Developing your first ROS 2 Publisher/Subscriber package

Kheng Lee Koay

February 4, 2026

Contents

Chapter 1 Developing first ROS 2 package

In this guide, you will learn how to set up our **Robot Operating System 2 (ROS 2) workspace** and walk through the **Python** development workflow to create your first **publisher** and **subscriber** (also known as a **listener**) nodes. You will learn the fundamentals of **ROS 2** communication, focusing on **publisher-subscriber** interaction between nodes, through a hands-on exercise in building a ROS 2 package with Python.

This tutorial assumes that you have **ROS 2 Humble** installed on **Ubuntu 22.04**, and that you are using **Visual Studio Code** (**VS Code**) as your code editor.

You are encouraged to use external resources, including ChatGPT and online documentation whenever needed. When seeking help from ChatGPT, be sure to include relevant details such as **package** name, **node** name, programming language, and build tool (e.g., colcon) to receive accurate and specific assistance.

All of our ROS 2 packages (i.e. codes) will be created inside the **source directory** (i.e. `src`) of the ROS 2 workspace, and we will use **Ubuntu Terminals** to run and test our nodes.

In this session, you will:

1. Set up the ROS 2 development environment
2. Create a new ROS 2 package, including:
   a. Generate the package structure
   b. Implement publisher and subscriber nodes
3. Build the package and run the nodes

By the end of this tutorial, you'll have a functional ROS 2 package that demonstrates basic message passing using Python, which is a foundational step in building more complex robotic systems.

1.1 ROS 2 Workspace

A **ROS 2 workspace** is a structured environment used to build, organise, and manage your ROS 2 projects. It acts as a central directory where you can create and maintain multiple ROS2 packages in a clean and organised way.

The workspace follows a specific directory structure that separates your development files from system files, helping to avoid conflicts and keeping your project man-

ageable. It also simplifies tasks such as building code, handling dependencies, and running nodes, making it an essential part of ROS 2 development.

1.1.1 Creating a Directory for Your ROS 2 Workspace

We will start by setting up a dedicate directory for your ROS 2 development. First, create a directory named `Workspaces` in your home directory. Inside this directory, we will create a ROS 2 workspace called `ros2_ws`, which will contain all your ROS 2 projects Within the `ros2_ws` workspace, create a `src` (source) directory. This is where all your personal ROS 2 packages will be stored and managed.

To set this up, open a `terminal` and run the following command:

```
# Create the ROS 2 workspace with -p flag ensures that all necessary \
   parent directories are created if they don't already exist.
~$ mkdir -p ~/Workspaces/ros2_ws/src
```

☞ **Remark**

Understanding the Terminal Prompt

When working in the Ubuntu terminal, you will often see a prompt that looks like this:

```
username@Ubuntu:~$
```

Here is a breakdown of what each part means:

username : The name of the currently logged-in user.

Ubuntu : The name of the computer (hostname).

~ : A shortcut representing the user's home directory(i.e., /home/username).

$: Indicates a standard user prompt (as opposed to #, which represents the root or superuser).

For example, in the following line:

```
username@Ubuntu:~$ mkdir Workspaces
```

`mkdir Workspaces` is the command being entered, and the prompt `username@Ubuntu:~$` shows that the command is being run by the user `username`, on a machine named `Ubuntu`, from the home directory `~`.

1.1.2 Source the ROS Humble environment

Sourcing the ROS Humble environment in your terminal sets up the necessary paths and environment variables required for using ROS 2. This step enables the terminal to access ROS packages, tools, and commands, allowing you to build, run, and manage your ROS projects effectively.

Because each new terminal session starts with a fresh environment, you need to source the ROS setup script every time you open a terminal before working with ROS.

