Share



**You said:**

I would like to do a complete setup on the laptop ubuntu 22.04 lts and try the obstacle avoidance code here. It ought to be less cumbersome to install and try code.

**ChatGPT said:**

ChatGPT

Memory updated

That sounds like a great plan! Setting up on a laptop with Ubuntu 22.04 LTS should streamline the process. Let me know if you need help with any specific installations or configurations for the obstacle avoidance system.

4o

**You said:**

yes, you will have to guide me through the entire installation of environment, using a common github repository for four installations - bullseye on pi 4b, bookworm on pi 4b, ubuntu 22.04 lts on pi4b, and ubuntu 22.04 lts on laptop. The others have already been tried upto a point. Now, it is the laptop's turn. Whichever proves to work smoothest will continue for now. When some performance is obtained with ROS2 on the winner, next steps can be planned.

**ChatGPT said:**

ChatGPT

Memory updated

Got it! We can start by setting up the environment on the Ubuntu 22.04 LTS laptop. Here's the plan:

**Step 1: Prepare the System**

1. **Update and Upgrade**: Ensure your system is up-to-date.

bash

Copy code

sudo apt update && sudo apt upgrade -y

1. **Install Git**: If not already installed.

bash

Copy code

sudo apt install git -y

**Step 2: Set Up the Development Environment**

1. **Install ROS 2 (Humble)**:
   * Follow the official guide for Humble on Ubuntu 22.04.
   * Install additional packages like colcon for building workspaces.
2. **Python and Dependencies**:
   * Ensure Python 3.10+ is installed (default in Ubuntu 22.04).
   * Install necessary Python libraries:

bash

Copy code

sudo apt install python3-pip -y

pip install --user numpy matplotlib opencv-python

1. **Docker (Optional)**:
   * If you plan to use Docker for portability, install Docker and set it up for non-root usage.

bash

Copy code

sudo apt install docker.io -y

sudo usermod -aG docker $USER

**Step 3: Set Up the GitHub Repository**

1. **Create/Clone Repository**:
   * Create a new repository or clone an existing one for your obstacle avoidance system.

bash

Copy code

git clone <your-repo-url>

cd <your-repo-directory>

1. **Organize the Repository**:
   * Structure the repository to support multi-environment configurations:

css

Copy code

obstacle\_avoidance/

├── src/

├── configs/

├── scripts/

├── docker/

├── README.md

**Step 4: Install Hardware-Specific Drivers**

1. **Camera Drivers**:
   * Install libcamera for Raspberry Pi cameras if using Pi-specific hardware.

bash

Copy code

sudo apt install libcamera-apps -y

1. **LIDAR Drivers**:
   * Identify your LIDAR model and install required drivers or libraries.
   * Example for RPLIDAR:

bash

Copy code

sudo apt install ros-humble-rplidar-ros

**Step 5: Test Basic Functionality**

1. **Clone and Run Existing Code**:
   * Test LIDAR and camera functionality.
   * Validate system performance with existing scripts.
2. **Push Updates to Repository**:
   * Commit and push any tested changes to the shared repository.

Let me know where you'd like to start, or if you'd like detailed commands for any specific step!

4o

**You said:**

i need an example for setting up github repository

**ChatGPT said:**

ChatGPT

Here's a step-by-step example of setting up a GitHub repository for your obstacle avoidance project:

**Step-by-Step Setup for GitHub Repository**

**Step 1: Create a New GitHub Repository**

1. **Log in to GitHub**:
   * Visit [GitHub](https://github.com) and log in with your credentials.
2. **Create a New Repository**:
   * Click on the '+' icon in the top right corner and select **New repository**.
3. **Fill Repository Details**:
   * **Repository name**: obstacle\_avoidance
   * **Description**: Project for obstacle avoidance using ROS 2 and sensors.
   * **Public/Private**: Select based on your preference (Public recommended for collaboration).
4. **Initialize Repository**:
   * Click **Create repository**.

**Step 2: Clone the Repository Locally**

1. **Clone the Repository**:

bash

Copy code

git clone https://github.com/your-username/obstacle\_avoidance.git

cd obstacle\_avoidance

**Step 3: Organize the Repository Structure**

1. **Directory Structure**:

plaintext

Copy code

obstacle\_avoidance/

├── src/

├── configs/

├── scripts/

├── docker/

├── README.md

└── .gitignore

1. **.gitignore**:
   * Add typical ignore files to avoid committing unnecessary files:

bash

Copy code

\*.pyc

\_\_pycache\_\_/

.vscode/

.idea/

\*.log

\*.swp

**Step 4: Add Files and Commit**

1. **Add Files**:

bash

Copy code

git add .

1. **Commit Changes**:

bash

Copy code

git commit -m "Initial setup for obstacle avoidance project"

**Step 5: Push to GitHub**

1. **Push to Remote**:

bash

Copy code

git push origin main

**Step 6: Manage Code in Repository**

1. **Develop and Test**:
   * Develop your code in the respective directories (src, configs, scripts, docker).
2. **Commit Changes**:
   * Regularly commit updates:

bash

Copy code

git add .

git commit -m "Updated sensor fusion logic"

git push origin main

1. **Collaboration**:
   * Team members can fork and contribute to your repository.

Let me know if you need further assistance with setting up specific parts or debugging any issues!

4o mini

**You said:**

do i make a text file called .gitignore?

**ChatGPT said:**

ChatGPT

Yes, you need to create a text file named .gitignore in the root of your repository. This file specifies which files or directories Git should ignore when committing changes.

Here’s how you can create and add a .gitignore file:

**Method 1: Using a Text Editor**

1. Navigate to your repository folder:

bash

Copy code

cd obstacle\_avoidance

1. Open a text editor to create the .gitignore file:

bash

Copy code

nano .gitignore

1. Add the following content (example):

plaintext

Copy code

# Ignore Python compiled files

\*.pyc

\_\_pycache\_\_/

# Ignore log files

\*.log

# Ignore temporary files

\*.swp

\*.tmp

# Ignore IDE-specific directories

.vscode/

.idea/

# Ignore ROS 2 build files

build/

install/

log/

1. Save and exit (Ctrl+O, then Enter, then Ctrl+X).

**Method 2: Create with a Command**

Run the following commands to create and add basic .gitignore content:

bash

Copy code

echo "\*.pyc" >> .gitignore

echo "\_\_pycache\_\_/" >> .gitignore

echo "\*.log" >> .gitignore

echo ".vscode/" >> .gitignore

echo ".idea/" >> .gitignore

echo "build/" >> .gitignore

echo "install/" >> .gitignore

echo "log/" >> .gitignore

**Committing .gitignore to Git**

Once created, add it to your repository:

bash

Copy code

git add .gitignore

git commit -m "Added .gitignore file"

git push origin main

This ensures unwanted files are ignored in all environments using your repository. Let me know if you need help tailoring .gitignore to your project!

