NOTICE OF CONFIDENTIALITY AND USE RESTRICTIONS

This PowerPoint presentation, including all of its contents, is the intellectual property of Pi Thanacha Choopojcharoen. It has been exclusively designed and created for the "RobotDevOps" class.

Authorized Users:

If you are currently enrolled in or have previously taken the "RobotDevOps" class and are in possession of this presentation, you acknowledge and agree to the following:

a) This presentation is provided to you solely for your personal reference and educational use related to the "RobotDevOps" class.

b) You do not have the right to reproduce, distribute, share, or disseminate this presentation, in whole or in part, to any third parties, especially those who have not registered for the "RobotDevOps" class, unless explicitly authorized by Pi Thanacha Choopojcharoen.

c) Any unauthorized use or distribution of this presentation will result in legal action and potential academic sanctions.

Unauthorized Possession:

a) If you have come into possession of this presentation but have not registered for the "RobotDevOps" class: You must delete this presentation immediately.

b) You do not have any rights to view, use, copy, or distribute this presentation.

c) Any unauthorized retention or use will result in legal action.

Your respect for intellectual property rights ensures a fair academic and professional environment for all. Violation of these terms is a breach of trust and may have serious legal and academic consequences.

RBE502: RobotDevOps

ROS2 Launch Engineering

by

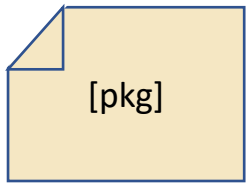
Pi Thanacha Choopojcharoen

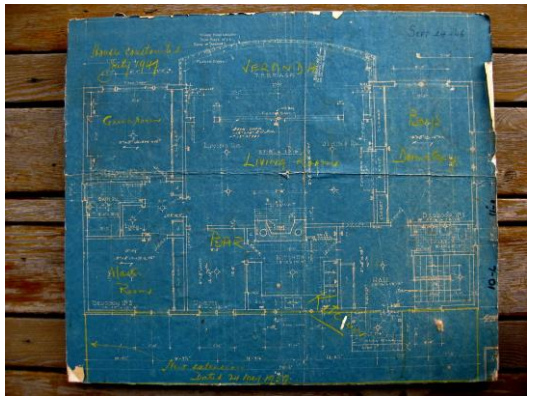
Agenda

- package structure (review)
- namespace
- parameters
- python argument/ argprase
- launch scripts & launch action
- launch arguments & launch configuration
- deserialization w/ YAML
- opaque function
- scheduling
- launch in launch

ROS2 Package Customization

The concept of Package

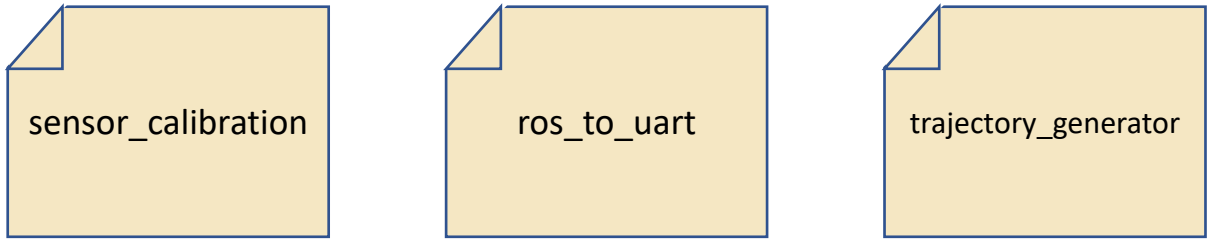




Package is a collection of organized files in a form of directory, which will used to "synthesize" nodes, launch system, etc. (usually for a specific task)

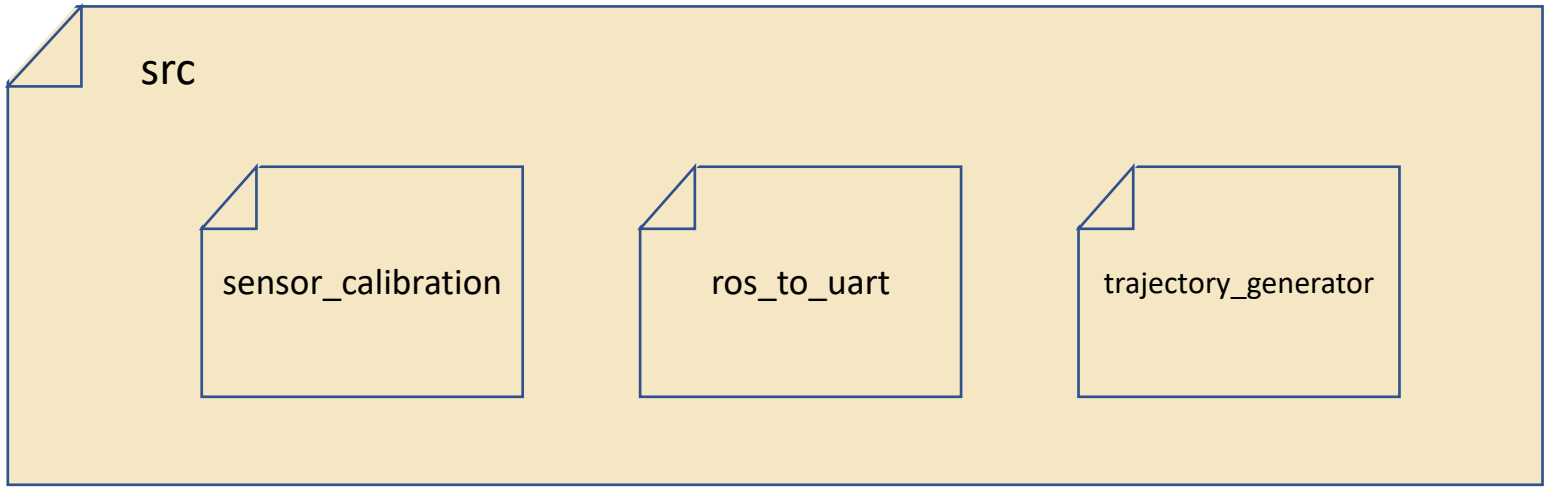
- An anology of a package would be a folder where we can keep all blue prints of a house but not the house itself.

