properly—so be sure SLAM Toolbox or a static map is publishing before launching.

Steps:

* Run this command:
  + Ros2 launch nav2\_bringup navigation\_launch.py use\_sim\_time:=true

### Visualize in Rviz2

RViz2 provides a graphical interface to monitor the robot's state, visualize the SLAM-generated map, and interact with the Nav2 stack (e.g., setting goals). The default Nav2 RViz config already includes the map, robot model, laser scan, and global/local plans.

Steps:

* Run the command:
  + Rviz2 -d /opt/ros/humble/share/nav2\_bringup/rviz/nav2\_default\_view.rviz

# Appendix

## Creating a Custom Bash Script

To simplify launching multiple ROS 2 nodes and configurations for the Scout Mini simulation or real-world operation, we can create a custom Bash script. This script automates the sequence of commands needed to start components like LiDAR drivers, SLAM Toolbox, Nav2, and RViz, reducing setup time and minimizing errors due to forgotten steps. A Bash script is especially useful when deploying in the field or when working with users unfamiliar with the full ROS 2 workflow.

Steps:

1. **Create the Script File**  
   Navigate to your desired directory and create a new file:

* touch bash\_name.sh

1. **Edit the Script**  
   Use a text editor like nano or vim to add the commands:

* nano bash\_name.sh

1. Make the Script Executable

* chmod +x bash\_name.sh

1. Run the Script

* ./bash\_name.sh

## Creating a Custom Python Launch File

To manage the execution of our custom ROS 2 nodes, we created a **Python-based launch file** using the launch and launch\_ros libraries. Unlike XML-based launch files used in ROS 1, Python launch files in ROS 2 offer more flexibility and logic control, allowing us to set parameters, remap topics, and structure complex launch behaviors programmatically.

In our case, we created a custom launch file for the **Multi-Sensor Point Cloud Data Fuser** node inside the cloud\_fuser/launch/ directory. This launch file:

* Instantiates the pointcloud\_fuser node
* Sets runtime parameters like z\_min, z\_max, and voxel\_size
* Defines output topic names and frame settings
* Ensures all required transforms are available at startup

This file allows us to bring up the node with a single command, making it easy to test, iterate, and integrate the fuser into larger system launch files.

A typical launch command looks like:

* ros2 launch cloud\_fuser cloud\_fuser.launch.py

This modular and scriptable structure makes it easier to maintain consistency across deployments, both in simulation and on physical robots.