4o

**You said:**

emote: Support for password authentication was removed on August 13, 2021. remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication. fatal: Authentication failed for 'https://github.com/raviamur/obstacle\_avoidance.git/'

**ChatGPT said:**

ChatGPT

The error occurs because GitHub no longer supports password authentication for HTTPS. Instead, GitHub recommends using **personal access tokens (PAT)** or **SSH keys** for authentication.

Here’s how you can resolve this issue:

**Method 1: Using a Personal Access Token (PAT)**

1. **Generate a Token**:
   * Go to your GitHub account's [Personal Access Tokens settings](https://github.com/settings/tokens).
   * Click **Generate new token** (classic).
   * Select scopes such as repo for access to private repositories.
   * Click **Generate token** and copy it.
2. **Use the Token**:
   * When Git prompts for your password, enter the token instead.
3. **Update Credentials**:
   * If Git cached the old credentials, update them:

bash

Copy code

git credential-cache exit

git clone https://github.com/raviamur/obstacle\_avoidance.git

* + Enter your **username** and the **PAT** as the password.

**Method 2: Using SSH Authentication**

1. **Generate an SSH Key**:
   * Run the following command:

bash

Copy code

ssh-keygen -t ed25519 -C "your-email@example.com"

* + When prompted, press Enter to save the key in the default location (~/.ssh/id\_ed25519).

1. **Add the SSH Key to GitHub**:
   * Copy the public key:

bash

Copy code

cat ~/.ssh/id\_ed25519.pub

* + Go to [GitHub SSH Settings](https://github.com/settings/keys), click **New SSH Key**, and paste the key.

1. **Test SSH Connection**:
   * Test the connection:

bash

Copy code

ssh -T git@github.com

1. **Clone Repository**:
   * Use the SSH URL to clone:

bash

Copy code

git clone git@github.com:raviamur/obstacle\_avoidance.git

Both methods work seamlessly, but SSH is more convenient if you frequently interact with GitHub. Let me know which method you'd like to try or if you encounter any issues!

4o

**You said:**

ssh -T git@github.com git@github.com: Permission denied (publickey)

**ChatGPT said:**

ChatGPT

The "Permission denied (publickey)" error indicates that your SSH key is either not added to your SSH agent or not linked to your GitHub account. Let's resolve this step by step.

**Step 1: Check for Existing SSH Keys**

1. Run the following command to check for existing SSH keys:

bash

Copy code

ls ~/.ssh

* + Look for files like id\_ed25519 or id\_rsa.
  + If no keys are present, proceed to Step 2 to generate one.

**Step 2: Generate a New SSH Key**

1. Generate a new SSH key:

bash

Copy code

ssh-keygen -t ed25519 -C "your-email@example.com"

* + Replace your-email@example.com with your GitHub email address.
  + When prompted to save the key, press Enter to accept the default location (~/.ssh/id\_ed25519).

1. Add the SSH key to the SSH agent:

bash

Copy code

eval "$(ssh-agent -s)"

ssh-add ~/.ssh/id\_ed25519

**Step 3: Add the SSH Key to GitHub**

1. Copy the public key to your clipboard:

bash

Copy code

cat ~/.ssh/id\_ed25519.pub

1. Add the key to your GitHub account:
   * Go to [GitHub SSH Settings](https://github.com/settings/keys).
   * Click **New SSH Key**.
   * Paste the key and give it a title (e.g., "Laptop SSH Key").
   * Click **Add SSH Key**.

**Step 4: Test the SSH Connection**

1. Test the connection to GitHub:

bash

Copy code

ssh -T git@github.com

* + You should see a message like:

vbnet

Copy code

Hi <username>! You've successfully authenticated, but GitHub does not provide shell access.

**Step 5: Clone the Repository Using SSH**

Now, use the SSH URL to clone your repository:

bash

Copy code

git clone git@github.com:raviamur/obstacle\_avoidance.git

If you still face issues, let me know so we can troubleshoot further!

4o

**You said:**

Okay, that got done. I will push some files now as you suggested.

**ChatGPT said:**

ChatGPT

Memory updated

Great to hear! Let me know if you need assistance organizing the files or troubleshooting any issues during the push.

4o

**You said:**

I did the commit command. I still do not see the files in the Git repository.

**ChatGPT said:**

ChatGPT

If the files are not visible in the GitHub repository after committing, it’s possible that they haven’t been pushed to the remote repository yet. Here’s how you can troubleshoot and resolve the issue:

**Step 1: Confirm the Commit**

1. Check the commit log to ensure the files were committed:

bash

Copy code

git log

* + Look for your recent commit with the message you used (e.g., "Initial setup for obstacle avoidance").
  + If you don’t see your commit, retry the commit process:

bash

Copy code

git add .

git commit -m "Your commit message"

**Step 2: Check the Current Branch**

1. Verify which branch you’re working on:

bash

Copy code

git branch

* + The default branch is often main or master. Ensure this matches the branch on the remote repository.

**Step 3: Push the Changes**

1. Push your commits to the remote repository:

bash

Copy code

git push origin main

* + Replace main with the branch name if it’s different.

**Step 4: Verify the Files on GitHub**

1. Go to your GitHub repository in your browser.
2. Refresh the page to see if the files appear.

**Common Issues**

* **Detached HEAD State**: If you see a warning about being in a detached HEAD state, you might not be on a branch. Switch to a branch:

bash

Copy code

git checkout main

git pull origin main

* **Authentication Issues**: If prompted for a password or token, ensure you’ve set up SSH or PAT correctly.

**Step 5: Debugging Further**

If the problem persists, run the following commands and share their output:

1. Check the repository’s remote URL:

bash

Copy code

git remote -v

1. Check the status of your repository:

bash

Copy code

git status

Let me know what you find, and I’ll help you resolve it!

