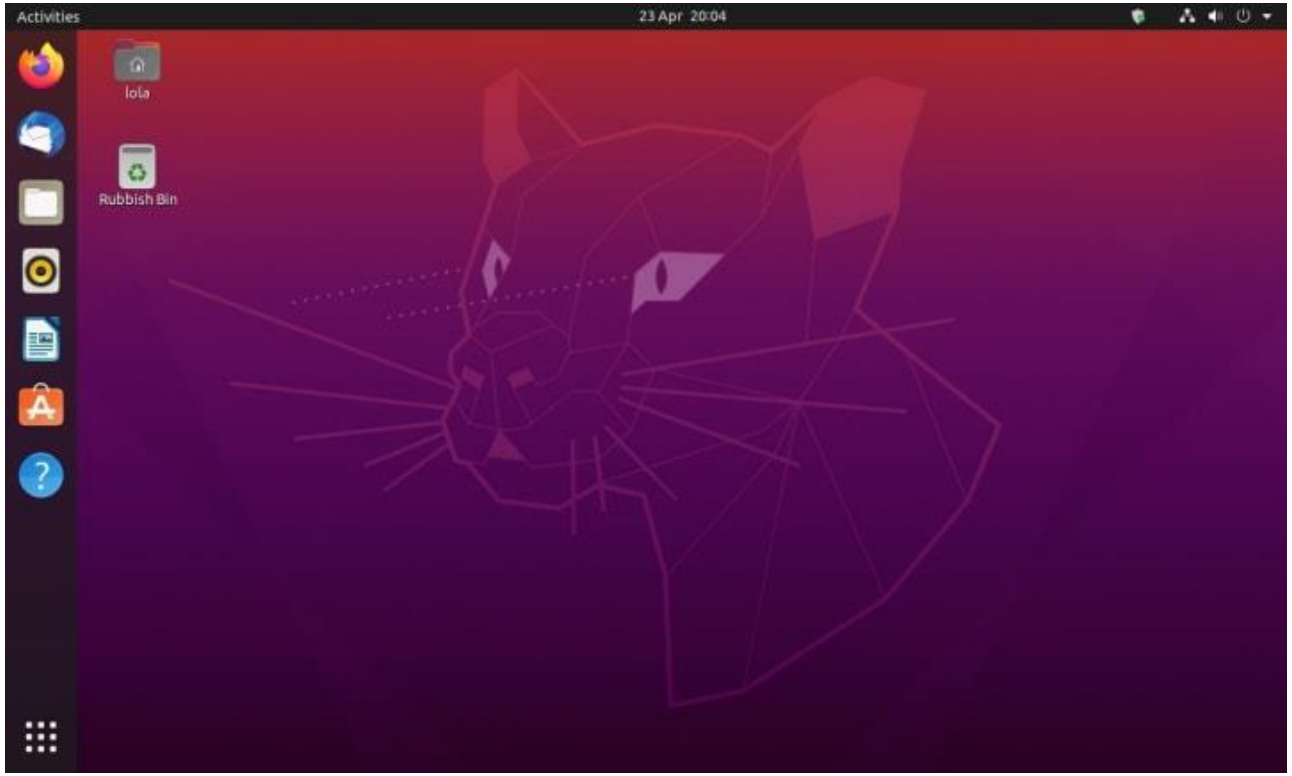
# ROBOTICS LAB 3
## Introduction to ROS

MUNADI SIAL


SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

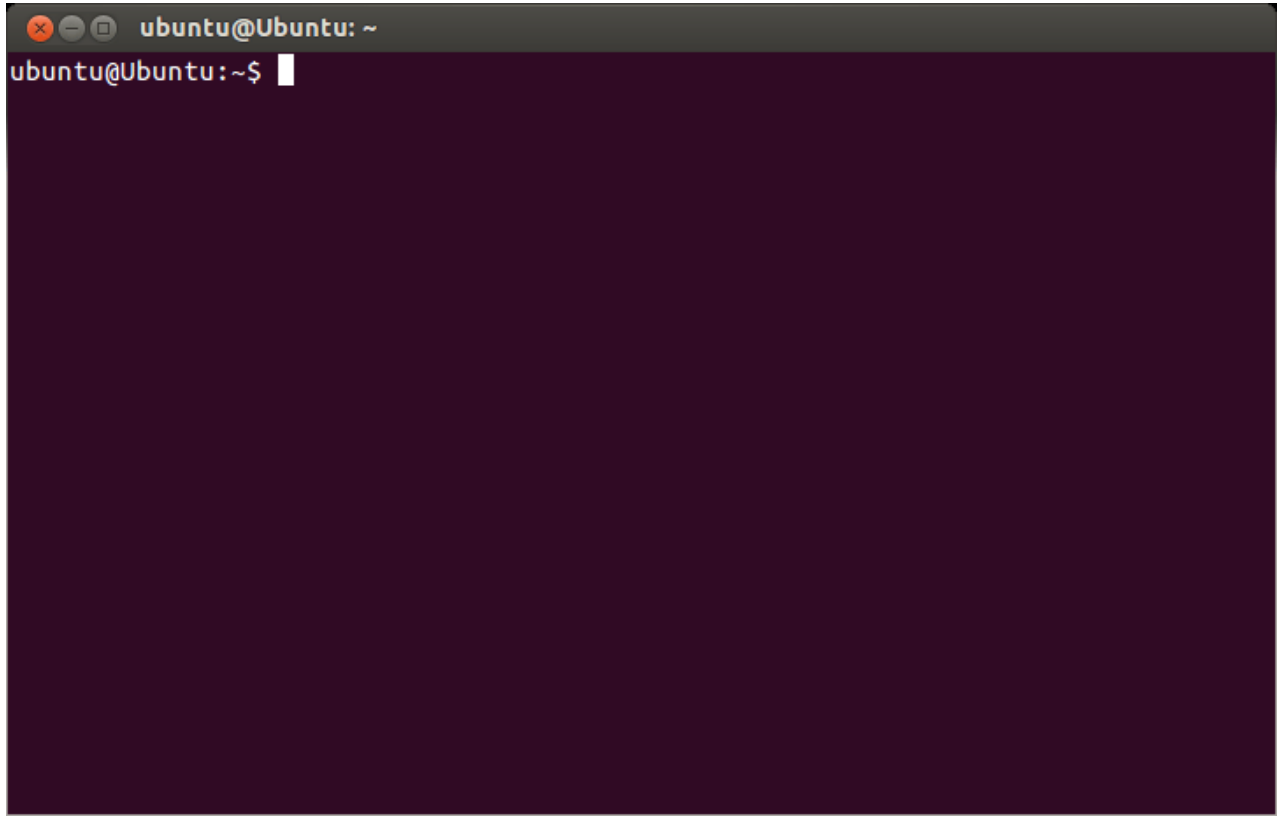
# Linux Ubuntu

- Ubuntu is a popular operating system based on the Linux distribution
- Ubuntu is open-source and has LTS (Long-term Support)
- Ubuntu has an easy-to-use installer that allows computers to dual-boot with another operating system (Windows or MacOS)
- We will use Ubuntu because it has the most support with ROS
- To use Ubuntu effectively, we need to learn the Terminal

# The Linux Terminal

- The Terminal is the command-line of Linux Ubuntu that gives precise control to the user and is extensively used for efficient programming even among professionals
- To learn ROS, familiarizing with the terminal is very important
- In this lab, we will use the terminal to execute python scripts and ROS commands




- To open the terminal, press Ctrl + Alt + T

# Terminal Commands

- Let's understand a few basic commands on the terminal:

| | |
|---|---|
| **pwd** | Print work directory |
| **ls** | List contents in directory |
| **cd <directory>** | Change directory |
| **cd ..** | Go back to parent directory |
| **mkdir** | Make directory |
| **python3 <script>** | Execute python script |
| **man <command>** | Manual page of shell command |
| **sudo** | Run in administrative mode |
| **apt-get** | Install Ubuntu package |
| **mv <source> <dest>** | Move a file |
| **cp <source> <dest>** | Copy a file |
| **rm <file>** | Remove a file |

# Script Execution

- Python scripts are files that end in **.py** format and contain the code
- To write a script, we will use **SublimeText** which is a text editor that allows syntax highlighting for python

- Let's write a simple python script and execute it via the terminal:

```
a = 10
b = "M"
c = 7.6
print(a,b,c)
```

- Once you have typed the above code, save the script as **test.py**
- Now open the terminal, go the directory (with cd) where you saved the script and execute it with the following command:

```
python3 test.py
```

# Import in Python

- In python, we use the **import** keyword to use libraries (modules) which contain functions and classes
- There are many libraries commonly used in python such as NumPy, Pandas, OpenCV, Matplotlib, SciPy, Tensorflow, Keras, PySerial, Math and Datetime etc

- We will use the Math library for the current lab:

```
import math
var = math.sqrt(64)
print(var)
```

```
8.0
```

- We can also assign an *alias* for the library:

```
import math as mt
var = mt.sqrt(25)
print(var)
```

```
5.0
```

# Import in Python

- The math library contains some common mathematical functions:

```
print( math.sqrt(64))
print( math.ceil(1.5) )
print( math.floor(1.5) )
print( math.pi )
print( math.inf )
print( math.sin(30) )
print( math.cos(30) )
print( math.tan(30) )
print( math.log(5) )
print( math.log10(5) )
print( math.radians(180) )
print( math.degrees(math.pi) )
print( math.exp(4) )
```

```
8.0
2
1
3.141592653589793
inf
-0.9880316240928618
0.15425144988758405
-6.405331196646276
1.6094379124341003
0.6989700043360189
3.141592653589793
180.0
54.598150033144236
```

# ROS - Robot Operating System

# ROS - Robot Operating System

- ROS, the Robot Operating System is an open-source framework for programming robots to perform tasks
- ROS is meant to serve as a common software platform for people who are building and using robots
- With ROS, people can share code and ideas more readily
- ROS is a collection of tools, libraries and programs that aim to simplify the task of creating complex and robust robotic platforms

- In this lab, we will use ROS2 Humble

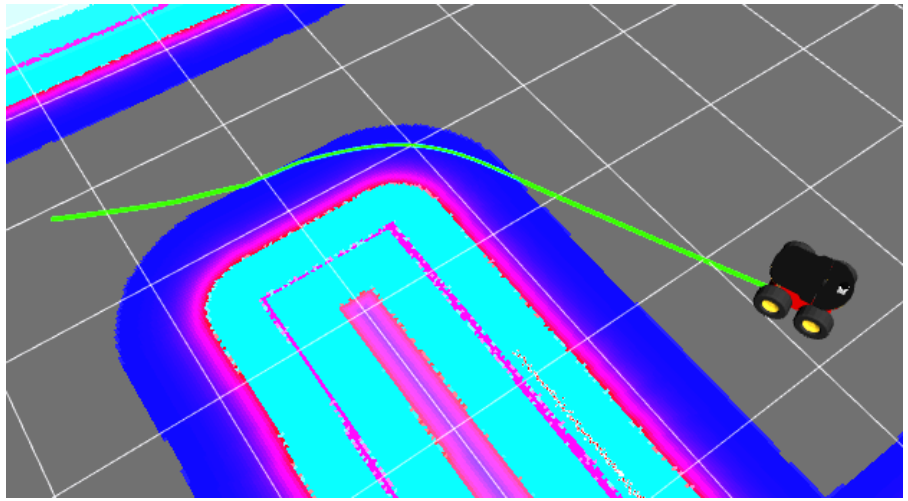
# ROS - Robot Operating System

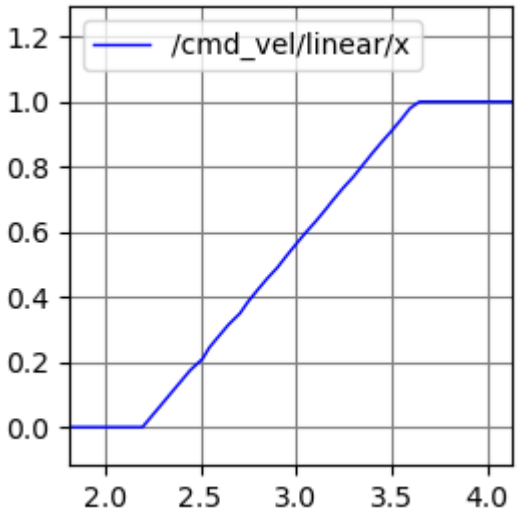
**Why should you learn ROS?**

The short answer is that it saves time. ROS provides all the parts of a robot software system that you would otherwise have to write on your own. ROS allows you to focus on parts of the system that you care about without worrying about the parts that you don't care about.