The concept of Package



But, where do we keep our packages ?

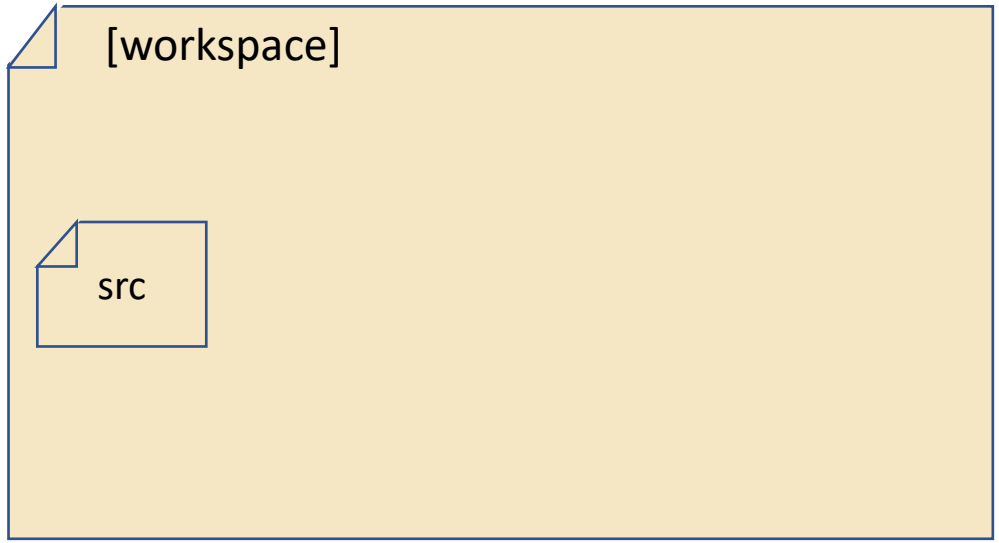
Source folder (src)



"src" directory is where we put all custom packages together.

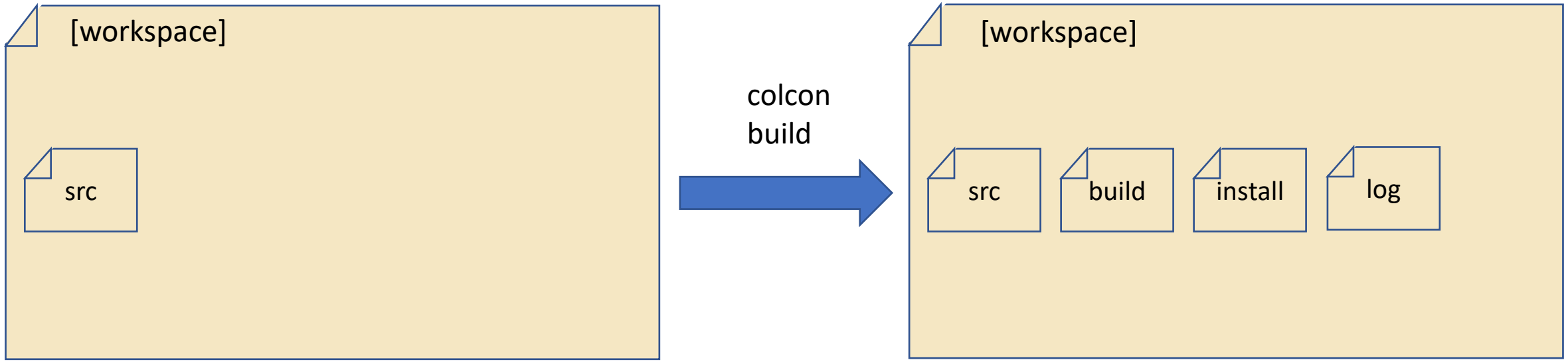
A package can be put inside another folder, which can be referred as a meta-package.