4o

**You said:**

This is how it went last time - git commit -m "Initial setup for obstacle avoidance project" [main 5bf4155] Initial setup for obstacle avoidance project 843 files changed, 625247 insertions(+) create mode 100644 References/Best Practices.docx create mode 100644 References/Development of Pikart-1.docx create mode 100644 References/RPLidar documentation.docx - and many more like this..

**ChatGPT said:**

ChatGPT

From the output, the commit seems successful, as Git acknowledges the files were added and committed to the local repository. However, these changes won’t appear in the GitHub repository until you **push** them to the remote repository.

Here’s how you can push your changes:

**Step 1: Push the Changes to the Remote Repository**

Run the following command:

bash

Copy code

git push origin main

* Replace main with your branch name if it’s different.

**Step 2: Verify the Push**

* After the push, you should see output indicating that changes have been uploaded to GitHub.
* Go to your GitHub repository in your browser and refresh the page to confirm the files are visible.

**Step 3: Troubleshoot If Files Still Don’t Appear**

1. **Check Remote URL**: Verify your local repository is linked to the correct GitHub repository:

bash

Copy code

git remote -v

* + Ensure the URL matches your GitHub repository's SSH or HTTPS URL.

1. **Check Push Status**: Ensure the push succeeded without errors.
2. **Reattempt the Push**: If something went wrong earlier, you can try re-pushing:

bash

Copy code

git push --force origin main

**Step 4: Double-Check Branch**

If you are working on a branch other than main, you might need to specify the branch:

bash

Copy code

git push origin <branch-name>

Let me know what happens when you run the git push command!

4o

**You said:**

Ok, that is done! Thanks

**ChatGPT said:**

ChatGPT

Memory updated

You're welcome! Glad it worked out. Let me know if you need help with anything else! 😊

4o

**You said:**

while trying to install vscode, this error came up. what is the next step? Download is performed unsandboxed as root as file '/home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb' couldn't be accessed by user '\_apt'. - pkgAcquire::Run (13: Permission denied)

**ChatGPT said:**

ChatGPT

The error occurs because the \_apt user, which is used by the package manager for secure operations, doesn't have the necessary permissions to access the .deb file. This typically happens if the file was downloaded by a user with elevated permissions (like root) or if the permissions of the file are restrictive.

Here’s how you can resolve it:

**Step 1: Check File Ownership and Permissions**

1. Verify the file's ownership and permissions:

bash

Copy code

ls -l /home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb

1. If the file is owned by root or has restrictive permissions, update it:

bash

Copy code

sudo chown ravi:ravi /home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb

sudo chmod 644 /home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb

**Step 2: Install the Package**

Try installing the package again:

bash

Copy code

sudo apt install ./home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb

**Step 3: Alternative Approach Using dpkg**

If the above still doesn't work, try using dpkg to install the .deb file:

bash

Copy code

sudo dpkg -i /home/ravi/Downloads/code\_1.96.0-1733888194\_amd64.deb

Then, resolve any missing dependencies:

bash

Copy code

sudo apt-get install -f

**Step 4: Use Snap as an Alternative**

If the .deb file installation fails repeatedly, you can install VS Code via Snap:

bash

Copy code

sudo snap install --classic code

After completing these steps, VS Code should be installed successfully. Let me know if you encounter any further issues!

4o

**You said:**

Step 3 worked.

**ChatGPT said:**

ChatGPT

Memory updated

Great! You're all set to start using VS Code. Let me know if you need help configuring it for your project. 😊

4o

**You said:**

How do I check which usb ports my lidar and external camera are connected to on the laptop usb hub?

**ChatGPT said:**

ChatGPT

To check which USB ports your LIDAR and external camera are connected to on your laptop, you can use a few commands in Linux to inspect connected devices and their associated ports.

**Method 1: Using lsusb**

1. **List USB Devices**: Run the following command to list all USB devices connected to your laptop:

bash

Copy code

lsusb

This will display a list of connected USB devices with their bus and device numbers. For example:

yaml

Copy code

Bus 002 Device 004: ID 046d:0825 Logitech, Inc. Webcam C270

Bus 002 Device 006: ID 05e3:0610 Genesys Logic, Inc. USB2.0 Hub

Bus 003 Device 003: ID 1e0e:9001 LIDAR Sensor

1. **Identify Your Devices**: Look for entries related to your LIDAR and external camera. The line that mentions your camera might look something like ID 046d:0825 Logitech, Inc. Webcam, and the LIDAR might mention a different ID depending on the model.

**Method 2: Using dmesg for Detailed Information**

1. **Use dmesg to See Kernel Logs**: You can use dmesg to get detailed information about USB device connections. After plugging in your devices, run:

bash

Copy code

dmesg | grep -i usb

This will give you output about devices as they are recognized by the system, such as:

csharp

Copy code

[12345.678901] usb 2-1.4: new high-speed USB device number 5 using ehci-pci

[12345.678902] usb 2-1.4: New USB device found, idVendor=1e0e, idProduct=9001

[12345.678903] usb 2-1.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3

1. **Look for Specific Devices**:
   * You can search for your LIDAR or camera by identifying their unique idVendor and idProduct.
   * Example output for your camera:

csharp

Copy code

[12345.678900] usb 3-1: new full-speed USB device number 4 using xhci\_hcd

[12345.678902] usb 3-1: New USB device found, idVendor=046d, idProduct=0825

* + You can match these IDs with the ones you found in lsusb.

**Method 3: Using lsusb -v for More Detail**

For more detailed information, you can run:

bash

Copy code

lsusb -v

This gives verbose output, including port numbers and device capabilities. You can use grep to filter for specific devices:

bash

Copy code

lsusb -v | grep -i "046d:0825"

**Method 4: Using usb-devices**

Another useful command is usb-devices, which provides detailed information about all connected USB devices:

bash

Copy code

usb-devices

Look for the entries that correspond to your LIDAR and camera, and it will display information like the bus and device number, along with other details.

**Method 5: udevadm**

You can also use udevadm to query the specific device:

bash

Copy code

udevadm info --name=/dev/ttyUSB0 --attribute-walk

Replace /dev/ttyUSB0 with your LIDAR device’s device file.