To source the ROS Humble environment manually, run this command:

```
# source ROS Humble environment
~$ source /opt/ros/humble/setup.bash
```

☞ Remark

To avoid running this command every time you open a terminal, you can add it to your `~/.bashrc` file. This will automatically source the ROS environment in every new terminal session. Run the following command once:

```
~$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

1.1.3 Building the ROS 2 Workspace

Building your ROS 2 workspace is a critical step in the development workflow. This process ensures that your ROS packages are correctly compiled, dependencies are resolved and everything is ready to run within the ROS 2 environment.

Although Python nodes do not need to be compiled, they still depend on a properly build workspace to manage dependencies and integrate with other ROS components. During the build process, ROS 2 performs the following tasks:

- ✿ Resolves dependencies between packages.
- ✿ Compiles source code (such as C++ code, if any).
- ✿ Links libraries and prepares executables and scripts for use.

Once your workspace is successfully built, you will be ready to run and test your ROS 2 packages.

Steps to build the Workspace:

1. Open a new terminal.
2. Navigate to the root of your ROS 2 workspace directory using the `cd` (change directory) command.

```
# Navigate to the ROS 2 workspace directory
~$ cd ~/Workspaces/ros2_ws
```

3. Run the following command to build the workspace using `colcon`:

```
# Build the workspace using colcon
~$ colcon build
```

After the build process completes, your workspace will be fully set up and ready for development and testing.

However, before using any packages from your workspace, you need to **source the workspace environment** to make its packages and paths available to your terminal session. See the next section (1.1.4) for instructions on sourcing the workspace.

☞ **Remark**

`Colcon` (short for **Co**mmand **L**ine **Con**struction) is the official and recommended build tool for ROS 2. The command `colcon build` is used to compile and organise all packages within your ROS 2 workspace. When run, it automatically:

- ✿ Detects all packages in the workspace.
- ✿ Determines their build dependencies.
- ✿ Builds them in the correct order.

For `C++` packages. `colcon` uses `CMake`, and for`Python` packages, it uses `setuptools`. This ensures full compatibility with ROS 2's build system. T After a successful build, `colcon` creates the following directories in your workspace:

- ✿ `build` : intermediate build files.
- ✿ `install` : final install-ready files (executables, scripts, config files).
- ✿ `log` : build and error logs.

To make the newly built packages usable in ROS 2 applications, you must source the `setup.bash` file inside the the `install/` directory. This step is described in detail in Section 1.1.4).

☞ **Remark**

If you are experiencing issues building the workspace on your personal machine (unlike the lab environment), it is possible that the required `colcon` extensions are not installed. To install the `colcon common extensions`, open a terminal and run the following command. You will need to enter your password, as it requires administrative (`sudo`) privileges.

```
~$ sudo apt install python3-colcon-common-extensions
```

1.1.4 Source the Workspace (Overlay)

After building your ROS 2 workspace, you must **source** it to make its packages, environment variables, and configurations available in your terminal session. This process overlays your custom workspace on top of the base ROS 2 installation (known as the underlay), allowing you to work with your own packages alongside the core ROS tools.

To source your ROS 2 Workspace, run the following command in a terminal:

```
# Source both your workspace (overlay) and the base ROS 2 installation \
   (underlay)
~$ source ~/Workspaces/ros2_ws/install/setup.bash
```

☞ **Remark**

To avoid sourcing the workspace manually every time you open a new terminal, you can add the sourcing command to your `~/.bashrc` file. This ensures your workspace is automatically sourced in each terminal session:

```
~$ echo "source ~/Workspaces/ros2_ws/install/setup.bash" >> ~/.bashrc
```

After adding the command to your `~/.bashrc`, either restart your terminal or run the following command to apply the changes immediately.