# ROS - Robot Operating System

- ROS consists of a number of parts:
  - Drivers to read data from sensors and send commands to actuators
  - Tools to implement simulation, visualize states, plot graphs and debug faulty behaviors etc
  - Algorithms for navigation, building maps, sensor fusion, motion planning and object manipulation etc
  - Computation infrastructure to communicate data across various components of a complex robot system





# Brief History of ROS

- ROS is a large project which started as the need for an open collaboration framework was felt by the robotics research community. Various projects at Stanford University in the mid-2000s created prototypes of flexible, dynamic software systems which were extended resources provided by Willow Garage Inc. in 2007. This led to the creation of ROS.
- From the beginning, ROS was being developed at multiple institutions and for multiple robots.

# Philosophy of ROS

- In a broad sense, ROS follows the Unix philosophy of software development:
  - **Peer-to-Peer:** ROS systems are made up of computer programs that connect to one another and exchange data (messages)
  - **Tools-based:** Complex tasks such as navigating source code tree, visualizing interconnections, plotting data streams and logging data etc are done by individual tools which are themselves relatively small and generic
  - **Multilingual:** ROS software modules can be written in any language for which a client library is written. ROS supports C++, Python, R, Java, Javascript, MATLAB and Ruby among others
  - **Open source:** The core of ROS is released under the BSD license which allows commercial and non—commercial use

# ROS1 vs ROS2

- With years of experience, the developers behind ROS1 have learned all of the needed features and improvements in ROS. Unfortunately, such modifications required making many changes which would have made ROS1 unstable. So, ROS2 was created from scratch.

- In ROS1, there is a ROS Master (roscore) which allows nodes to discover each other. In ROS2, there is no ROS Master; each node has the capacity to discover other nodes
- In ROS1, there is a global parameter server which was handled by the ROS Master. In ROS2, each node declares and manages its own parameters, i.e. each node has its own parameter server
- In ROS1, the build system is *catkin*. In ROS2, the build system is *ament* along with the *colcon* build tool

# Installation

- To install ROS2 Humble, open the terminal and execute the following commands:

```
locale  # check for UTF-8
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale  # verify settings
```

```
apt-cache policy | grep universe
```

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

# Installation

- To install ROS2 Humble, open the terminal and execute the following commands:

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

```
sudo apt update
```

```
sudo apt upgrade
```

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

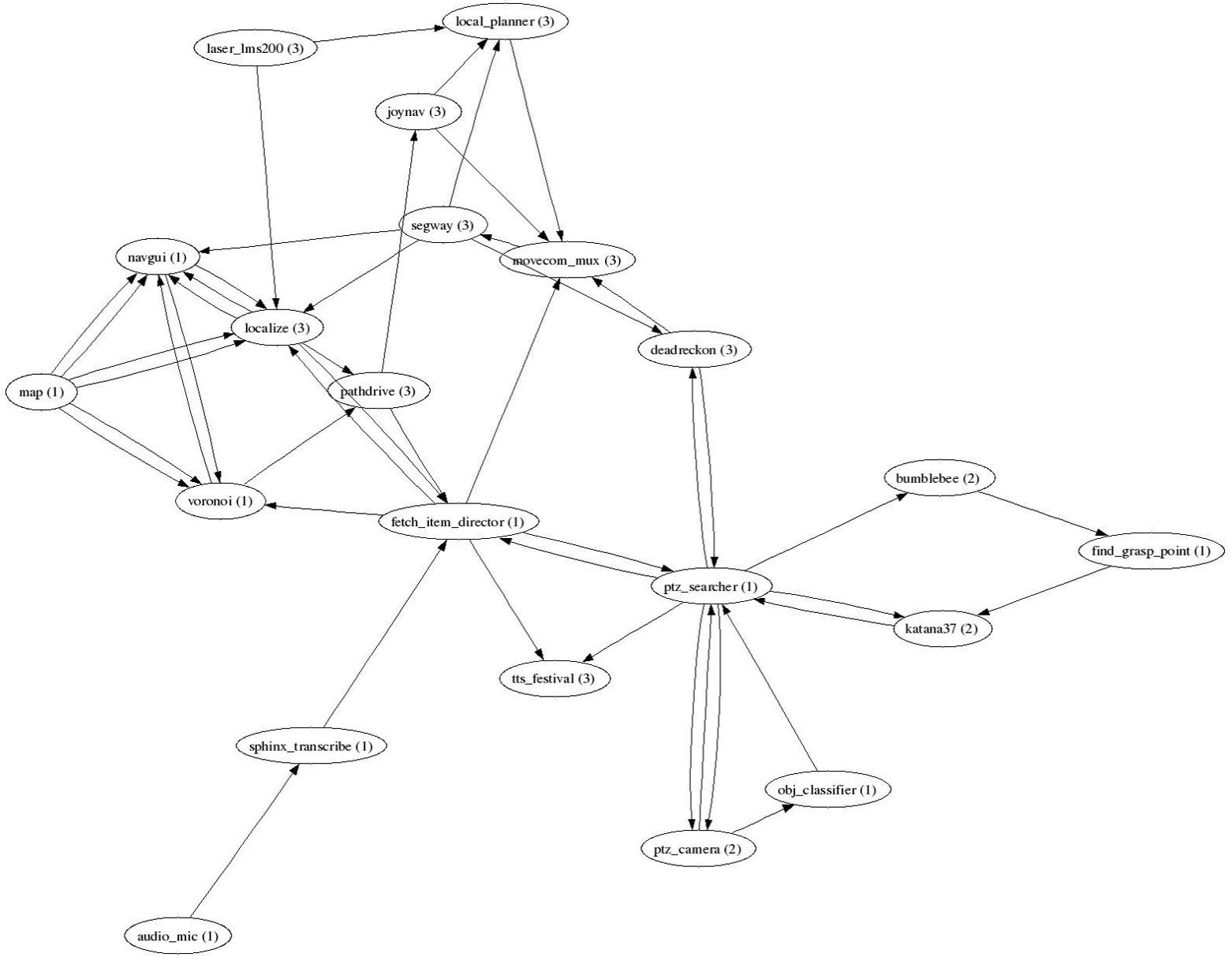
Complete installation guide can be found at official ROS documentation:
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

# The ROS Graph

- ROS systems are made up of a large number of programs that are constantly communicating with each other
- Consider a simple "fetch-an-item" problem for a robot which has several cameras, laser scanners, manipulator arm and a wheeled based. The robot has to navigate the building, find the item and deliver it to the required location
- Such robot tasks lead to some observations about robotics software:
  - The task can be decomposed into many independent subsystems such as navigation, computer vision, grasping etc
  - These subsystems can be used for other tasks such as doing patrols, cleaning, delivering mail etc
  - With proper hardware and abstraction, the majority of application software can run on any other robot

# The ROS Graph

- These subsystems can be represented by a **graph** which consists of nodes and edges
- The figure shows a graph for the "fetch-an-item" robot
- Each node is an individual ROS program (script file)
- Nodes communicate by passing *messages* to other nodes
- Each edge is a message stream