**Additional Tip:**

If your devices don’t show up as expected, try unplugging and re-plugging them into different USB ports while running dmesg or lsusb in real-time to monitor how the system reacts to the connection.

Let me know how it goes or if you need further clarification!

4o

**You said:**

RPlidar is installed with ROS2 but not available to my existing python code. How do I install the library so the old code works

**ChatGPT said:**

ChatGPT

If your RPLIDAR is installed and functioning with ROS 2, but your existing Python code isn't recognizing it, you likely need to install the appropriate RPLIDAR Python library (which may be rplidar or another similar package) and ensure it is accessible to your Python environment.

Here are the steps to make sure everything is set up correctly:

**Step 1: Install the rplidar Python Library**

You can install the rplidar Python library, which is commonly used to interface with the RPLIDAR sensor in Python, by running the following command:

bash

Copy code

pip install rplidar

If you're using Python 3 (which is typical for ROS 2), and pip is linked to Python 2, you might need to use pip3 instead:

bash

Copy code

pip3 install rplidar

This will install the Python package for RPLIDAR that your old Python code needs to communicate with the LIDAR sensor.

**Step 2: Ensure the Correct Python Environment**

If you're using a **virtual environment** for your project, make sure the environment is activated before installing the rplidar package. For example:

1. **Activate your virtual environment**:

bash

Copy code

source /path/to/your/venv/bin/activate

1. **Install the rplidar package** in that environment:

bash

Copy code

pip install rplidar

**Step 3: Check if ROS 2 and Python Code Can Access the Same LIDAR Port**

If you’re using ROS 2 for LIDAR and your Python code is trying to access the sensor, make sure they aren’t conflicting. Your Python code should be able to communicate with the LIDAR directly (via the appropriate serial port).

1. Find the device file for your RPLIDAR sensor by running:

bash

Copy code

ls /dev/ttyUSB\*

The device could be something like /dev/ttyUSB0 or /dev/ttyACM0.

1. If needed, configure the serial port in your Python code:

python

Copy code

from rplidar import RPLidar

lidar = RPLidar('/dev/ttyUSB0') # Adjust the port if necessary

1. Ensure that your user has permissions to access the serial port (especially if you're not running the code as root). If you encounter permission errors, add your user to the dialout group:

bash

Copy code

sudo usermod -aG dialout $USER

Then, log out and log back in, or restart your system for the change to take effect.

**Step 4: Check Python Code**

Make sure your Python code is using the correct RPLidar API. Here’s an example of how to initialize and get data from the RPLIDAR using the rplidar library:

python

Copy code

from rplidar import RPLidar

# Connect to the LIDAR on the appropriate port

lidar = RPLidar('/dev/ttyUSB0') # Adjust to your LIDAR port

# Get LIDAR information (e.g., scan data)

try:

for scan in lidar.iter\_scans():

print(scan)

except KeyboardInterrupt:

print('Shutting down')

lidar.stop()

lidar.disconnect()

**Step 5: Test Communication**

Once you've installed the necessary libraries and updated your Python code, test whether your Python code can successfully communicate with the LIDAR sensor. The iter\_scans() function will provide a continuous stream of scan data.

**Step 6: Troubleshooting (If Needed)**

* **Check ROS 2 and Python Conflicts**: If you are running ROS 2 in parallel with Python, make sure there’s no conflict over serial port access.
* **Test the LIDAR in ROS 2**: If your LIDAR works in ROS 2 but not in Python, make sure the LIDAR is still connected and recognized by the system.

Let me know if you run into any issues during this setup or need further clarification on any step!

4o

**You said:**

both my camera and lidar testing codes work with ttyUSB0. How would that be?

**ChatGPT said:**

ChatGPT

If both your **camera** and **LIDAR** are working with /dev/ttyUSB0, it's possible that either:

1. **Both devices are recognized by the same serial port**: While it's uncommon, some systems may try to assign the same device node (e.g., /dev/ttyUSB0) to multiple devices, especially if the devices are connected and recognized sequentially or have similar device attributes. However, in most cases, two devices should not share the same serial port.
2. **One device is using a virtual serial port or software interface**: Some cameras or sensors may create virtual serial interfaces (e.g., for diagnostics or configuration), and this could be using /dev/ttyUSB0 as well. But typically, the camera would use a different kind of interface (USB video class, /dev/video0), not a serial port.

**How to investigate:**

**1. Check the /dev/ directory for both devices**

* The camera typically uses /dev/video0 or similar, not /dev/ttyUSB0.
* To see if another device is using /dev/ttyUSB0, you can run:

bash

Copy code

ls -l /dev/ttyUSB0

**2. Use dmesg to see which device is using /dev/ttyUSB0**

After connecting the devices, use the dmesg command to inspect kernel logs and see which device is being assigned to /dev/ttyUSB0:

bash

Copy code

dmesg | grep tty

This will show you what device is associated with ttyUSB0 and should clarify if both devices are being assigned to it.

* For example, the output might look like:

csharp

Copy code

[12345.678901] usb 2-1.4: new high-speed USB device number 5 using ehci-pci

[12345.678902] usb 2-1.4: New USB device found, idVendor=1e0e, idProduct=9001

[12345.678903] usb 2-1.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3

[12345.678904] usb 2-1.4: A new USB device connected to the system at /dev/ttyUSB0

Or for a camera:

csharp

Copy code

[12345.678905] usb 2-1: new high-speed USB device number 6 using ehci-pci

[12345.678906] usb 2-1: New USB device found, idVendor=046d, idProduct=0825

[12345.678907] usb 2-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3

[12345.678908] usb 2-1: A new USB device connected to the system at /dev/ttyUSB0

**3. Check for Virtual or Composite Devices**

* Sometimes, a single USB hub or composite device might create multiple virtual interfaces that can appear as ttyUSB0, even though physically only one device is connected.
* To see all available USB interfaces, run:

bash

Copy code

lsusb

Then, look for entries related to your camera and LIDAR.

**4. Check for Udev Rules (optional)**

* If there are any custom udev rules on your system, they may be altering device names or mapping multiple devices to the same serial port.
* Check the udev rules directory for any custom rules:

bash

Copy code

ls /etc/udev/rules.d/

**5. Test the Devices Independently**

* **Disconnect one device**: Unplug the LIDAR or the camera and check the dmesg output again to see which device is connected to /dev/ttyUSB0.
  + If you unplug the LIDAR and your camera is still working, it suggests the camera isn't actually using ttyUSB0 but another interface like /dev/video0.