The concept of Workspace



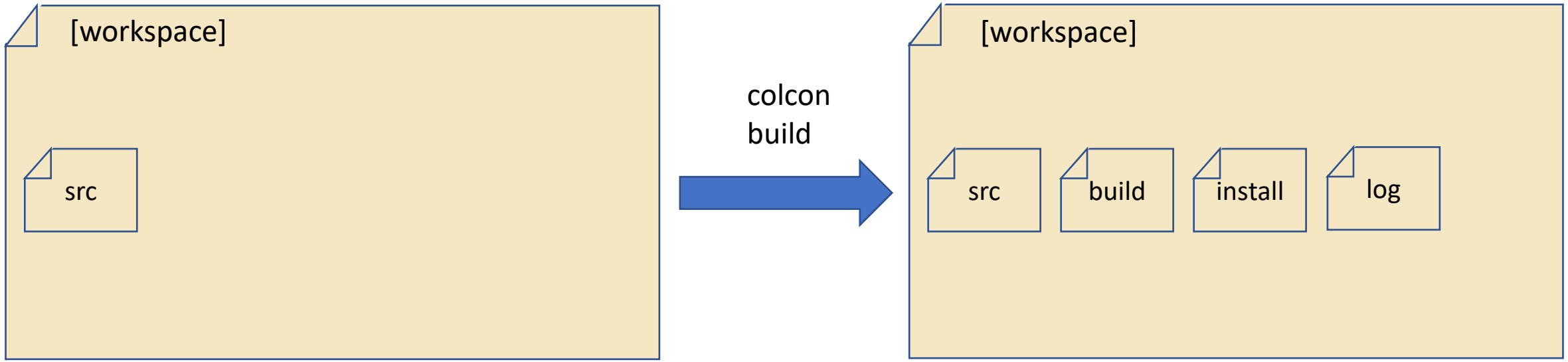
[ROS2] workspace is where we put every custom packages for a project.

Build System



"colcon build" command will build every packages in the source directory and generate 3 additional directories. When modify "src", always re-build these 3 directories. (with an exception of using symlink install)

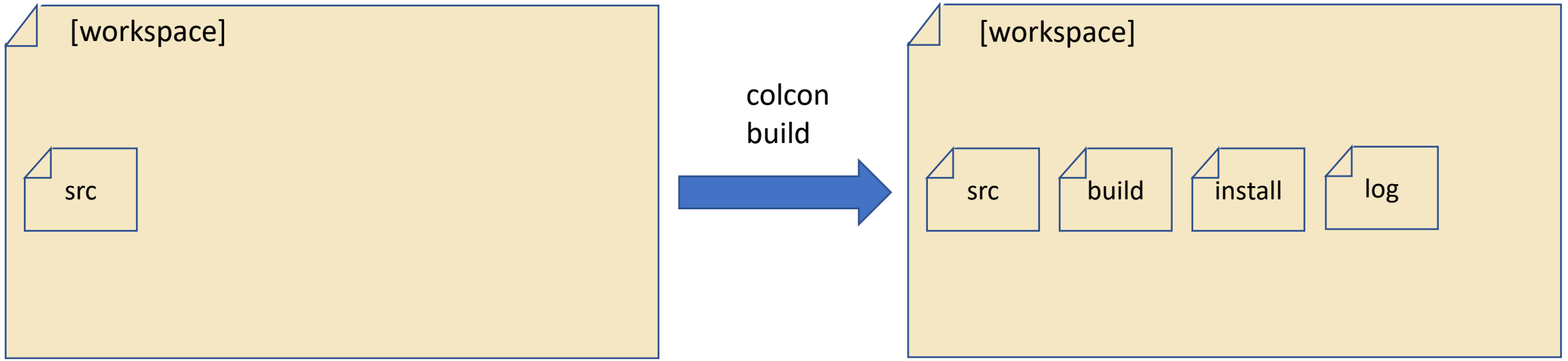
Build System



"Install" directory is the location of codes that will be used by the ROS2 system. Therefore, only modifying "src" will not change the behavior of your system in run time.

Build System

You must not create workspace inside another workspace



"Install" directory is the location of codes that will be used by the ROS2 system. Therefore, only modifying "src" will not change the behavior of your system in run time.