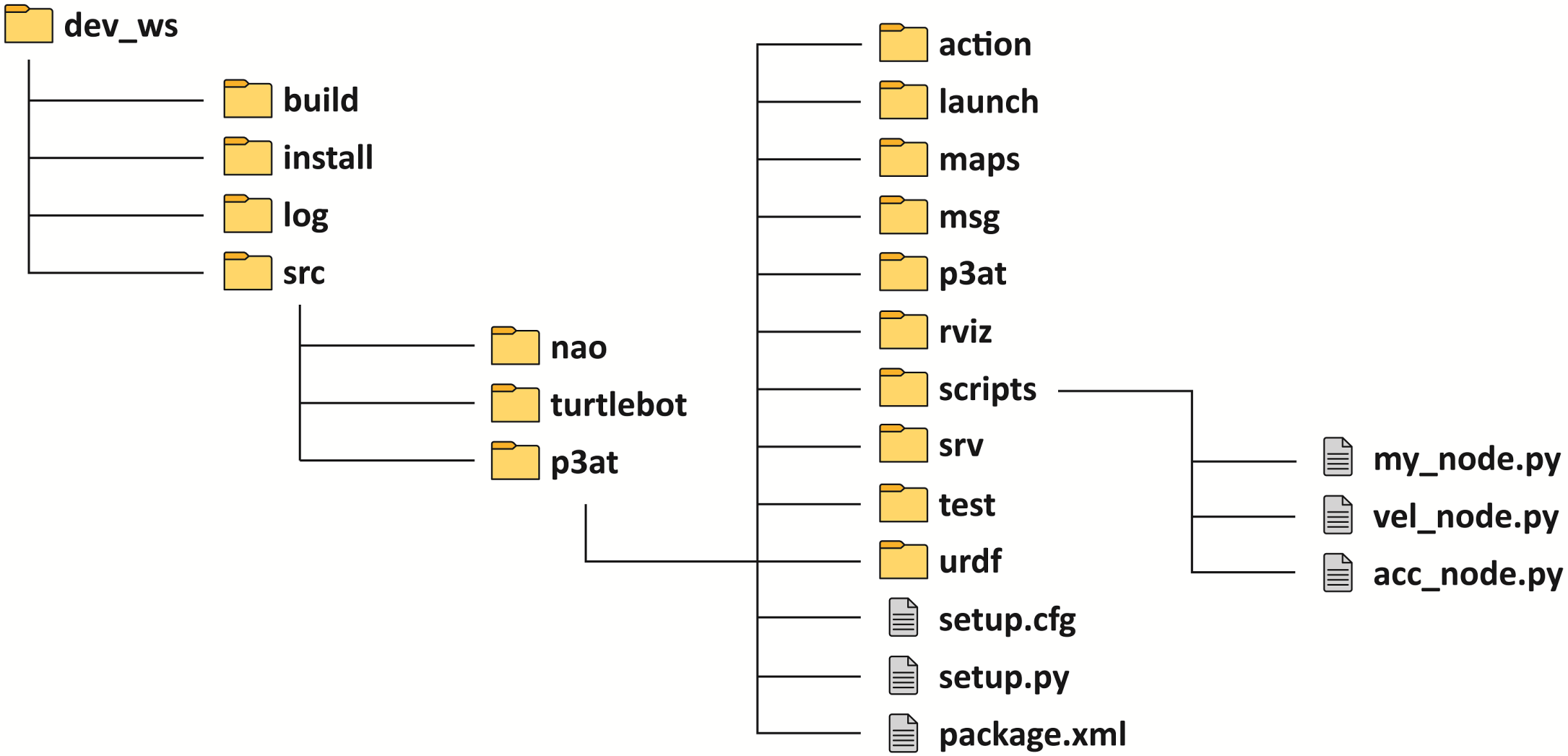


# Using ROS

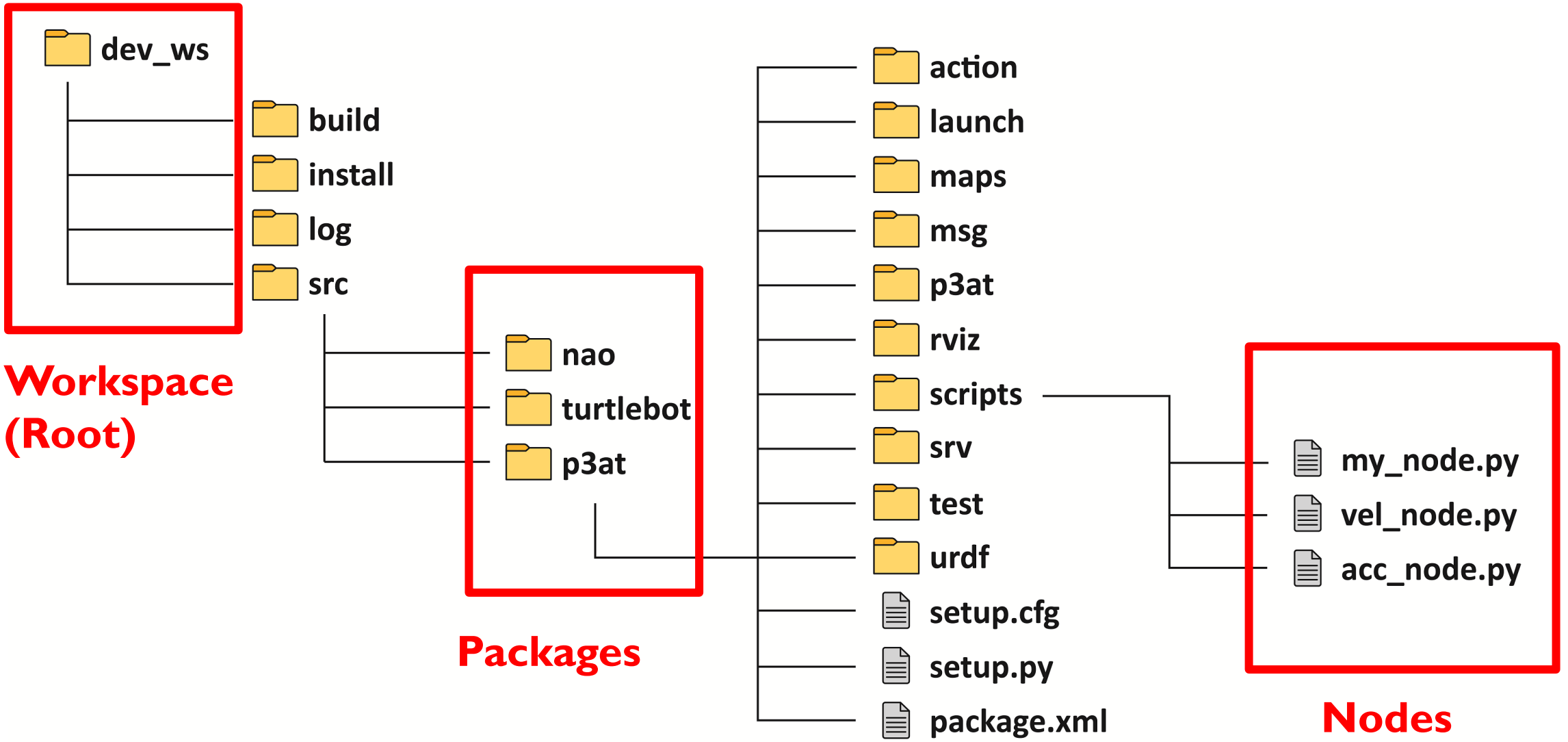
- Before using ROS to program robots, we need to familiarize with a number of ROS concepts such as workspaces, packages and nodes etc
- The following sequence is used for ROS programming:
  - Initializing a **workspace** for the ROS environment
  - Creating/Cloning a **package** in the workspace
  - Creating a script **node** in the package
  - Building the package with colcon build
  - Executing the node with rosrun or roslaunch

- Node execution is of two types:
  - **Rosrun:** Executing an individual node
  - **Roslaunch:** Executing multiple nodes at the same time

# ROS Filesystem



# ROS Filesystem



# Workspaces

- A **Workspace** is a location (set of directories) in the system where you are developing with ROS
- In ROS1, you could run only one workspace at a time
- In ROS2, you can run several active workspaces; the core workspace is called the "underlay" while subsequent workspaces are called "overlays"
- Multiple workspaces make development against different ROS versions (called distributions or distros) much easier
- Multiple workspaces allow installation of different ROS distros on the same computer and switching between them
- In this lab, we will mainly use one workspace

# Add Source to Shell Startup

- The workspace contains the ROS files (code, model, map etc)
- To use the files, the Terminal must be made aware of the workspace. To do this, you need to "source" your workspace (by sourcing the setup file) by typing the following command in the terminal:

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

- You need to source whenever you open a new terminal
- When using ROS, we will use multiple terminals and it is easy to forget to source every time a new terminal is opened
- For this, we place the above command in ".bashrc" which is a script that executes each time a new terminal is opened
- By placing the command in bashrc, sourcing is done automatically
- To add source to bashrc with the terminal:

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

# Packages

- ROS software is organized into **packages** which contain some combination of code, data, documentation, maps, models etc.
- A workspace can have many different packages
- We can either create our own package or clone an existing package from an open repository (such as GitHub)
- A Package can contain multiple nodes
- We can execute nodes from a package
- We can also execute nodes from different packages
- These nodes can communicate with each other by sending data (called messages)
- Packages are simply organized locations in the workspace and nodes are the executable programs in the packages

# Build System

- The build system is the set of tools that ROS uses to generate executable code, libraries, scripts and interfaces that other code can use
- The build system of ROS2 is called **ament** (ament_cmake, ament_python) which is supported by the building tool called **colcon** which uses the package.xml specification
- Once we have created a workspace, we can build it by using the following command from the workspace directory

```
colcon build
```

- Each time we write (or edit) scripts of a package, we have to build the workspace so we can execute those scripts (nodes)
- After building, we have to source the setup.bash in order to add the executables to the path:

```
. install/setup.bash
```

# Creating a Package

- To create a package, we use the following command:

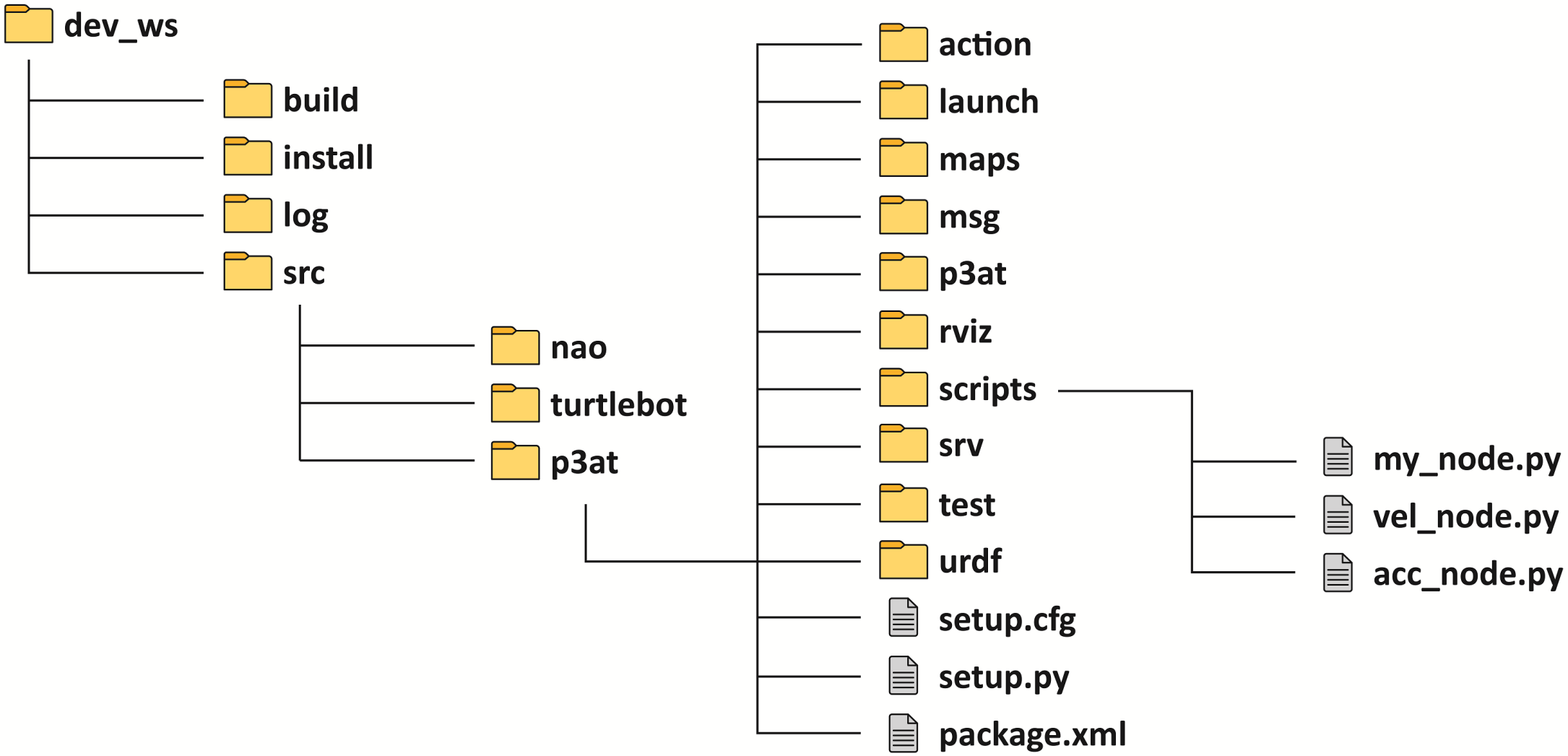
```
ros2 pkg create –build-type ament_python myPackage
```

- To create a package with dependencies, we use the following command:

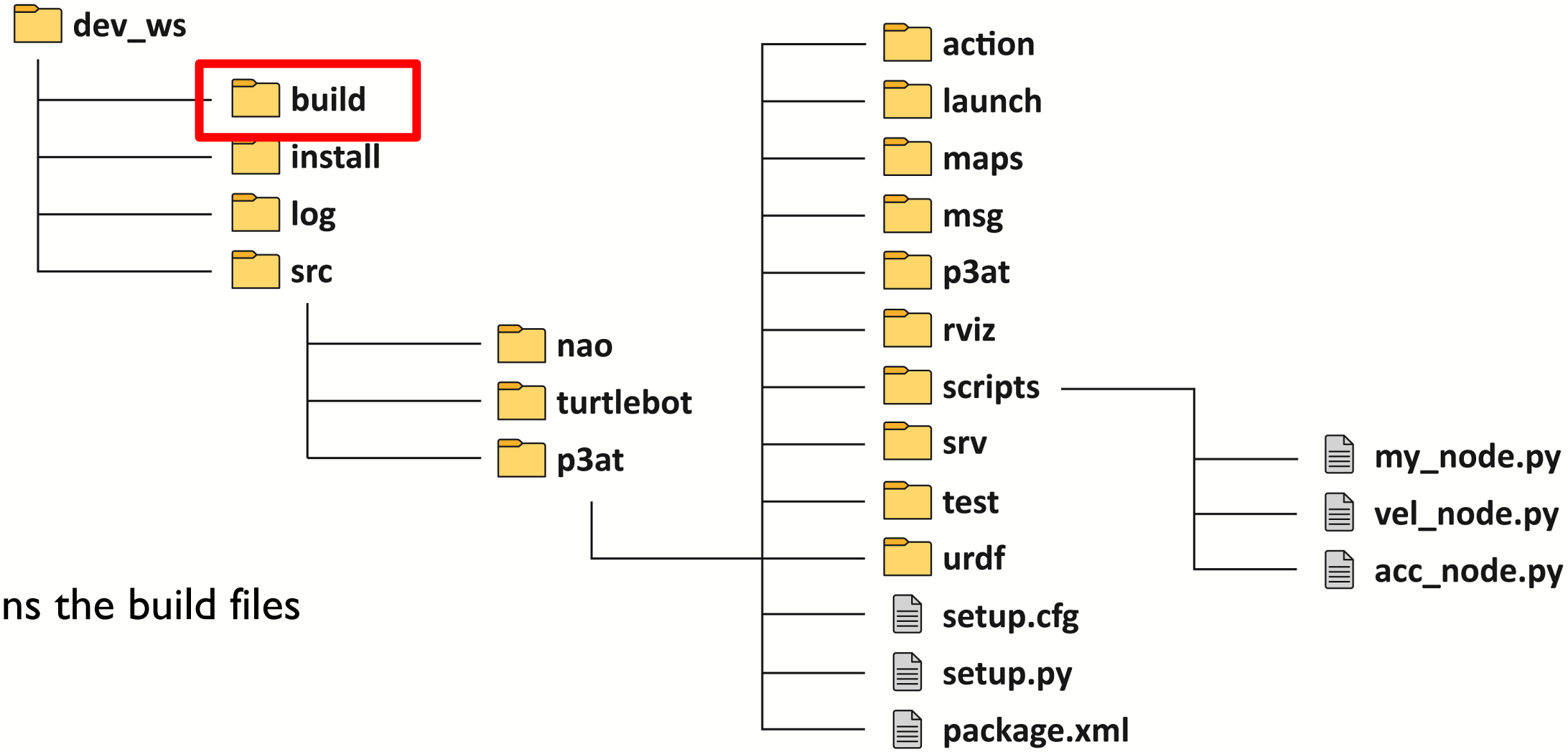
```
ros2 pkg create –build-type ament_python myPackage
--dependencies rclpy std_msgs
```