**Resolving Potential Conflicts:**

If you confirm that both the LIDAR and camera are using /dev/ttyUSB0, here's how to address it:

1. **Check Device Settings**: If one device is mistakenly configured to use a serial interface, check if it can be configured to use a different port or connection type.
2. **Create a Custom Udev Rule**: You can force the camera or LIDAR to use a specific device name. For example, you can create a udev rule that assigns a unique name to the camera or LIDAR when plugged in, based on attributes such as vendor ID or product ID.
   * Example rule for LIDAR:

bash

Copy code

SUBSYSTEM=="usb", ATTR{idVendor}=="1e0e", ATTR{idProduct}=="9001", SYMLINK+="rplidar"

1. **Use lsusb and dmesg for Debugging**: Once you've identified the conflicting devices and how they're being assigned ports, you can either update the device connection to avoid conflicts or modify your Python code to handle both.

Let me know if any of these steps help you clarify which device is connected to which port!

4o

**You said:**

dmesg | grep tty dmesg: read kernel buffer failed: Operation not permitted

**ChatGPT said:**

ChatGPT

The error message dmesg: read kernel buffer failed: Operation not permitted occurs because, starting from newer Linux kernel versions (especially on Ubuntu 22.04 and later), accessing kernel logs with dmesg requires elevated privileges. The dmesg command is restricted to root access due to security reasons.

To fix this and view the kernel logs related to tty devices, you can either:

**Option 1: Run dmesg with sudo**

Use sudo to view the dmesg logs:

bash

Copy code

sudo dmesg | grep tty

This will provide the necessary permissions to access the kernel buffer and show you the information related to connected devices, including your LIDAR and camera.

**Option 2: Check for Logs in /var/log/**

You can also view kernel logs directly in the /var/log/ directory. Here’s how to check syslog for USB device information:

bash

Copy code

grep tty /var/log/syslog

or

bash

Copy code

grep tty /var/log/kern.log

These logs will contain the same information as dmesg but without needing to run the command directly as sudo.

Once you have the output from dmesg, you should be able to identify the devices and check if your camera or LIDAR is being assigned to the same serial port (/dev/ttyUSB0). Let me know if you encounter any issues!

4o

**You said:**

The internal camera of the laptop and the Lidar are on USB0 and the external usb camera has the index 2

**ChatGPT said:**

ChatGPT

It seems like there’s a conflict where both the internal camera and the LIDAR are being assigned to /dev/ttyUSB0, while your external USB camera is using a different interface (possibly /dev/video2 or something similar). This can happen when multiple devices are connected to a USB hub, and the system assigns them similar or the same device paths for convenience, but this can lead to conflicts.

**Here’s how you can resolve this situation:**

**1. Confirm Device Paths for All Devices**

Run the following commands to list the exact device nodes for each of your devices:

* **List all video devices** (for cameras):

bash

Copy code

ls /dev/video\*

This should show all video devices (e.g., /dev/video0, /dev/video2, etc.). Check if your external camera is assigned as /dev/video2 and the internal one as /dev/video0.

* **List serial devices** (for LIDAR):

bash

Copy code

ls /dev/ttyUSB\*

This will show serial devices like /dev/ttyUSB0, which could be your LIDAR.

**2. Check Which Devices Are Using /dev/ttyUSB0**

Since both the internal camera and the LIDAR are using /dev/ttyUSB0, we need to clarify their usage. If the internal camera is mistakenly using a serial port, it should be corrected. You can verify the correct devices by running:

bash

Copy code

dmesg | grep tty

If you still get permission errors, use:

bash

Copy code

sudo dmesg | grep tty

This will show you which devices are being assigned to which port, and you can determine whether any conflicts exist.

**3. Fix the Conflicts: Create Udev Rules**

To ensure the LIDAR and the internal camera don’t conflict over /dev/ttyUSB0, you can create custom udev rules that assign unique device names based on their **vendor** and **product IDs**. This will help you control the naming and assignment.

**Example of creating custom udev rules:**

1. **Identify the device vendor and product ID**: First, use lsusb to list the devices and identify the IDs:

bash

Copy code

lsusb

Example output:

yaml

Copy code

Bus 002 Device 004: ID 046d:0825 Logitech, Inc. Webcam C270

Bus 003 Device 003: ID 1e0e:9001 RPLIDAR

In this case:

* + Camera: ID 046d:0825 (Logitech webcam)
  + LIDAR: ID 1e0e:9001 (RPLIDAR)

1. **Create a udev rule**: Edit or create a udev rule to assign a unique symbolic link to each device. For example:
   * Create a rule for LIDAR (/dev/rplidar):

bash

Copy code

sudo nano /etc/udev/rules.d/99-rplidar.rules

Add the following line:

bash

Copy code

SUBSYSTEM=="usb", ATTR{idVendor}=="1e0e", ATTR{idProduct}=="9001", SYMLINK+="rplidar"

This ensures that the LIDAR device always gets a symlink called /dev/rplidar.

* + Create a rule for the internal camera (/dev/internal\_camera):

bash

Copy code

sudo nano /etc/udev/rules.d/99-camera.rules

Add the following line:

bash

Copy code

SUBSYSTEM=="video4linux", ATTR{idVendor}=="046d", ATTR{idProduct}=="0825", SYMLINK+="internal\_camera"

1. **Reload udev rules**: After saving the udev rule files, reload the rules:

bash

Copy code

sudo udevadm control --reload-rules

1. **Verify the Changes**: After reloading the rules, unplug and reconnect your devices. Check that the LIDAR and internal camera now have different device paths:
   * LIDAR: /dev/rplidar
   * Internal camera: /dev/internal\_camera

You can also use:

bash

Copy code

ls -l /dev/rplidar

ls -l /dev/internal\_camera

**4. Update Python Code to Use Custom Device Paths**

Once you've created the udev rules and ensured that both the LIDAR and internal camera have unique device paths, you can update your Python code to reference these paths instead of /dev/ttyUSB0.