```
~$ source ~/.bashrc
```

1.1.5 Environment variables

ROS 2 uses a middleware called the **Data Distribution Service (DDS)** to handle communication between nodes. By default, this communication is not limited to a single machine. Nodes across different systems on the same physical network can discover and communicate with one another.

To manage network-level communication, ROS 2 uses a concept called the **Domain ID**, which helps isolate groups of nodes. By default, all nodes use:

```
ROS_DOMAIN_ID=0
```

This means any ROS 2 nodes running on the same domain ID can freely discover and communicate with each other. However, when multiple groups or robots share the same network, such as in classrooms or labs, this can lead to unintended interference. To avoid this, you can assign a unique `ROS_DOMAIN_ID` to each group.

It is recommended to choose a unique integer between `1--101` or `215--232` [1]. For demonstration purposes, we will use `19` as our domain ID:

```
# set ROS_DOMAIN_ID with a unique integer
~$ export ROS_DOMAIN_ID=19
```

☞ **Remark**

Tip: Persisting Your Domain ID

To avoid setting the `ROS_DOMAIN_ID` every time you open a terminal, you can append the following line to your `~/.bashrc` file. This ensures the variable is automatically configured for each new session:

```
~$ echo "export RROS_DOMAIN_ID=19" >> ~/.bashrc
```

After modifying your `~/.bashrc`, either restart your terminal or run:

```
~$ source ~/.bashrc
```

to apply the change immediately.

Restricting Communication to Localhost --- If all your ROS 2 nodes are running on a single machine, you can prevent network-based discovery entirely by setting:

```
~$ export ROS_LOCALHOST_ONLY=1
```

This ensures that nodes will only communicate over the local loopback interface (localhost). This is especially useful in shared environments, like labs or classrooms, to prevent multiple users' nodes from interfering with each other [2]. This environment variable can be unset with:

```
~$ unset ROS_LOCALHOST_ONLY
```

☞ **Remark**

To persist this configuration across terminal sessions, run the following command to append an environment variable to your `~/.bashrc` file, so it is available in new Bash shells:

```
~$ echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

1.2 Creating a ROS 2 Package with `ament_python`

In ROS 2, a package is the fundamental organisational unit for your code. It allows you to manage, build, and share your software efficiently.

1.2.1 Creating a Python-based ROS 2 Package

The `ros2 pkg create` command simplifies creating a new ROS 2 package by automatically generating the necessary directory structure and essential files.

Before creating a new package, navigate to the `src` directory within your ROS 2 workspace, since all your ROS 2 packages should be located there:

```
# Change directory to the workspace's src folder
~$ cd ~/Workspaces/ros2_ws/src
```

To create a ROS 2 package named `my_py_pkg` using the `ament_python` build type, run the following command. This package will declare dependencies on `rclpy` (the ROS 2 Python client library) and `std_msgs` (standard ROS 2 message types like `String` and `Int32`):

```
# Create 'my_py_pkg' package with required dependencies
~/Workspaces/ros2_ws/src$ ros2 pkg create --build-type ament_python --\
   license Apache-2.0 my_py_pkg --dependencies rclpy std_msgs
```

Here is a breakdown of the key options used in the command:

`--build-type ament_python` : Specifies that the package uses the `ament_python` build system, designed for Python-based packages.

`--license Apache-2.0` : Declares the license type as `Apache 2.0`. This information is recorded in the `package.xml` file.

`my_py_pkg` : The name of the package to be created, which becomes the root directory of the package.

`--dependencies rclpy std_msgs` : Lists ROS 2 packages that `my_py_pkg` depends on.

After running the command, the package directory `my_py_pkg` will be created with the following structure:

```
my_py_pkg
├── LICENSE
├── my_py_pkg
```

```
    └── __init__.py
├── package.xml
├── resource
    └──my_py_pkg
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

Here is a brief explanation of the main components:

package.xml : Contains metadata about the package such as name, version, description, license, and dependencies.

resource/my_py_pkg: A marker file used internally by the ROS 2 tools to identify the package.

setup.cfg : Configuration file required for packages with executables, allowing `ros2 run` to locate and execute Python scripts.

setup.py : Contains installation instructions and declares entry points for ROS 2 nodes.

my_py_pkg/: The Python module directory, named after the package, containing implementation code. Includes an `__init__.py` file to mark it as a Python package.

1.2.2 Create Publisher and Subscriber nodes

This tutorial provides a simple guide to creating a publisher node that sends "`Hello World`" messages and a subscriber node that receives them. For a more comprehensive explanation, refer to the official ROS 2 tutorial: Writing a Simple Publisher and Subscriber. You are also encouraged to explore other online resources as needed.

1. **Create the publisher node:** Navigate to the `my_py_pkg/my_py_pkg` directory and create a new Python file named `publisher.py`. Make it executable:

```
# Navigate to the package's Python module directory
~/Workspaces/ros2_ws/src$ cd my_py_pkg/my_py_pkg

# Create publisher.py
~/Workspaces/ros2_ws/src/my_py_pkg/my_py_pkg$ touch publisher.py
```

```
# Make publisher.py executable
~/Workspaces/ros2_ws/src/my_py_pkg/my_py_pkg$ chmod +x publisher.py
```

2. textbfImplement the publisher code:
   Open `publisher.py` in your code editor (e.g., VS Code) and write the publisher node code as shown below:

```python
# publisher.py
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1

def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    # Destroy the node explicitly
    # (optional -otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

Listing 1.1. publisher.py

3. **Create the subscriber node:**
   Similarly, create a `subscriber.py` file in the same directory and make it executable:

```
# Create subscriber.py
~/Workspaces/ros2_ws/src/my_py_pkg/my_py_pkg$ touch subscriber.py

# Make subscriber.py executable
~/Workspaces/ros2_ws/src/my_py_pkg/my_py_pkg$ touch chmod +x \
    subscriber.py
```

4. **Implement the subscriber code:**
   Open `subscriber.py` in your editor and add the subscriber node code as below:

```
# subscriber.py
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
                            String,
                            'topic',
                            self.listener_callback,
                            10)
        self.subscription # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    # Destroy the node explicitly
```

```
    # (optional -otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

Listing 1.2. subscriber.py

5. **Register your nodes in `setup.py`:**
   To enable ROS 2 to locate and execute your nodes, you need to register them in the `console_scripts` section of the `entry_points` field in your package's `setup.py` file:

```
    entry_points={
        'console_scripts': [
            'publisher=my_py_pkg.publisher:main',
            'subscriber=my_py_pkg.subscriber:main',
        ],
    },
```

Listing 1.3. setup.py

   This step ensures that ROS 2 recognises your publisher and subscriber nodes, allowing you to launch them using the `ros2 run` command.

1.2.3 Check and Install Missing Dependencies

Before building your package, use `rosdep` in the root of your ROS 2 workspace (e.g., `ros2_ws`) to automatically check for and install any missing dependencies:

```
~/Workspaces/ros2_ws$ rosdep install -i --from-path src --rosdistro \
   $ROS_DISTRO -y
```

☞ Remark
If you encounter the following error message:

```
"Command 'rosdep' not found, but can be installed with:
sudo apt install python3-rosdep2"
```

it means that `rosdep2` is not installed on your system. You can resolve this by installing it and updating the dependency database:

```
~$ sudo apt install python3-rosdep2
~$ rosdep update
```

1.2.4 Build the package

To compile your package, navigate to the root of your ROS 2 workspace and use the `--packages-select` option with `colcon build` to build only your specific package:

```
~/Workspaces/ros2_ws$ colcon build --packages-select my_py_pkg
```

1.2.5 Run the Publisher and Subscriber Nodes

To run your nodes, open a new terminal and source the ROS 2 environment along with your workspace:

```
# Source your ROS 2 workspace (ros2_ws) as an overlay
~/Workspaces/ros2_ws$ source install/setup.bash
```

Run the command below to list all discovered packages and check that your package is listed:

```
~/Workspaces/ros2_ws$ ros2 pkg list
```

Now start the `publisher` node:

```
 # Run the publisher node
~$ ros2 run my_py_pkg publisher
```

Next, open a separate terminal, source the workspace again, and run the `subscriber` node:

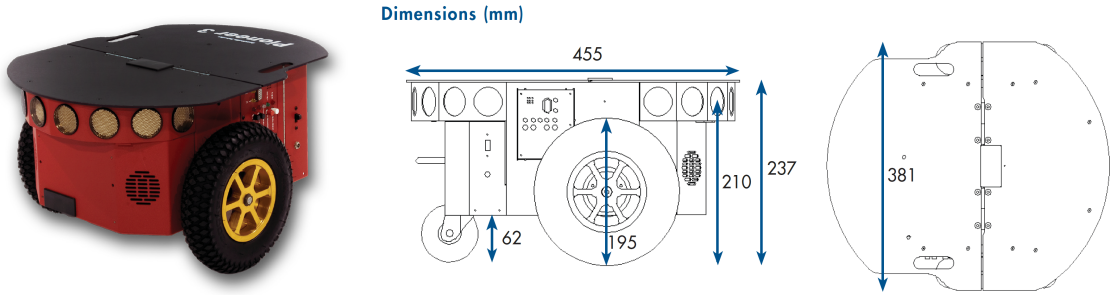
```
 # In a new terminal
~$ source Workspaces/ros2_ws/install/setup.bash
~$ ros2 run my_py_pkg subscriber
```

You should now see the `subscriber` node receiving messages from the `publisher` node. This confirms your ROS 2 package with a basic `publisher` and `subscriber` is functioning correctly.

For further exploration, refer to the official ROS 2 tutorial:
Writing a Simple Py Publisher and Subscriber.

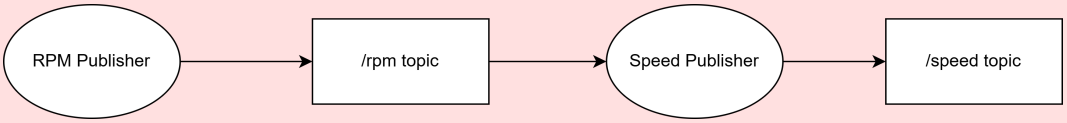
1.3 Task

In this task, you will apply what you learned about publishers and subscribers in a hands-on scenario involving the **Pioneer 3-DX** mobile robot shown below. The robot is equipped with a **differential drive system** and wheels of a specific radius, each featuring a rotary encoder sensor to measure revolutions per minute (RPM). For simplicity, assume that both wheels are rotating at a constant speed (e.g., 10 RPM). Your task is to calculate and publish the robot's speed, specifically its linear velocity in meters per second (m/s).



Exercise 1.1

Create a ROS 2 package containing two Python nodes. The first node (e.g. RPM Publisher) should simulate the RPM[a] of the robot's wheel by publishing a value representing the wheel's RPM. The second node (e.g., Speed Publisher) should subscribe to the topic that receives RPM values, calculate the robot's speed based on these values, and then publish the calculated speed.



[a] RPM in the context of DC motors stands for "Revolutions Per Minute." It is a measure of how many times the motor's shaft completes a full rotation around its axis in one minute. Essentially, it indicates the speed of the motor.

Execution and Output:

1. Run each node in separate terminals.
2. Observe and record the output from each node:
   - the first node should display the RPM
   - the second node should show the calculated speed of the robot

Resources

✿ Standard Message Types in ROS 2 (std_msgs)

✿ Elements of Robotics, 5.8 Wheels Encoder [3]

Bibliography

[1] Open Robotics. (2025) The ros_domain_id variable. [Online]. Available: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Domain-ID.html

[2] ------. (2025) The ros_localhost_only variable. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html#the-ros-localhost-only-variable

[3] M. Ben-Ari and F. Mondada, *Robotic Motion and Odometry*. Cham: Springer International Publishing, 2018, pp. 63--93. [Online]. Available: https://doi.org/10.1007/978-3-319-62533-1_5