Creating a new workspace

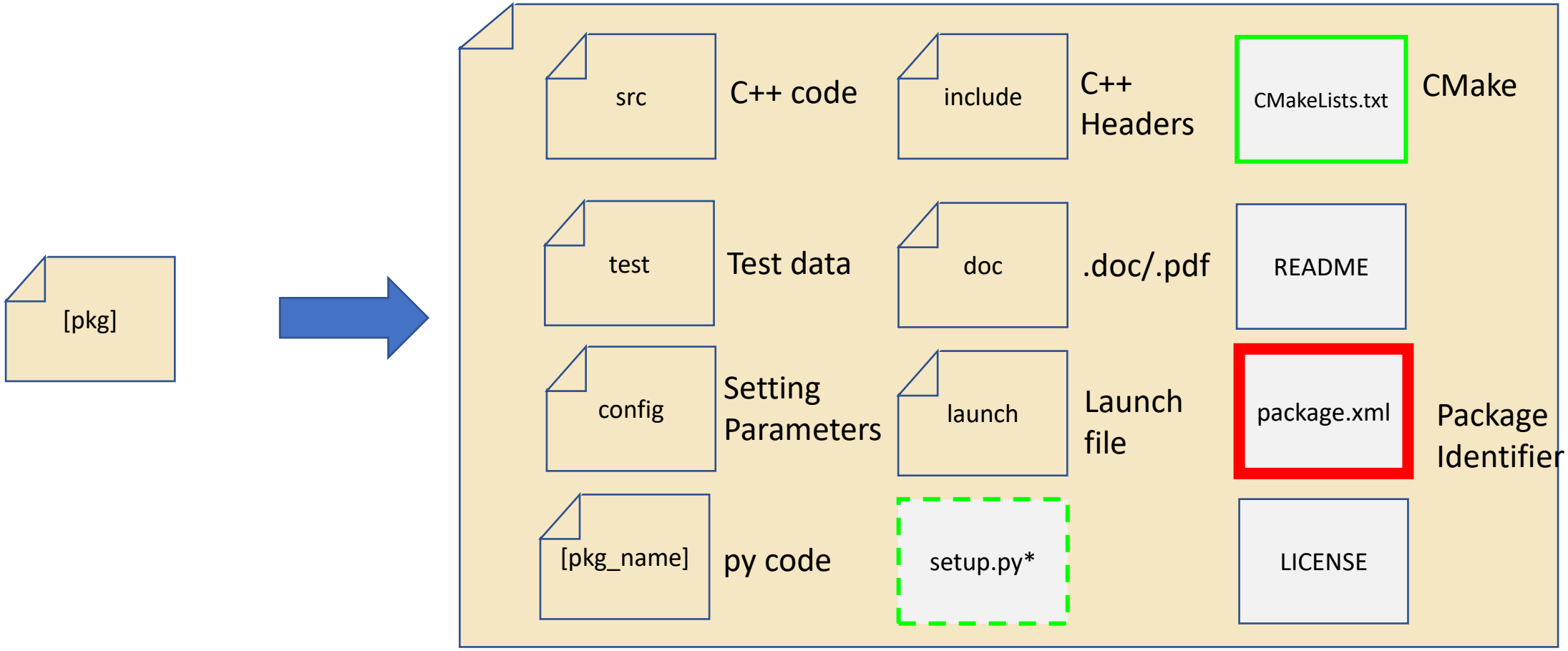
Create and build a new workspace

```
>> mkdir –p ~/[xxx]_ws/src
>> cd ~/[xxx]_ws
>> colcon build
>> source install/setup.bash
```

Adding workspace to .bashrc

```
source ~/[xxx]_ws/install/local_setup.bash
```

Package Layout



https://automaticaddison.com/organizing-files-and-folders-inside-a-ros-2-package/

Exercise

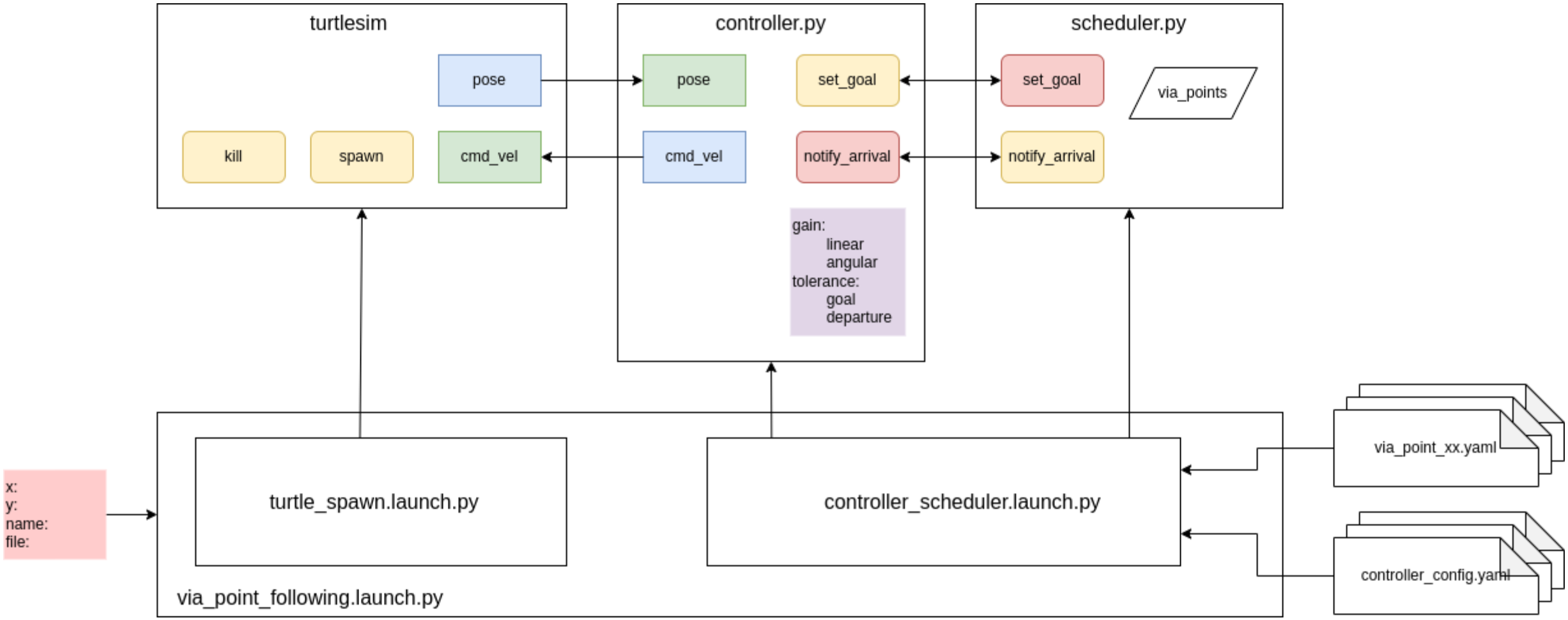
Example Repo- Branch: launch-exercise

https://github.com/kittinook/FRA501/tree/launch-exercise

Download & add “turtlesim_control” & “turtlesim_interfaces” to the “src” directory of your workspace. Then build these packages.