For example:

* **LIDAR**:

python

Copy code

lidar = RPLidar('/dev/rplidar')

* **Internal camera**:

python

Copy code

cap = cv2.VideoCapture('/dev/internal\_camera')

This will ensure that each device is accessed correctly and avoids any conflicts over /dev/ttyUSB0.

**5. Testing and Debugging**

* Test your Python code to ensure it can access both the LIDAR and camera correctly.
* If there are still issues, check the dmesg or syslog output again to ensure the devices are being properly detected and that no conflicts exist.

Let me know how it goes!

4o

**You said:**

lsusb Bus 001 Device 004: ID 1ea7:0066 SHARKOON Technologies GmbH [Mediatrack Edge Mini Keyboard] Bus 001 Device 008: ID 10c4:ea60 Silicon Labs CP210x UART Bridge Bus 001 Device 007: ID 045e:0770 Microsoft Corp. LifeCam VX-700 Bus 001 Device 006: ID 090c:2320 Silicon Motion, Inc. - Taiwan (formerly Feiya Technology Corp.) USB\_Flash\_Disk Bus 001 Device 005: ID 046d:0a19 Logitech, Inc. Logitech Z205 Bus 001 Device 003: ID 067b:2586 Prolific Technology, Inc. USB 2.0 Hub Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub Bus 002 Device 005: ID 0cf3:3005 Qualcomm Atheros Communications AR3011 Bluetooth Bus 002 Device 003: ID 04f2:b290 Chicony Electronics Co., Ltd Sony Visual Communication Camera Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub

**ChatGPT said:**

ChatGPT

From the output of lsusb, we can see the list of connected USB devices:

1. **Logitech Camera**:
   * ID 045e:0770 Microsoft Corp. LifeCam VX-700
   * This seems to be your **external USB camera**.
2. **Chicony Camera**:
   * ID 04f2:b290 Chicony Electronics Co., Ltd Sony Visual Communication Camera
   * This could be your **internal laptop camera**.
3. **LIDAR**:  
   It appears that the **RPLIDAR** is not listed in the output. This could indicate that the LIDAR device might not be recognized or it may be connected through a different USB port.

**Next Steps:**

1. **Verify the Device Path of the LIDAR:** You can try reconnecting the LIDAR device and checking again with dmesg or ls /dev/ttyUSB\* to confirm whether the LIDAR is being assigned to a serial port like /dev/ttyUSB0. It might be listed as a USB serial device.
2. **Check the LIDAR Device Again:** Run the following command to see if the LIDAR is connected and being recognized:

bash

Copy code

sudo dmesg | grep tty

1. **Create Udev Rules for Camera and LIDAR:** Now that we know the IDs for your devices, you can create the custom udev rules for the **external camera** and **internal camera** to avoid conflicts, as well as for the **LIDAR**.

Here’s an example of how to create the udev rules:

* + For the **external USB camera** (ID 045e:0770):

bash

Copy code

sudo nano /etc/udev/rules.d/99-external-camera.rules

Add the following rule:

bash

Copy code

SUBSYSTEM=="video4linux", ATTR{idVendor}=="045e", ATTR{idProduct}=="0770", SYMLINK+="external\_camera"

* + For the **internal camera** (ID 04f2:b290):

bash

Copy code

sudo nano /etc/udev/rules.d/99-internal-camera.rules

Add this rule:

bash

Copy code

SUBSYSTEM=="video4linux", ATTR{idVendor}=="04f2", ATTR{idProduct}=="b290", SYMLINK+="internal\_camera"

* + For the **RPLIDAR** (if it's connected via /dev/ttyUSB0 or a different port): You would need to know the exact vendor and product ID of the RPLIDAR. You can use the lsusb output to find the correct IDs. For example, if the LIDAR is connected and recognized, the rule might look like:

bash

Copy code

SUBSYSTEM=="usb", ATTR{idVendor}=="1e0e", ATTR{idProduct}=="9001", SYMLINK+="rplidar"

1. **Reload the udev Rules:** After saving the udev rules, reload them:

bash

Copy code

sudo udevadm control --reload-rules

1. **Check the Device Paths Again:** Unplug and reconnect the devices. Then, check:

bash

Copy code

ls -l /dev/external\_camera

ls -l /dev/internal\_camera

ls -l /dev/rplidar

These commands should confirm that each device now has a unique device path.

After following these steps, you should be able to use the correct device paths in your Python code and avoid conflicts between the camera and LIDAR. Let me know if you encounter any issues along the way!

4o

**You said:**

I think the lidar is showing up as 210x etc.

**ChatGPT said:**

ChatGPT

It looks like the **LIDAR** is showing up as a **Silicon Labs CP210x UART Bridge** device (ID 10c4:ea60), which is a USB-to-serial converter. This is common for many LIDAR models, as they typically interface through a serial port.

**Here's what you can do to confirm and address the LIDAR device:**

1. **Verify LIDAR as a Serial Device:** Since it's showing up as a USB-to-serial device (CP210x), it will likely be mapped to a serial port like /dev/ttyUSB0. You can confirm this by running:

bash

Copy code

ls /dev/ttyUSB\*

This should show the LIDAR device if it’s mapped to a port.

1. **Check the Serial Port:** If it’s indeed showing up as /dev/ttyUSB0 or another serial device path, you can check the output of dmesg to confirm which USB device is being mapped to the serial port:

bash

Copy code

sudo dmesg | grep tty

You should see something like:

arduino

Copy code

[12345.678901] usb 1-1.3: cp210x converter now attached to ttyUSB0

1. **Create a Custom udev Rule for the LIDAR:** Once you've confirmed the LIDAR device, you can create a custom udev rule for it to avoid conflicts with other devices, particularly the internal camera.
   * Create a udev rule for the **LIDAR** (CP210x):

bash

Copy code

sudo nano /etc/udev/rules.d/99-lidar.rules

Add the following rule (replace idVendor and idProduct with the correct values from lsusb):

bash

Copy code

SUBSYSTEM=="usb", ATTR{idVendor}=="10c4", ATTR{idProduct}=="ea60", SYMLINK+="rplidar"

This rule will create a symlink /dev/rplidar for your LIDAR device.