- In the above command, rclpy and std_msgs are dependencies
- A dependency is another package or library which we are using in our current package

# ROS Filesystem

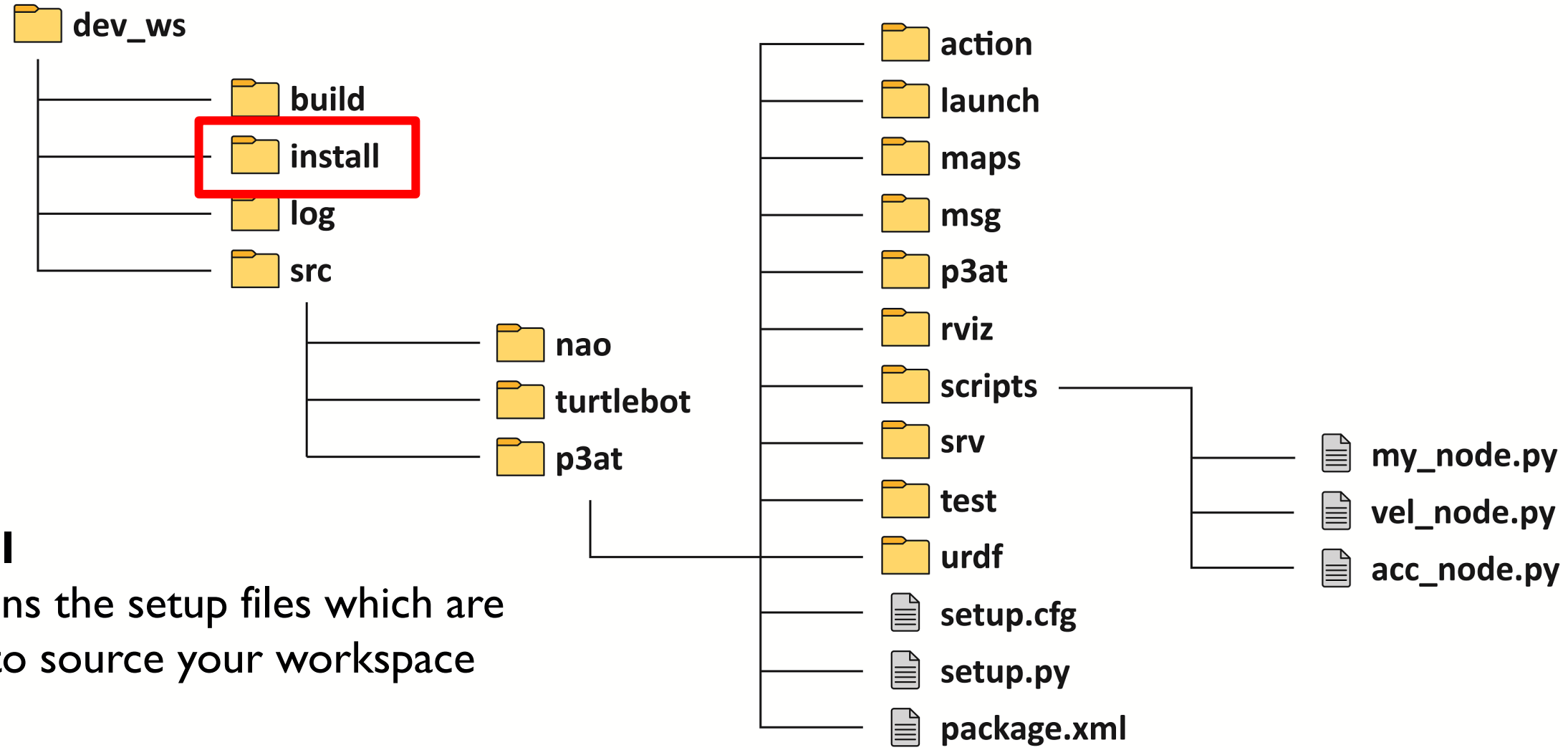


# ROS Filesystem



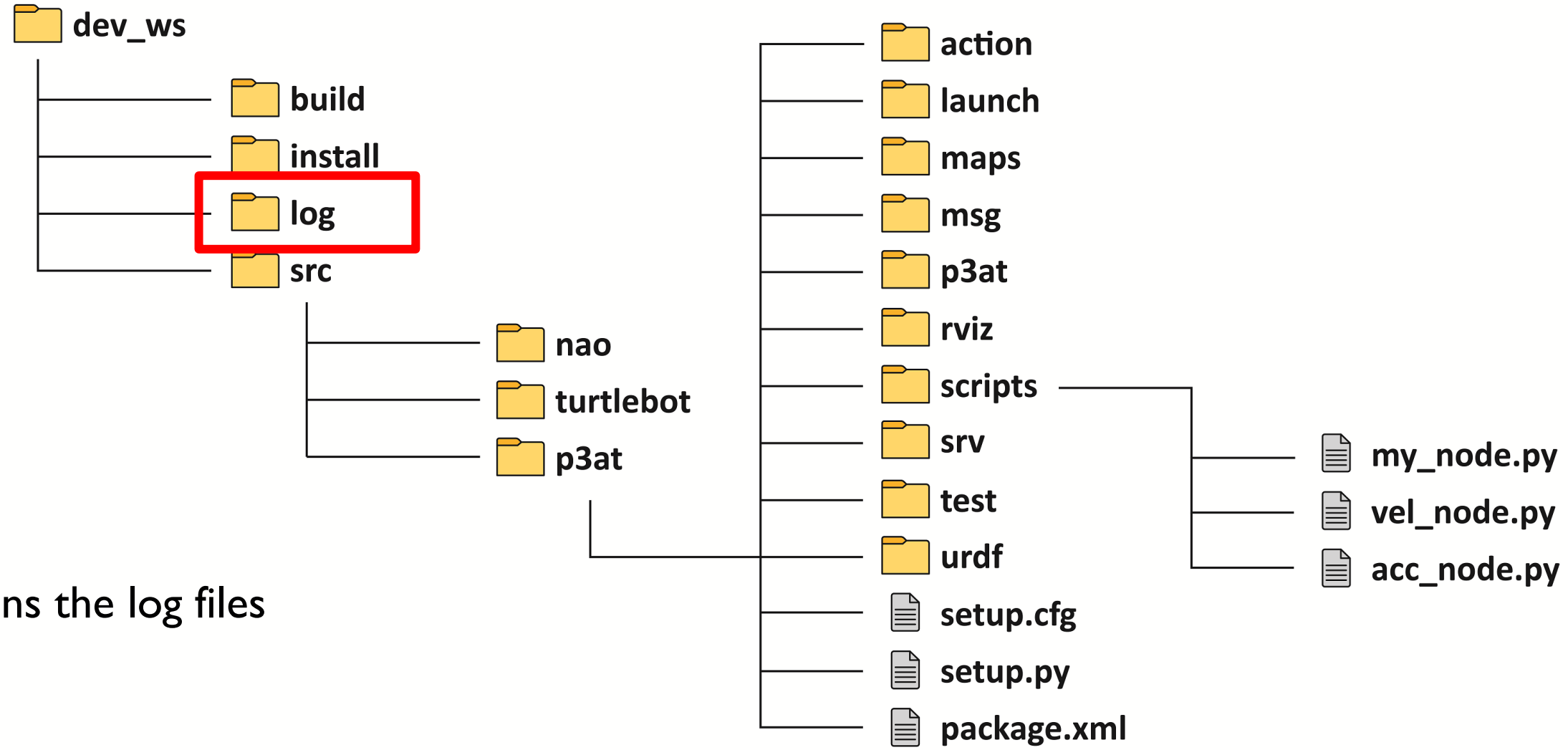

**build**
contains the build files

# ROS Filesystem



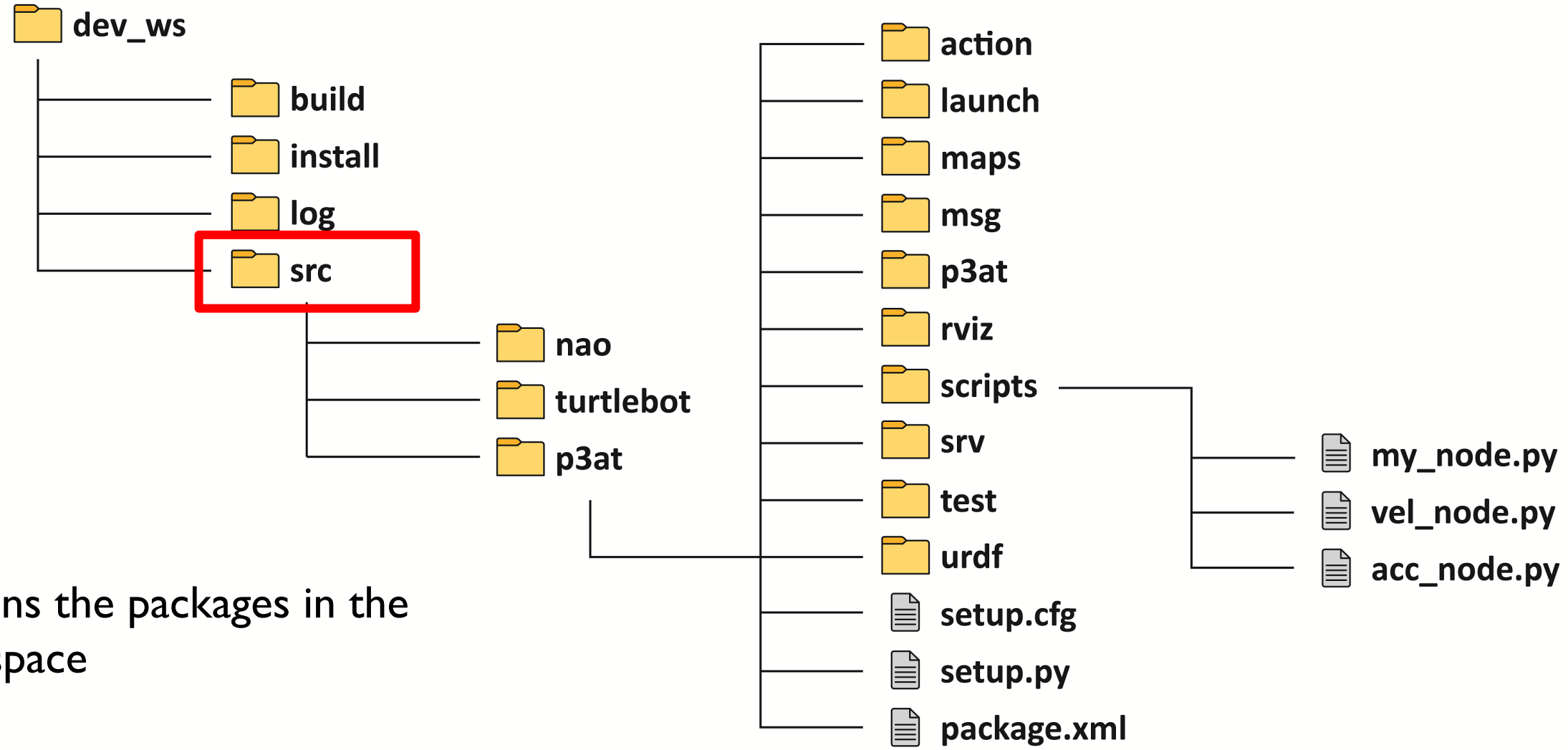

**install**
contains the setup files which are used to source your workspace

# ROS Filesystem



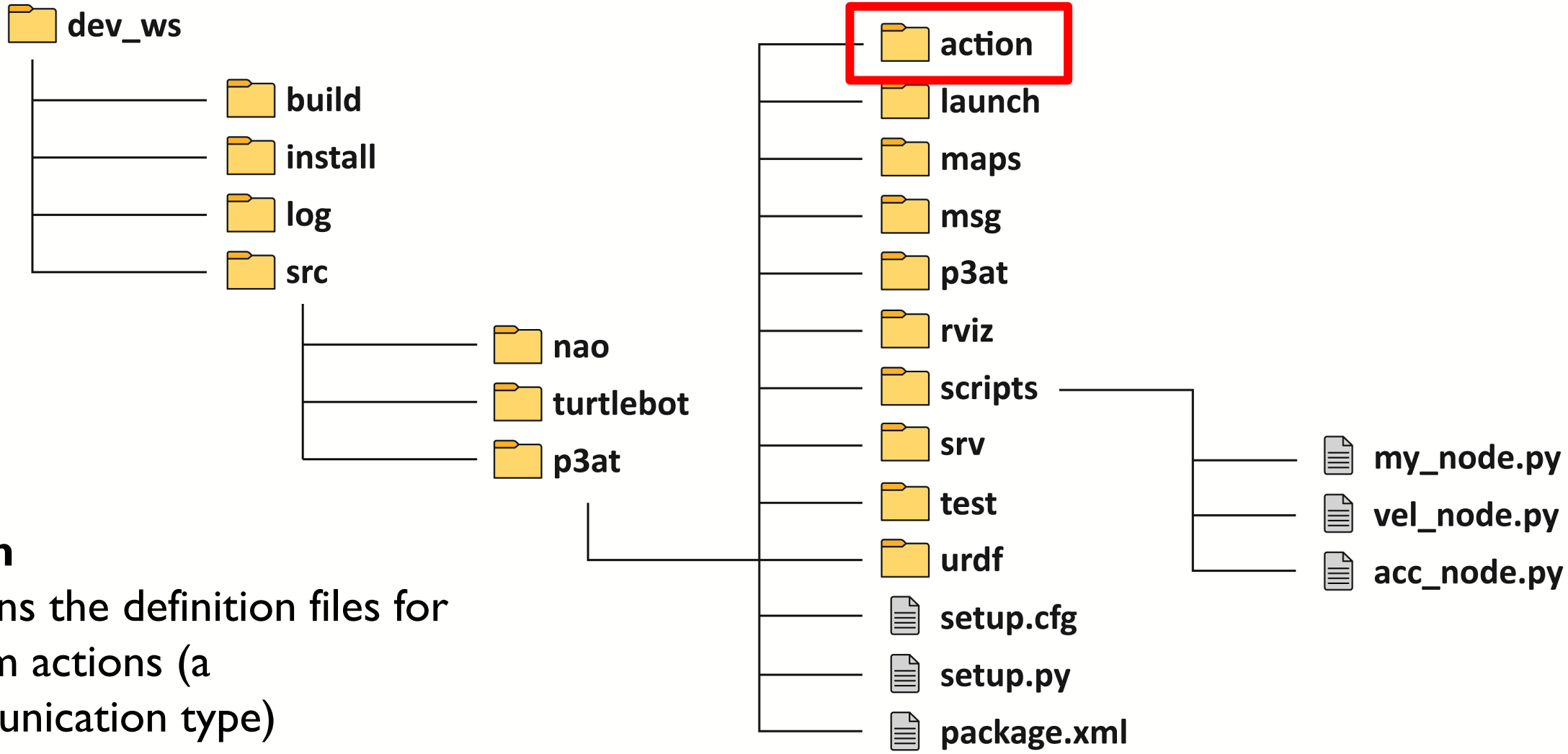

**log**
contains the log files

# ROS Filesystem



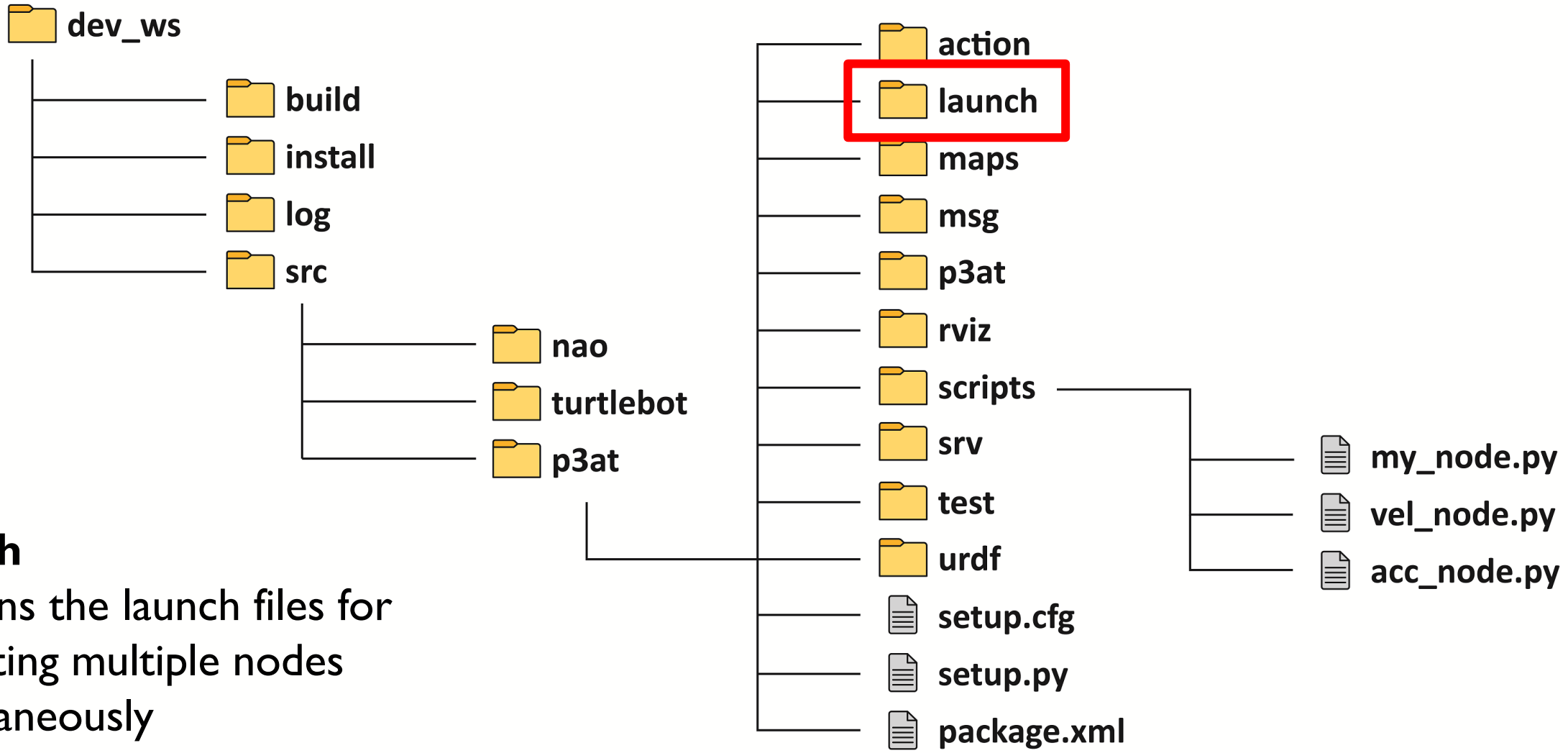