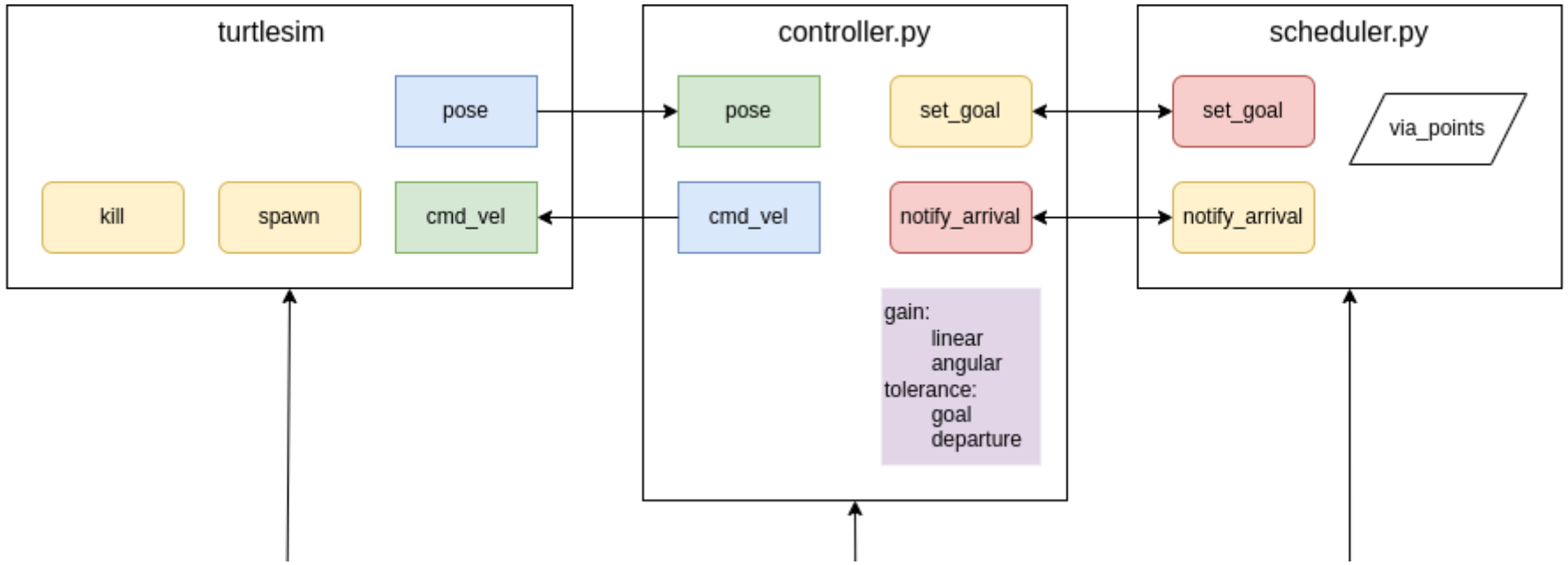
Follow the instruction on README.md

Example



Exercise

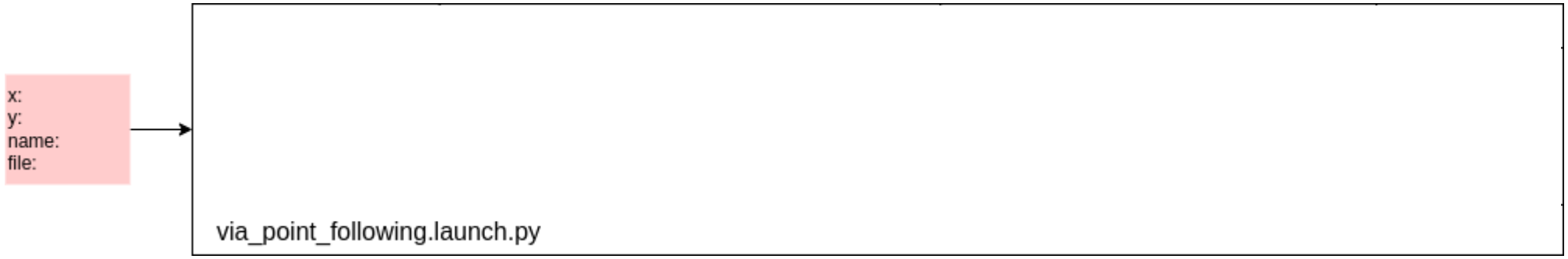
Goal 1: Launch multiple nodes



Launch turtlesim_node, controller.py, and scheduler.py with proper namespace, and executable arguments.