1. **Reload the udev Rules:** After saving the rule, reload the udev rules to apply the changes:

bash

Copy code

sudo udevadm control --reload-rules

1. **Verify Device Paths:** Once the rules are reloaded, unplug and reconnect your LIDAR. You can check the symlink to ensure the LIDAR is correctly assigned:

bash

Copy code

ls -l /dev/rplidar

**Final Check:**

Now, you should have:

* **External Camera**: /dev/external\_camera
* **Internal Camera**: /dev/internal\_camera
* **LIDAR**: /dev/rplidar

With these device paths, you can use them directly in your Python code, e.g.,:

* **LIDAR**:

python

Copy code

lidar = RPLidar('/dev/rplidar')

* **Camera**:

python

Copy code

cap = cv2.VideoCapture('/dev/external\_camera')

Let me know how it goes!

4o

**You said:**

The sudo nano /etc/udev...... line will probably set the usb index once. Should be saved in some file for coming up everytime after booting?

**ChatGPT said:**

ChatGPT

Yes, you're right! The custom udev rule that you create will ensure that the **LIDAR** and **cameras** get consistently assigned to the same device paths every time the system boots.

The udev rule you create in /etc/udev/rules.d/99-lidar.rules (or similar) is persistent across reboots. This is because udev is responsible for managing device nodes in Linux, and the rules you define will be applied automatically each time the system detects devices during boot or when they are plugged in.

**Here's a breakdown of how this works:**

1. **udev Rules Persist Across Reboots:**
   * When you create the rule in /etc/udev/rules.d/, it tells udev how to handle devices with specific **vendor** and **product IDs** (like your LIDAR or cameras).
   * These rules are automatically applied when the system detects devices at boot or when new devices are plugged in.
2. **The Rule Creates a Persistent Symlink:**
   * The line SYMLINK+="rplidar" in your udev rule ensures that each time the LIDAR is detected, it is **assigned** to /dev/rplidar (or whatever symlink you specify).
   * This symlink (/dev/rplidar) will remain consistent, even across reboots, so your code can always reference it, regardless of which USB port the LIDAR is plugged into.
3. **No Need to Manually Set the USB Index:**
   * You don't need to worry about manually setting the USB index or worrying about the device path changing on each boot. The udev rule takes care of this.
4. **Example of Persistent Rule:** Once you add the udev rule (e.g., for the LIDAR), it will automatically be applied when the LIDAR is connected. No additional steps are needed, and it should work reliably every time.

**To Recap:**

* The udev rule you added will ensure that the device is always given the same symbolic link (/dev/rplidar) even after a reboot.
* The udev system will handle device naming automatically, so you don't have to worry about the device index (USB port) or it changing.

If you want to ensure everything works properly, you can reboot the system, unplug and reconnect the devices, and check if the symlinks (/dev/rplidar, /dev/external\_camera, etc.) are present and consistent.

Let me know if anything else needs clarification!

4o

**You said:**

Inspite of adding internal camera with its ids in the udev rules file, it still shares the index with internal camera. External camera has a different index, and shows up in ls -l /dev. But Internal camera does not show up.

**ChatGPT said:**

ChatGPT

The behavior you're experiencing suggests that the internal camera might not be recognized by udev in the same way as USB devices, as internal cameras are often managed differently in Linux. Here's how you can address this situation:

**Step-by-Step Debugging and Solutions**

1. **Verify Internal Camera Detection:** The internal camera might not be showing in /dev because it is managed by the kernel as a non-USB device. To confirm its presence:

bash

Copy code

ls /dev/video\*

You should see something like /dev/video0, /dev/video1, etc., where the internal camera might be video0.

1. **Check Which Device is the Internal Camera:** To identify which /dev/video\* corresponds to the internal camera:

bash

Copy code

v4l2-ctl --list-devices

This will list all available video devices and their associated device paths, e.g.:

javascript

Copy code

Sony Visual Communication Camera (usb-0000:00:14.0-6):

/dev/video0

External USB Camera (usb-0000:00:14.0-3):

/dev/video1

1. **udev Rules for Internal Camera:** Internal cameras are sometimes tricky because they might not be managed through standard USB protocols. If your internal camera does have a vendor and product ID (from lsusb), you can try using those in a udev rule. However, if it's not working:
   * Use v4l2-ctl to confirm the camera is detected.
   * Ensure the IDs you are using are correct for the internal camera.
2. **Force an Internal Camera Symlink with udev:** If the internal camera is managed through /dev/video\*, you can still force a symlink. Create a rule like this:

bash

Copy code

SUBSYSTEM=="video4linux", KERNEL=="video0", ATTRS{idVendor}=="04f2", ATTRS{idProduct}=="b290", SYMLINK+="internal\_camera"

Replace the idVendor and idProduct with the actual IDs for your internal camera from lsusb.

Save the rule to /etc/udev/rules.d/99-internal-camera.rules, reload, and test:

bash

Copy code

sudo udevadm control --reload-rules

sudo udevadm trigger

ls -l /dev/internal\_camera

1. **Check for Conflicts:**
   * If the internal camera still shares an index (/dev/video0) with another device, ensure no other device is being prioritized or assigned the same index in the udev rules.
2. **Alternative Solution (Manual Assignment):** If udev is still not working for the internal camera:
   * Use the output of v4l2-ctl to identify the device path manually (/dev/video0 or similar).
   * Directly reference the identified path in your code for now.

**Testing in Code**

Once you've confirmed the paths for the cameras, test in your Python code to ensure they are accessed correctly. For example:

python

Copy code

import cv2

# Test internal camera

cap\_internal = cv2.VideoCapture('/dev/internal\_camera')

ret, frame = cap\_internal.read()

if ret:

print("Internal camera working")

else:

print("Internal camera not detected")

cap\_internal.release()

**Next Steps**

If these steps don't resolve the issue, let me know:

* The output of lsusb
* The output of v4l2-ctl --list-devices
* Any messages from dmesg | grep video

We can troubleshoot further based on that!

4o

**You said:**

ls -l /dev/internal\_camera lrwxrwxrwx 1 root root 6 Dec 13 15:35 /dev/internal\_camera -> video0 ravi@ravi-VPCEG3AEN:~$ ls -l /dev/external\_camera lrwxrwxrwx 1 root root 15 Dec 13 15:39 /dev/external\_camera -> bus/usb/001/010 ravi@ravi-VPCEG3AEN:~$ ls -l /dev/rplidar lrwxrwxrwx 1 root root 7 Dec 13 15:39 /dev/rplidar -> ttyUSB0 - this is the present status..