**src**
contains the packages in the workspace

# ROS Filesystem



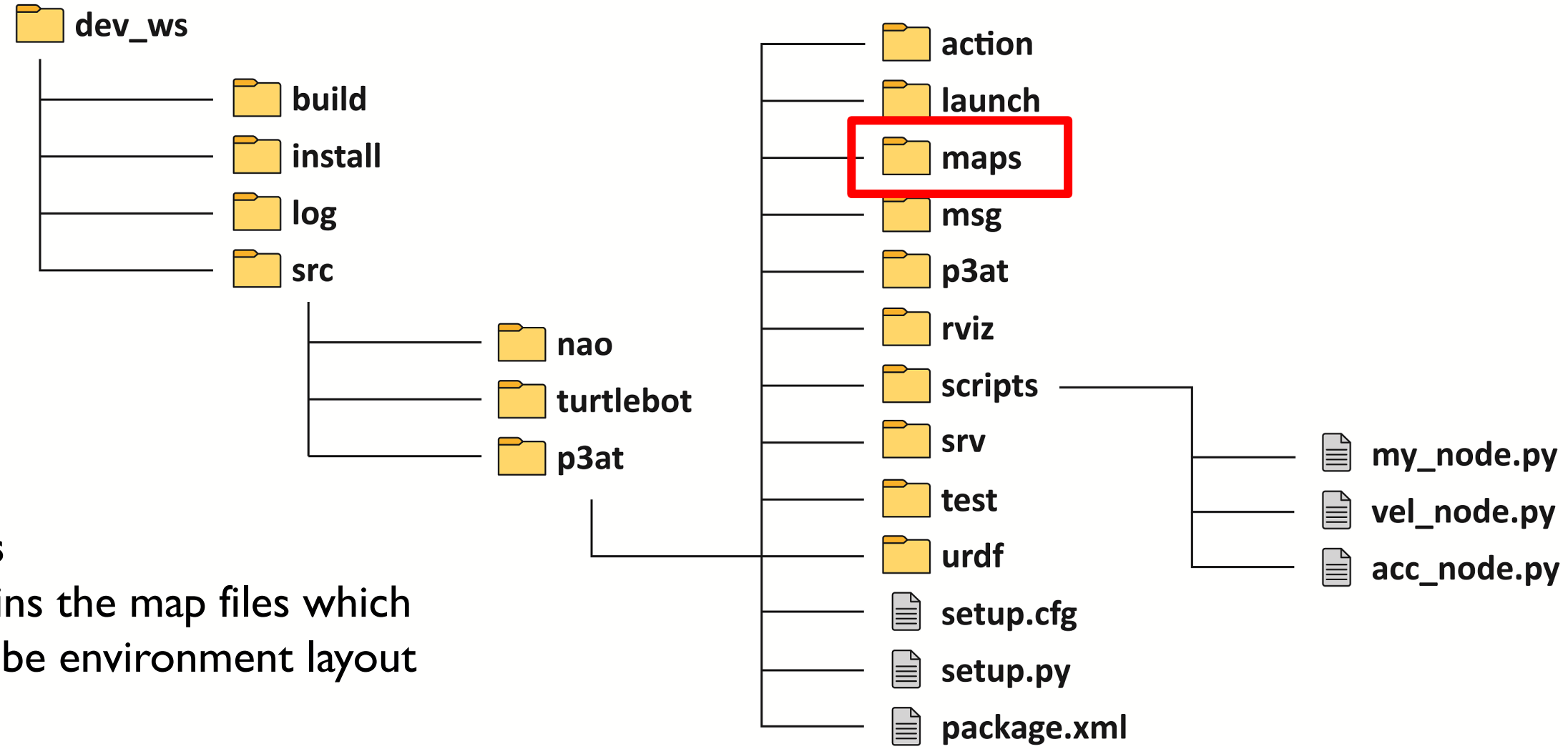

**action**
contains the definition files for custom actions (a communication type)

# ROS Filesystem



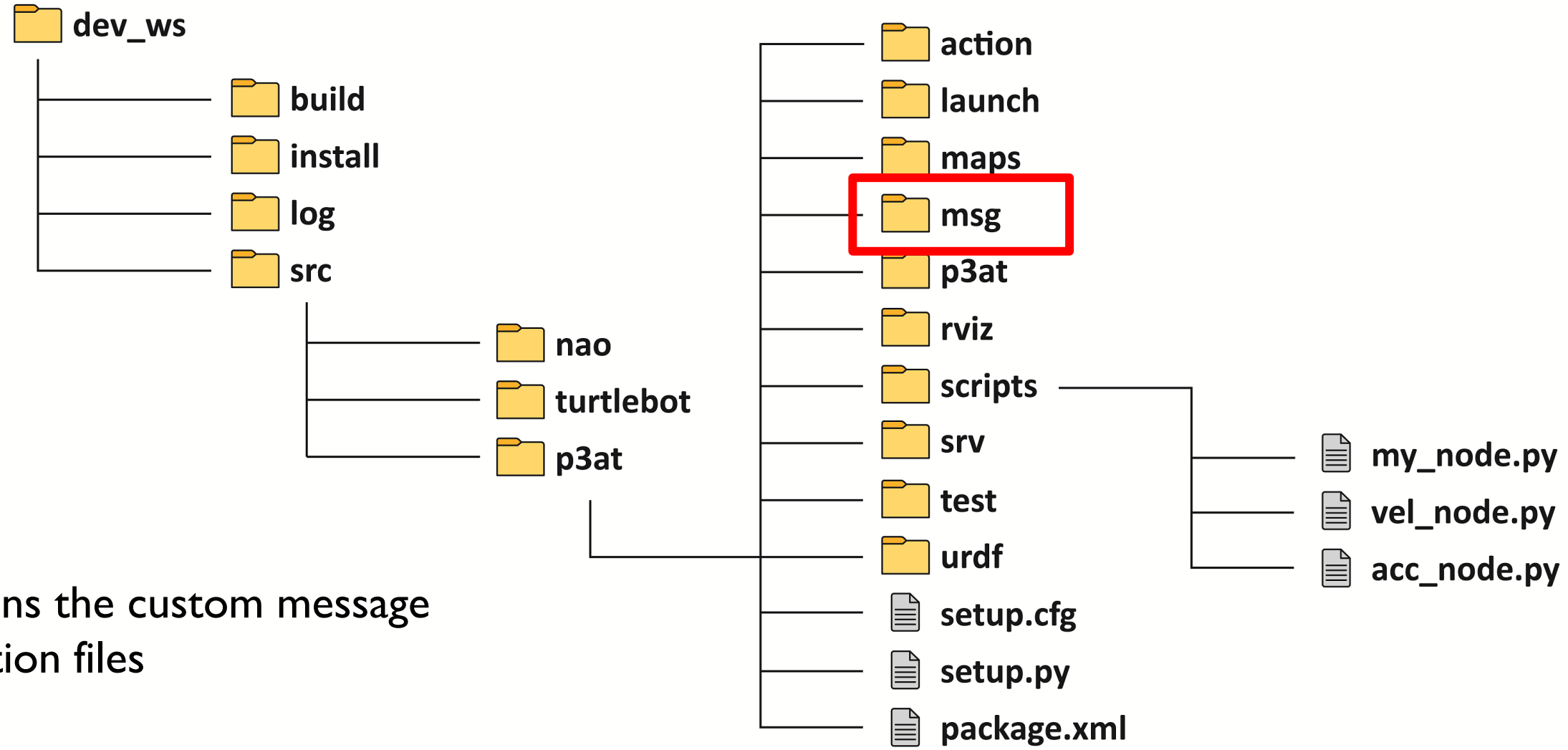

**launch**
contains the launch files for executing multiple nodes simultaneously

# ROS Filesystem



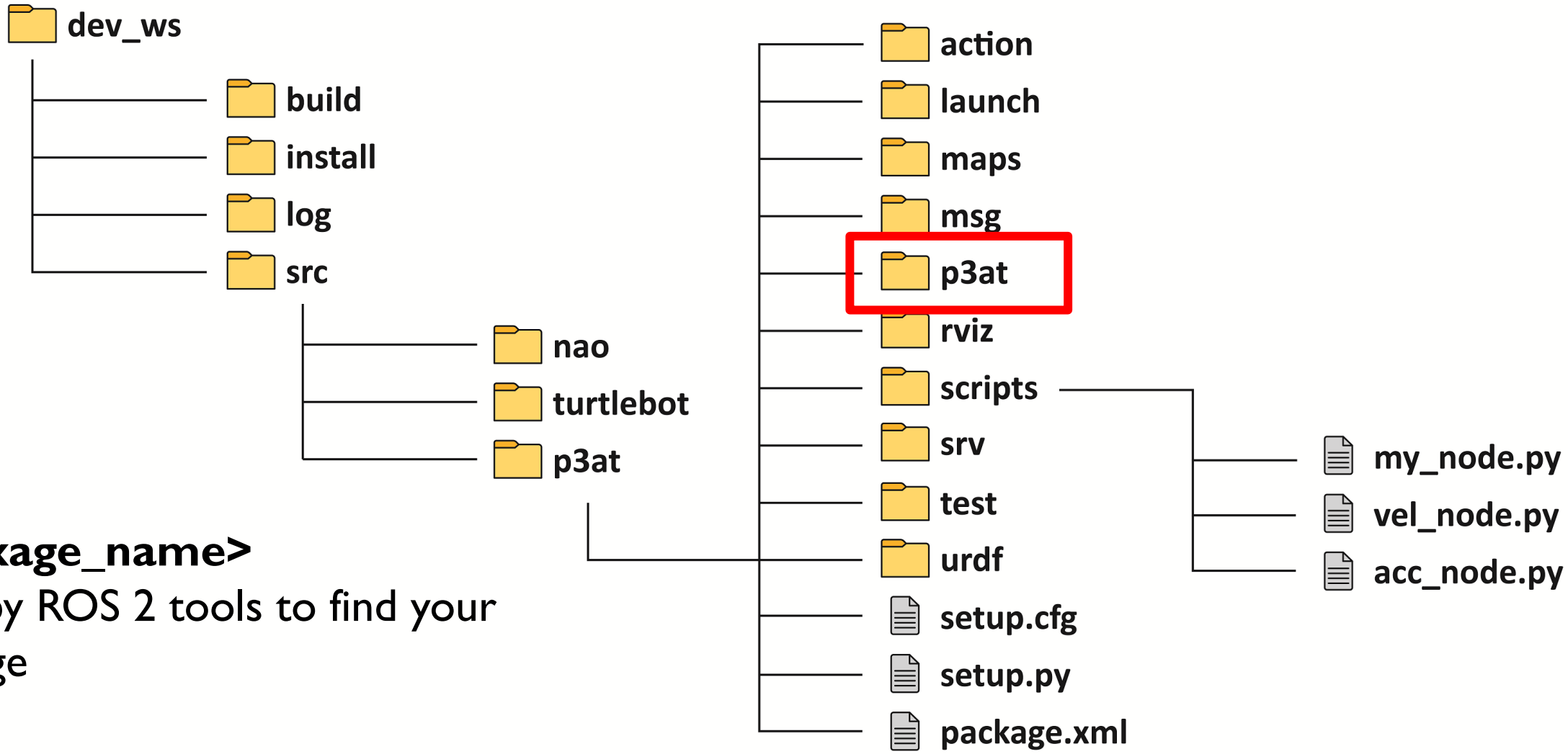

**maps**
contains the map files which describe environment layout

# ROS Filesystem



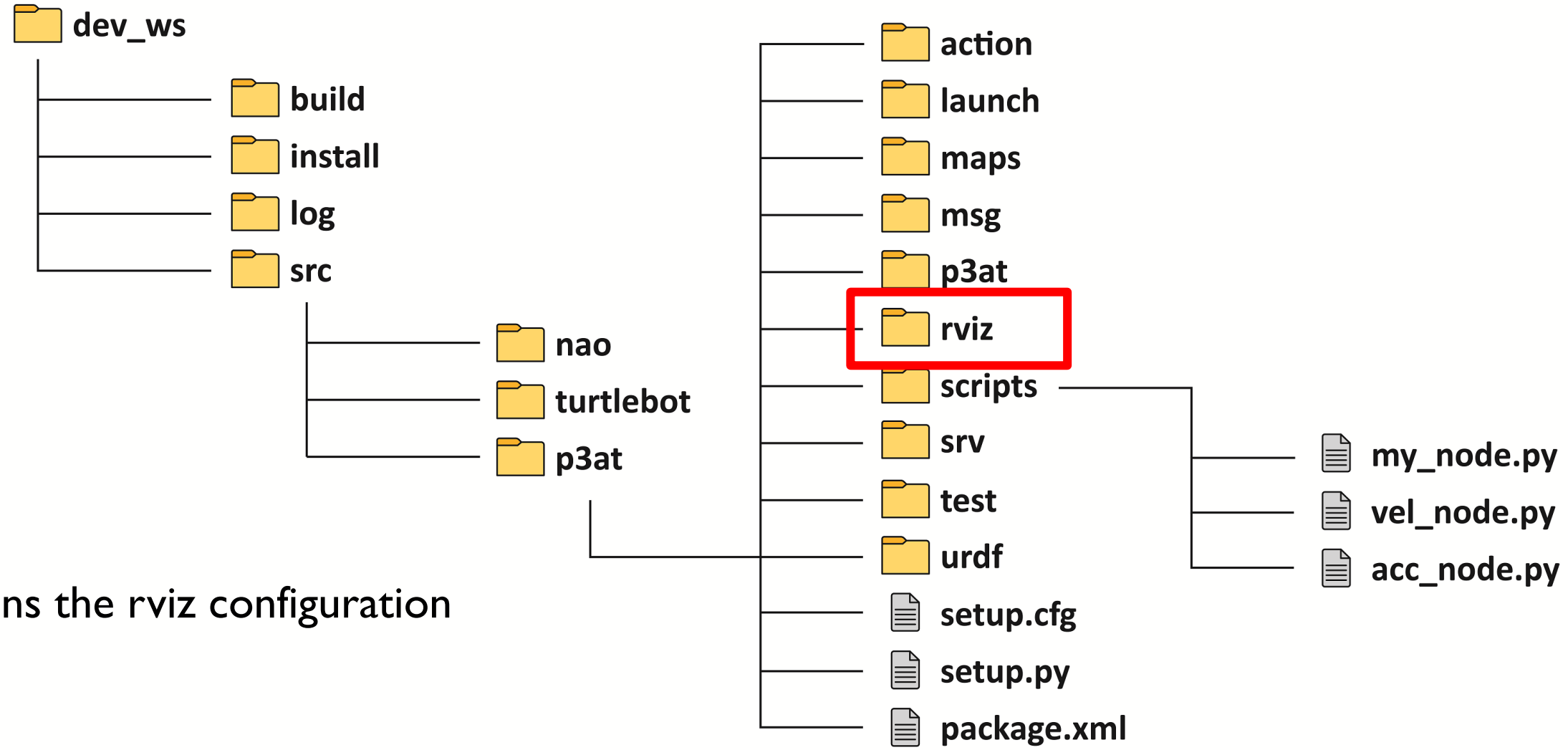

**msg**
contains the custom message definition files