Exercise

Goal 2: Accepting Launch Arguments

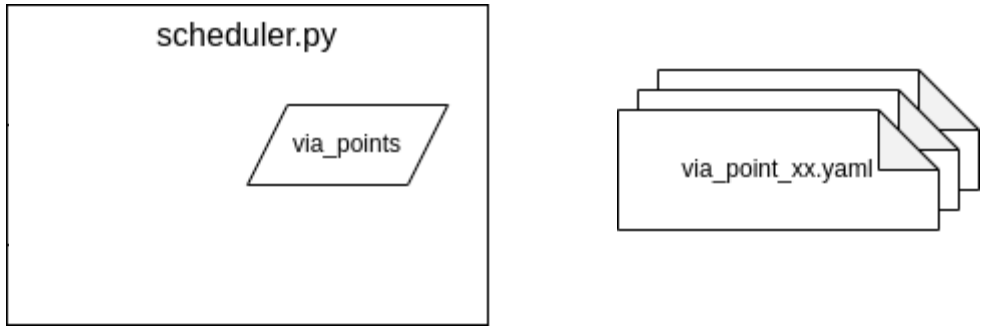


The Launch script must accept 4 optional arguments:

- x: spawn location in x-direction
- y: spawn location in y-direction
- name: name of the turtle
- file: path of the via point file (relative to the ‘via point’ directory of the package)

Exercise

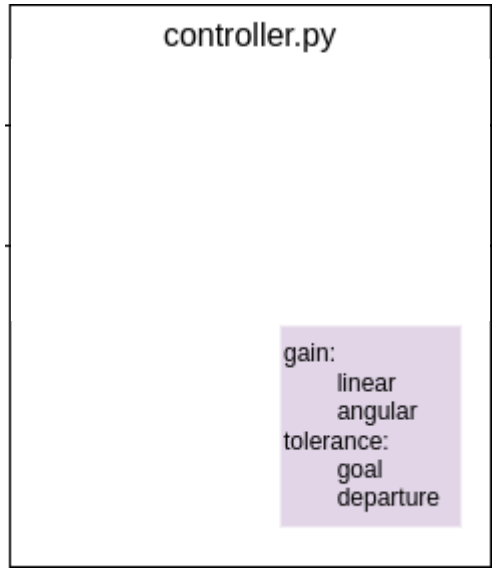
Goal 3: Pass parameters via a YAML file

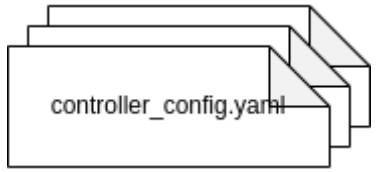


Pass the full path to the via point file in the "via_point" directory of the package based on the given (relative) file name to the scheduler

Exercise

Goal 4: Creating a new YAML file





Given a “base” configuration file for a generic controller node, create a new configuration file for a fully-qualified node and update the parameters of the node using the file.

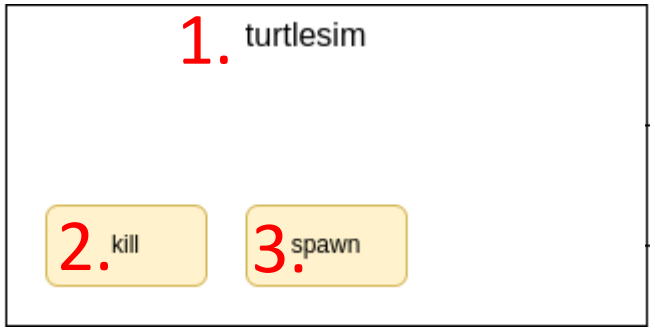
The base configuration conists of 'linear_gain','angular_gain',and 'tolerance’

The base configuration file is named “controller_config.yaml” as should look like this. [The parameters in the diagram are incorrect.]

```
controller:
   ros__parameters:
      linear_gain:1.0
      angular_gain:5.0
      tolerance:0.2
```

Exercise

Goal 5: Scheduling processes



When turtlesim_node starts, kill any existing turtle in turtlesim, then spawn a new one with the given name and location.

Exercise

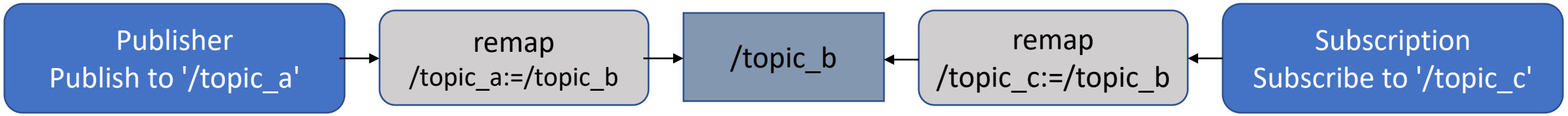
Remapping & namespace

Inconsistent topic names

Remapping Topics

Outside of our code, we can "remap" the name of the topic in the command line.

In the code, we can change the subscribed topic to "pose" instead of using "/turtle1/pose".
We can apply the same idea to other topics.