**ChatGPT said:**

ChatGPT

The udev rules have successfully created the symbolic links (/dev/internal\_camera, /dev/external\_camera, and /dev/rplidar) to the appropriate devices. However, there seems to be a misalignment in how these symbolic links are mapped. Let's address the issue and verify the setup:

**Observations**

1. **Internal Camera (/dev/internal\_camera)**:
   * Linked to /dev/video0. This seems correct for video devices. You can test this with a video application or Python script.
2. **External Camera (/dev/external\_camera)**:
   * Linked to /bus/usb/001/010. This is incorrect, as /bus/usb/ paths represent the USB device hierarchy, not the video input device. The actual video device for the external camera should also be /dev/video\*.
3. **RPLIDAR (/dev/rplidar)**:
   * Linked to /dev/ttyUSB0. This seems correct and aligns with the usual naming for USB serial devices.

**Fix the udev Rule for the External Camera**

The external camera should also link to a /dev/video\* device rather than a USB path. Update the udev rule for the external camera:

1. **Identify the Correct Video Path**: Run the following to list video devices and confirm the external camera’s actual device path:

bash

Copy code

v4l2-ctl --list-devices

Look for the external camera (e.g., "Microsoft LifeCam VX-700") and note the associated /dev/video\*.

1. **Update the Rule**: Edit the udev rule for the external camera (/etc/udev/rules.d/99-external-camera.rules or similar):

bash

Copy code

SUBSYSTEM=="video4linux", ATTRS{idVendor}=="045e", ATTRS{idProduct}=="0770", SYMLINK+="external\_camera"

Replace the idVendor and idProduct with the values from lsusb for your external camera (they match your output: 045e and 0770).

1. **Reload udev Rules**: Apply the changes:

bash

Copy code

sudo udevadm control --reload-rules

sudo udevadm trigger

1. **Test the New Setup**: Confirm the new link:

bash

Copy code

ls -l /dev/external\_camera

It should now point to /dev/video\*.

**Verify All Devices**

After fixing the external camera, verify all devices are working:

1. **Check Symbolic Links**:

bash

Copy code

ls -l /dev/internal\_camera

ls -l /dev/external\_camera

ls -l /dev/rplidar

1. **Test with Code**: Test each device in Python:
   * **Internal and External Cameras**:

python

Copy code

import cv2

# Test internal camera

cap\_internal = cv2.VideoCapture('/dev/internal\_camera')

ret, frame = cap\_internal.read()

if ret:

print("Internal camera working")

else:

print("Internal camera not detected")

cap\_internal.release()

# Test external camera

cap\_external = cv2.VideoCapture('/dev/external\_camera')

ret, frame = cap\_external.read()

if ret:

print("External camera working")

else:

print("External camera not detected")

cap\_external.release()

* + **RPLIDAR**: Run your existing RPLIDAR Python test code.

**Next Steps**

If the external camera still links incorrectly, let me know:

* The output of v4l2-ctl --list-devices
* Any additional dmesg | grep video output when plugging in the external camera

With that information, we can troubleshoot further.

4o

**You said:**

lrwxrwxrwx 1 root root 6 Dec 13 15:46 /dev/internal\_camera -> video0 lrwxrwxrwx 1 root root 6 Dec 13 15:46 /dev/external\_camera -> video2 lrwxrwxrwx 1 root root 7 Dec 13 15:46 /dev/rplidar -> ttyUSB0

**ChatGPT said:**

ChatGPT

Great! Now, everything seems to be correctly mapped:

1. **Internal Camera**: /dev/internal\_camera -> video0
2. **External Camera**: /dev/external\_camera -> video2
3. **RPLIDAR**: /dev/rplidar -> ttyUSB0

**Next Steps**

**1. Verify Cameras:**

* Test both internal and external cameras using Python code:

python

Copy code

import cv2

# Test internal camera

cap\_internal = cv2.VideoCapture('/dev/internal\_camera')

ret, frame = cap\_internal.read()

if ret:

print("Internal camera working")

else:

print("Internal camera not detected")

cap\_internal.release()

# Test external camera

cap\_external = cv2.VideoCapture('/dev/external\_camera')

ret, frame = cap\_external.read()

if ret:

print("External camera working")

else:

print("External camera not detected")

cap\_external.release()

**2. Verify RPLIDAR:**

* Run your existing RPLIDAR Python code to confirm it works with /dev/rplidar.

**Troubleshooting**

If you encounter any issues:

* Ensure your udev rules are correctly applied by checking the /etc/udev/rules.d directory for any leftover rule files.
* Double-check the hardware IDs (idVendor and idProduct) in the udev rules to ensure they match your specific hardware.

Let me know how the tests go!

4o mini

**You said:**

Internal camera working External camera working

**ChatGPT said:**

ChatGPT

That's fantastic! Both the internal and external cameras are working as expected. Now that everything is set up, your RPLIDAR is also correctly mapped and functional.

**Summary:**

* **Internal Camera**: /dev/internal\_camera -> video0
* **External Camera**: /dev/external\_camera -> video2
* **RPLIDAR**: /dev/rplidar -> ttyUSB0

If everything is working smoothly, you can continue integrating these into your obstacle avoidance system. If you encounter any issues in the future, feel free to reach out!

Great work! 🎉

4o mini

**You said:**

Environment setupℑ Sourcing the setup scriptℑ Set up your environment by sourcing the following file. # Replace ".bash" with your shell if you're not using bash # Possible values are: setup.bash, setup.sh, setup.zsh source /opt/ros/humble/setup.bash - this is what was done.

**ChatGPT said:**

ChatGPT

It looks like you've sourced the setup script for ROS Humble successfully. This step sets up the necessary environment variables so that your ROS workspace and packages are accessible.

**Additional Steps After Setting Up the ROS Environment**

1. **Verify ROS Environment**:

bash

Copy code

echo $ROS\_VERSION

echo $ROS\_DISTRO

1. **Verify ROS Packages**:

bash

Copy code

rospack list

1. **Initialize ROS Workspace**:

bash

Copy code

source ~/your\_ws/devel/setup.bash

1. **Run ROS Node**: You can now run your ROS nodes:

bash

Copy code

ros2 run <package\_name> <node\_name>

If there are any issues or additional tasks you’d like to proceed with, let me know!