# ROS Filesystem



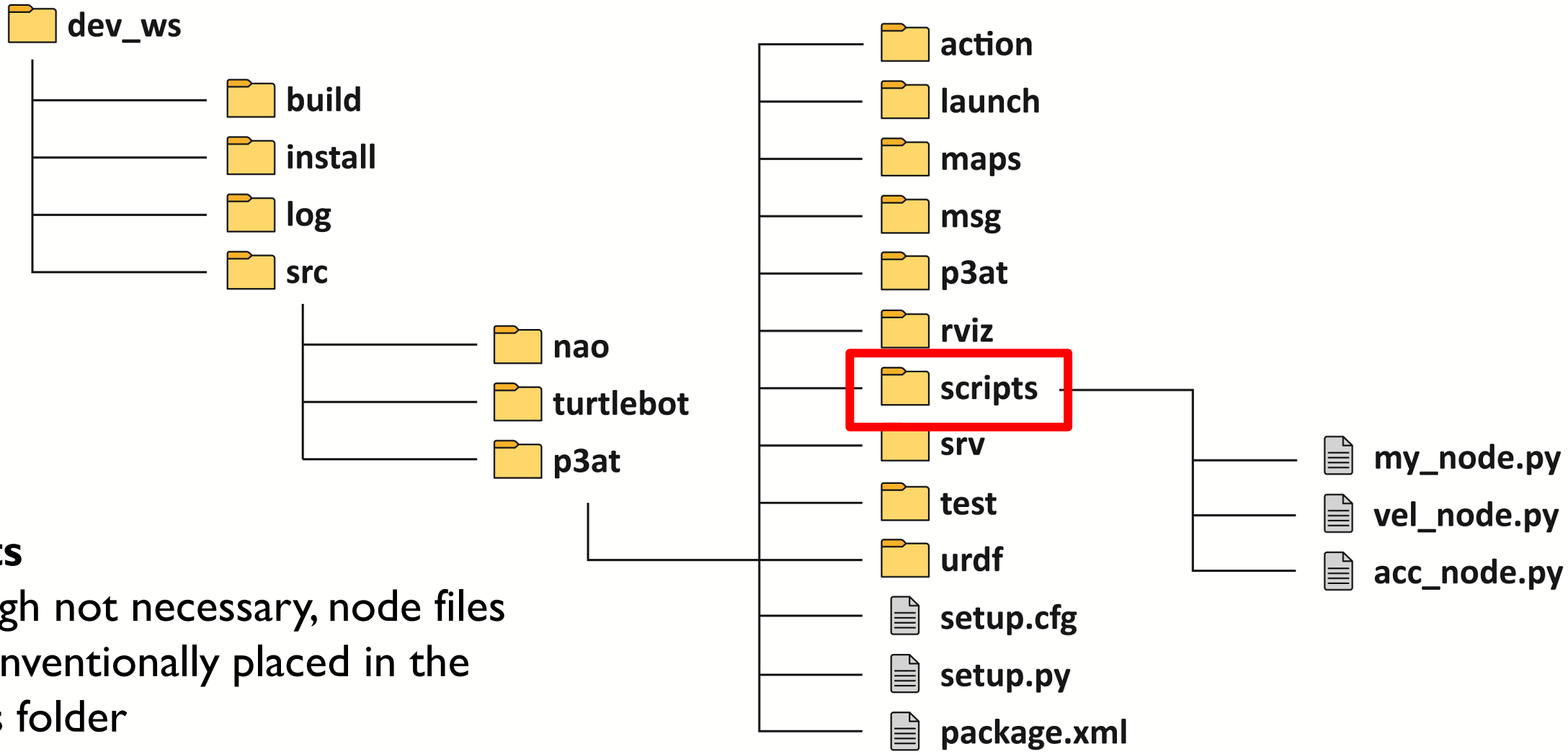

**<package_name>**
used by ROS 2 tools to find your package

# ROS Filesystem



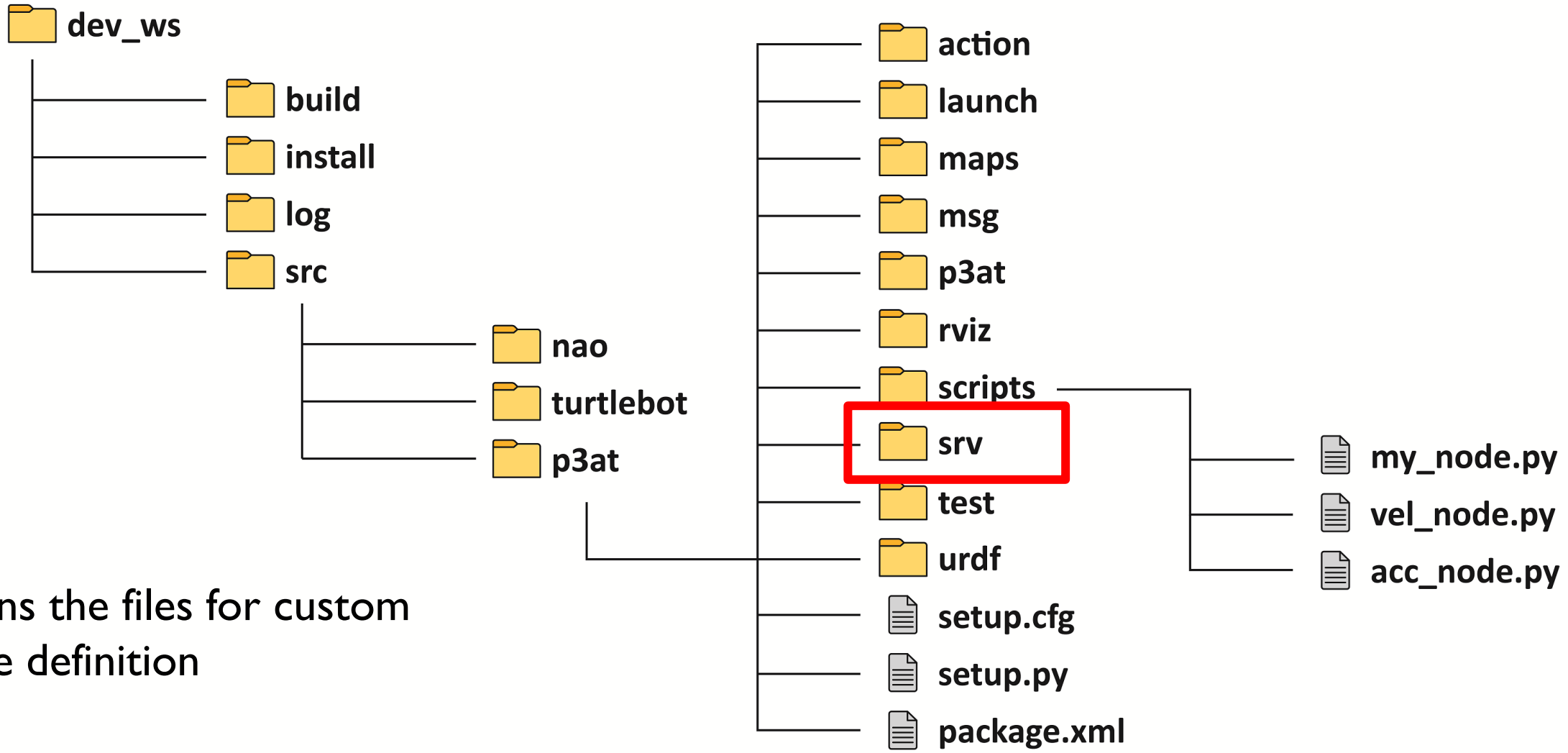

**rviz**
contains the rviz configuration files

# ROS Filesystem



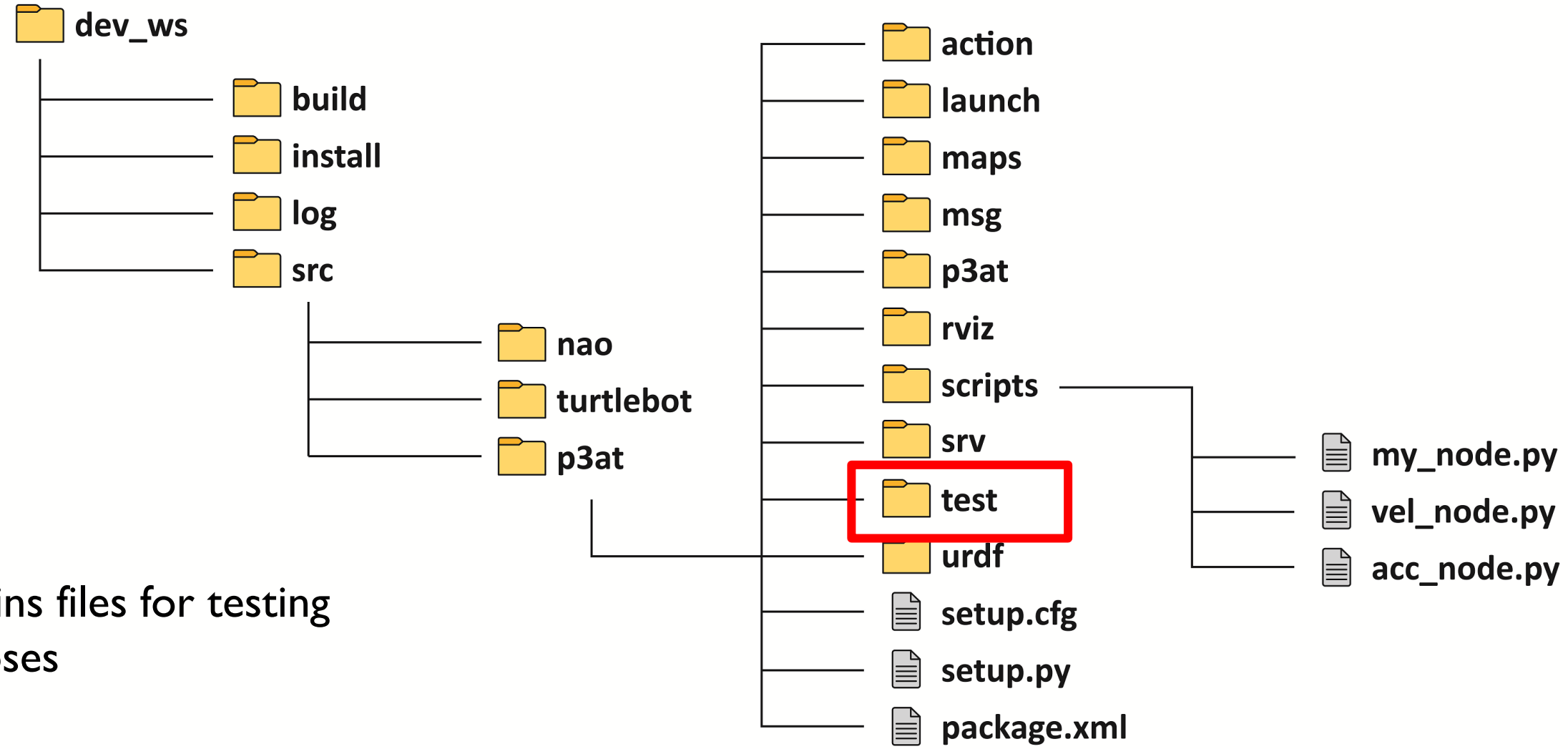

**scripts**
although not necessary, node files are conventionally placed in the scripts folder

# ROS Filesystem



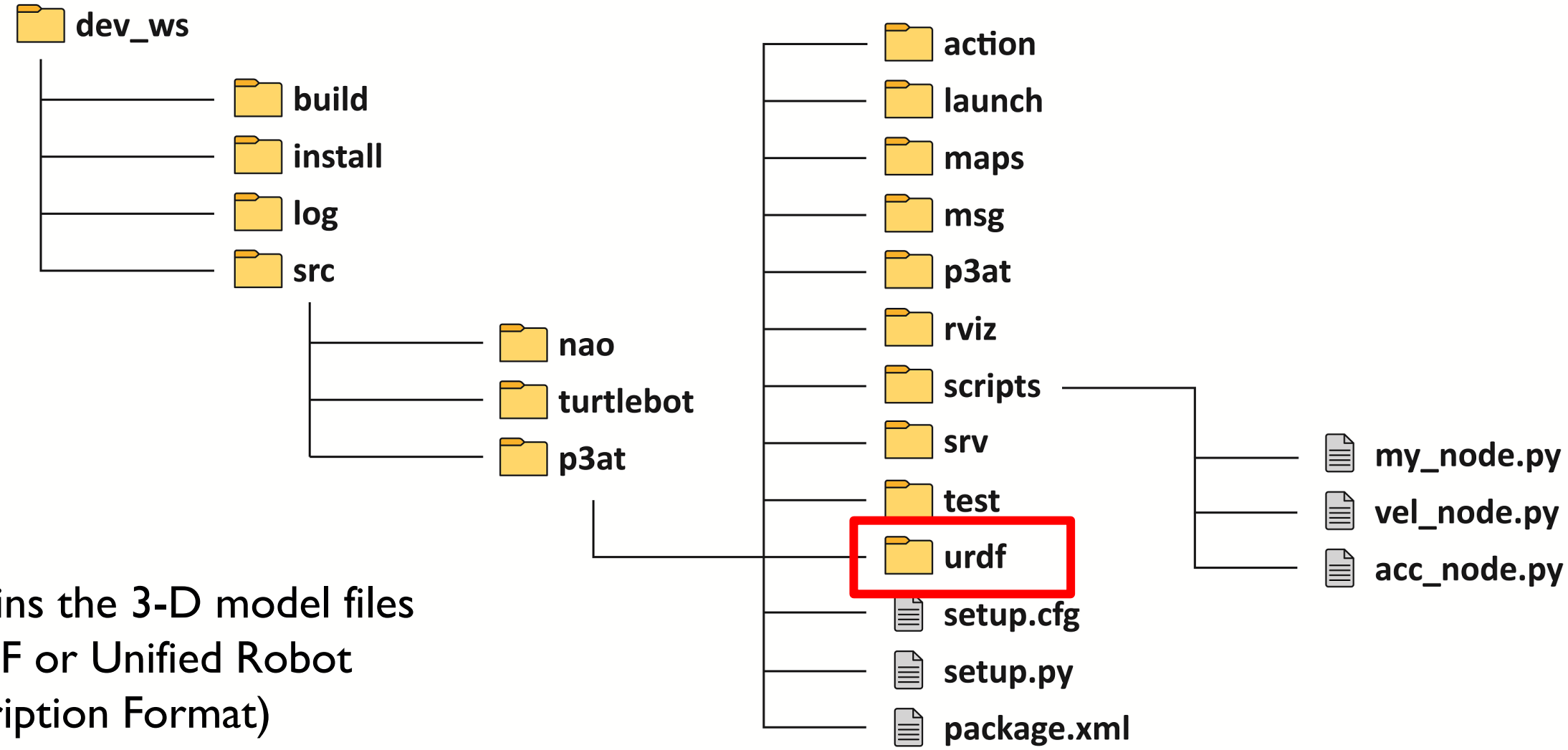

**srv**
contains the files for custom service definition

# ROS Filesystem



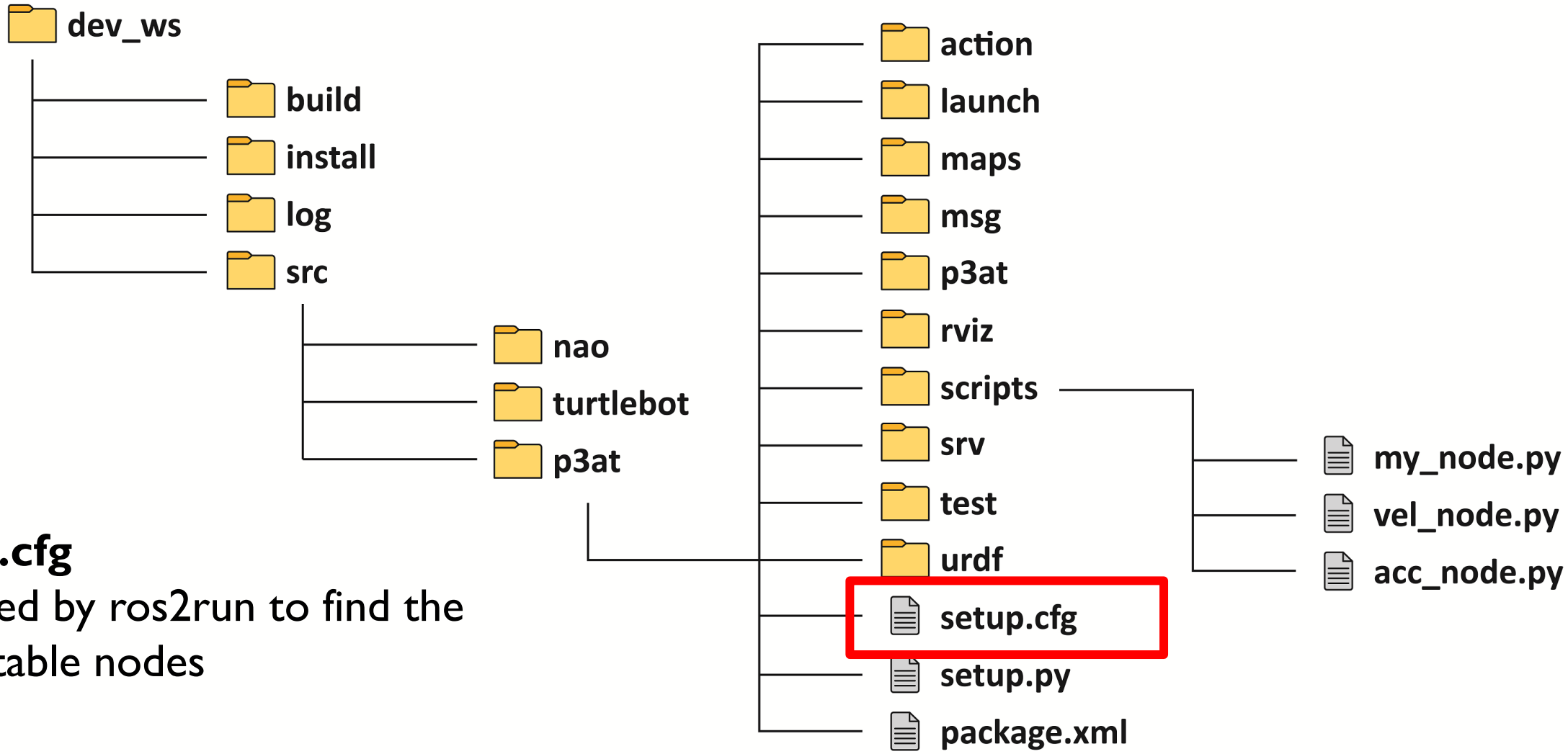

**test**
contains files for testing purposes