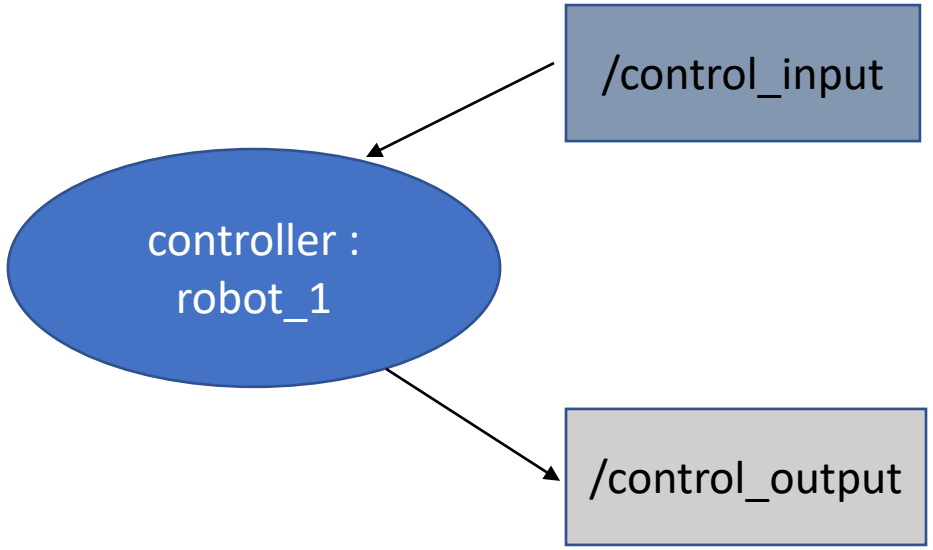


In the command line, we can add arguments at the end.

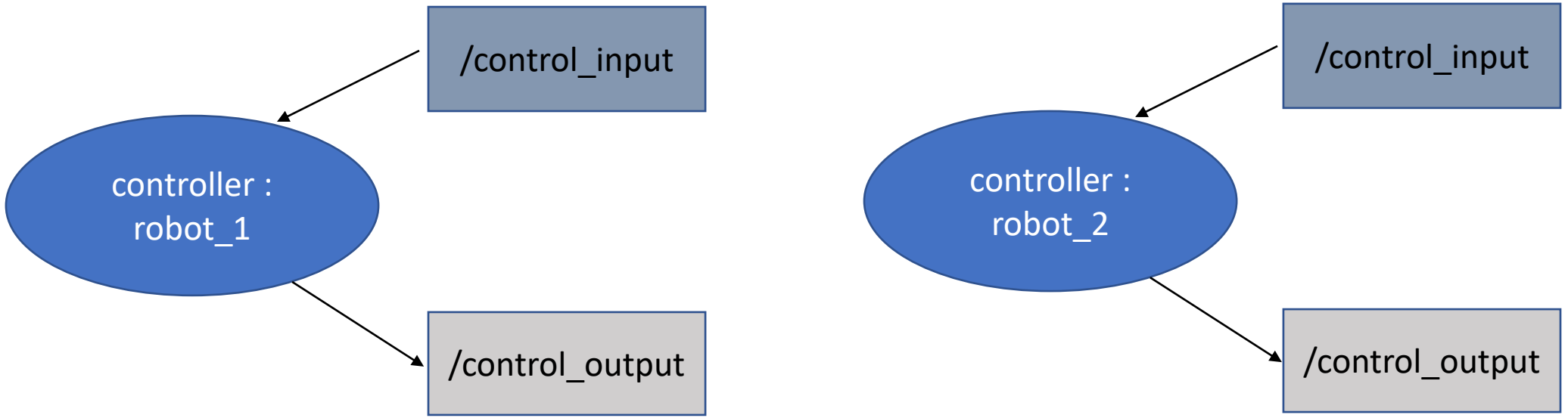
>> ros2 run turtlesim_control - -ros-args -r /pose:=/turtle1/pose