# ROS Filesystem



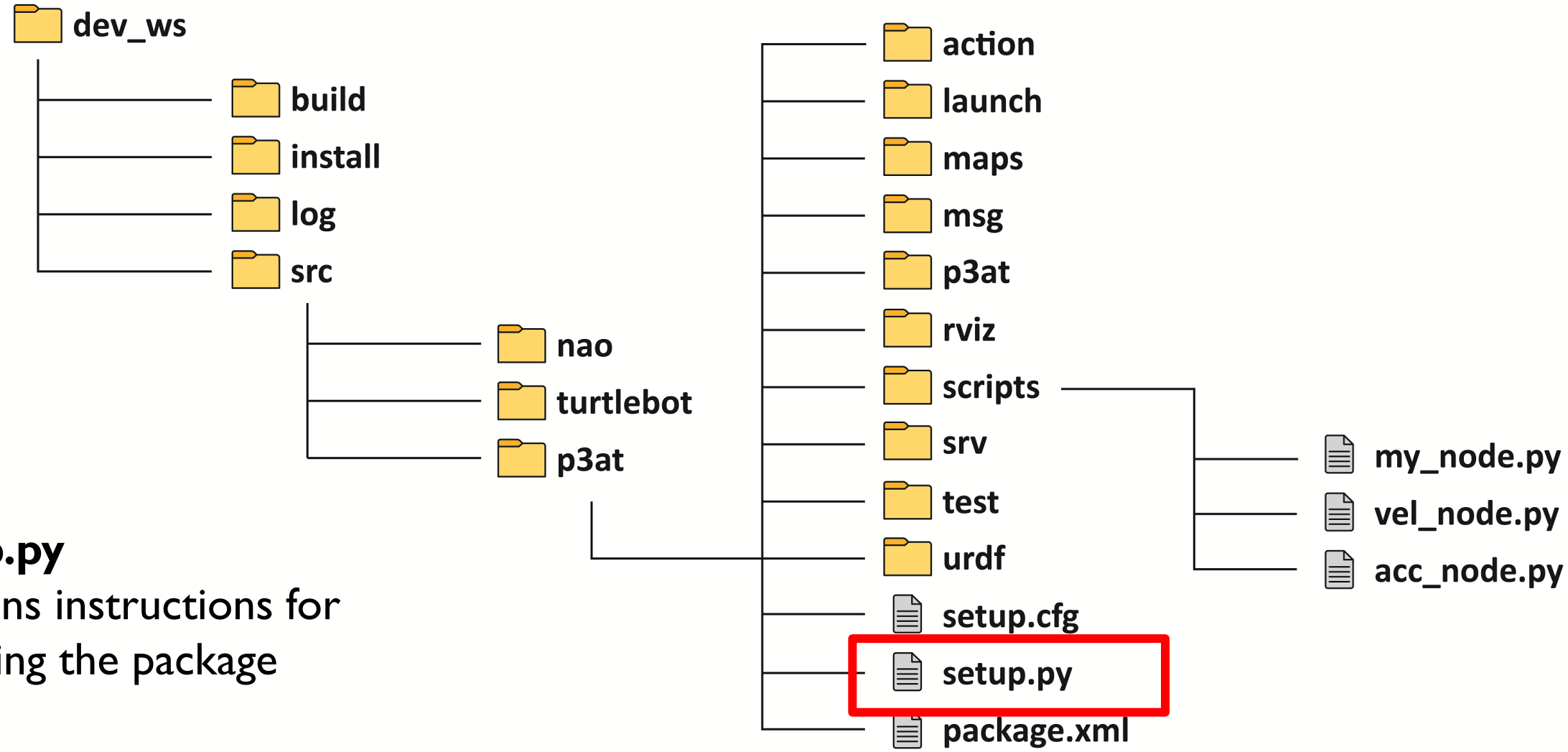

**urdf**
contains the 3-D model files (URDF or Unified Robot Description Format)

# ROS Filesystem



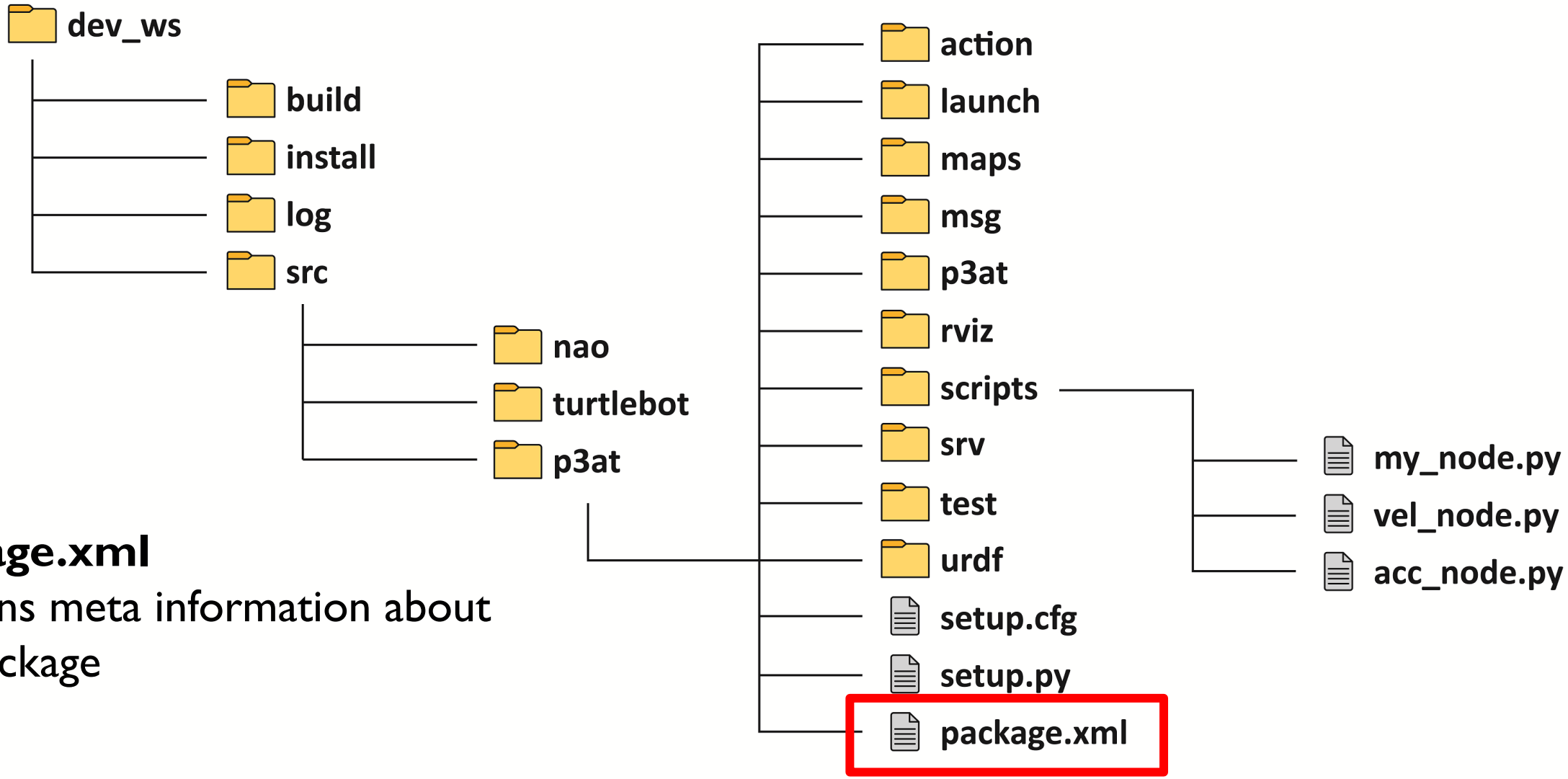

**setup.cfg**
required by ros2run to find the executable nodes

# ROS Filesystem




**setup.py**
contains instructions for installing the package

# ROS Filesystem




**package.xml**
contains meta information about the package

# Nodes

- Each ROS package contains the nodes which are scripts written in Python or C++
- Each node serves a modular objective for robot
- Nodes communicate by passing messages to other nodes (via topics, services or actions)
- When using ROS 2, we assign a ROS Domain ID. Nodes which share the same ROS Domain ID can freely communicate with each other
- To run a single node (after building), we can use the **ros2 run** command:

```
ros2 run <package_name> <node_name>
```

- To run multiple nodes simultaneously, we can use the **ros2 launch** command:

```
ros2 launch <package_name> <launch_file>
```

# Lab Tasks

- Download the manual from LMS
- Perform the Lab Tasks as given in the manual and submit it on LMS
- Remember to execute scripts with the terminal