This only change the name, not the type

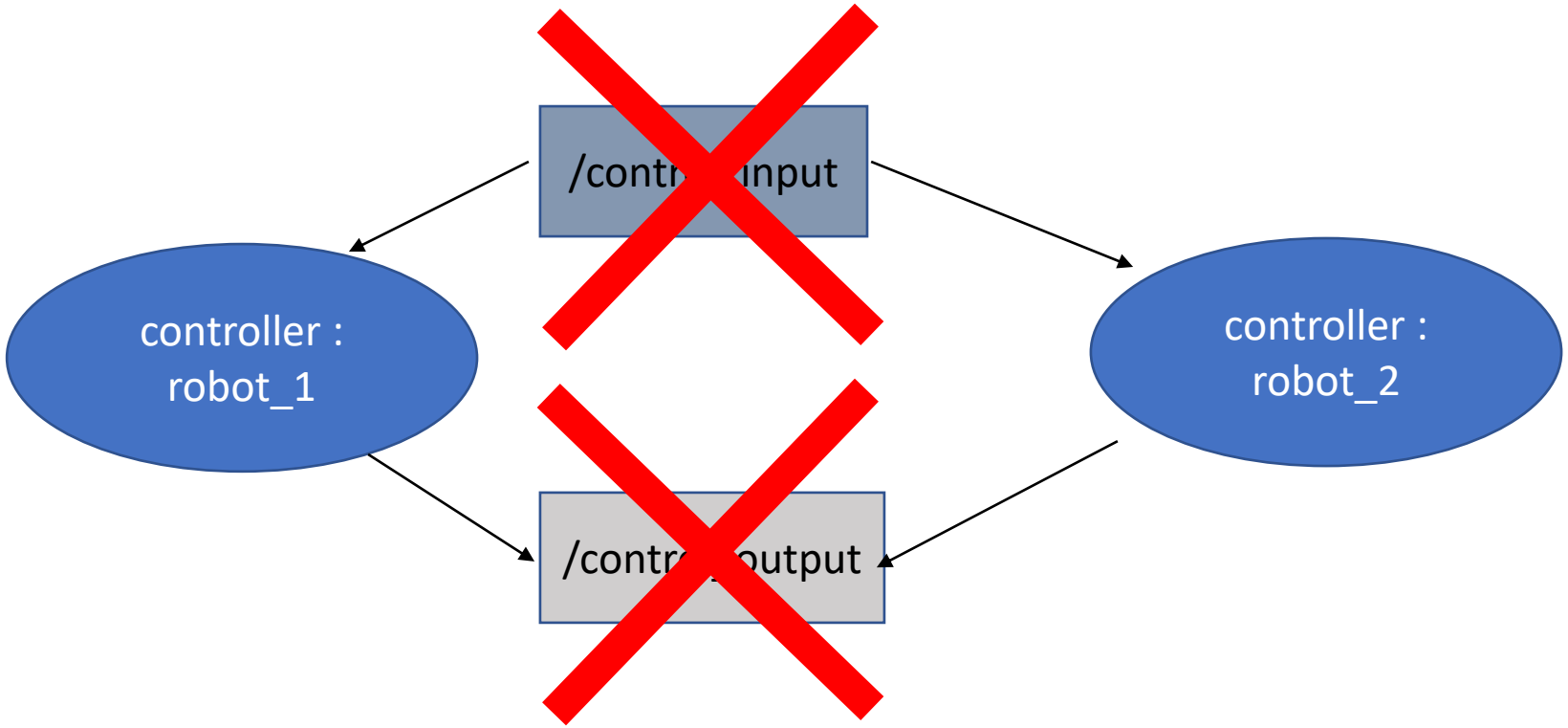
Having the same type of nodes in ROS network



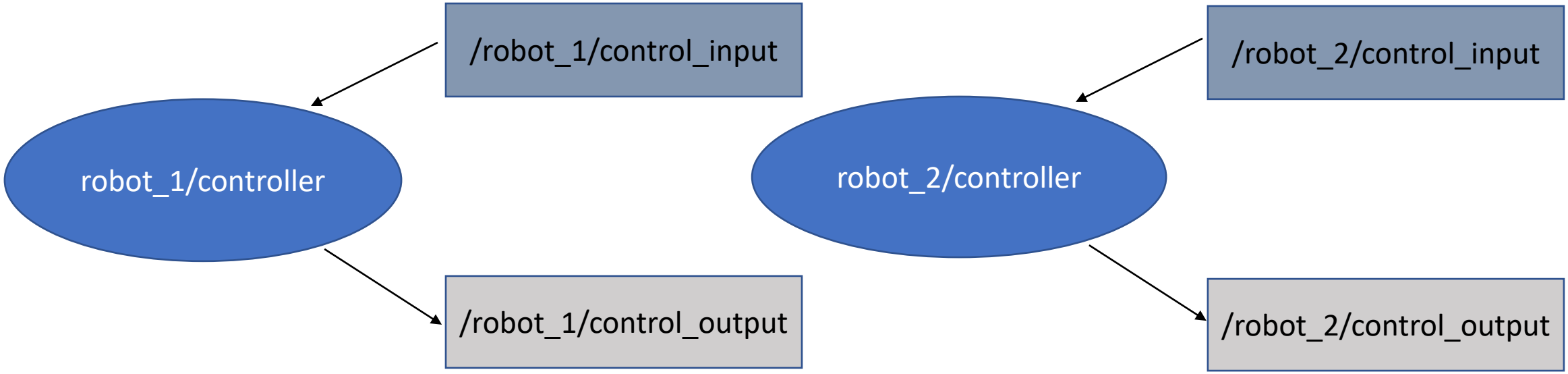
Having the same type of nodes in ROS network



Having the same type of nodes in ROS network



Distinguishing topics using namespace



Name

Absolute name refers to a name that cannot be modified by other name token.

```
Topic : /pose
```

Relative name refers to a name that can be modified by other name token.

```
Topic : pose
```

Private name refers to a name that will be automatically add its associated node in front.

```
Topic : ~/pose
```

Hidden name refers to a name that will be hidden in ROS API interface.

```
Topic : _pose
```

http://design.ros2.org/articles/topic_and_service_names.html

Namespace และ Uniform Resource Locators (URL)

Example :

```
Topic : /fleet_1/turtle_1/pose
Base name : pose
Namespace : /fleet_1/turtle_1

Topic : rosservice://turtle_1/clear
Base name : clear
Namespace : turtle_1
URL : rosservice://
```

Input Name	Node: `my_node` NS: none	Node: `my_node` NS: `/my_ns`
`ping`	`/ping`	`/my_ns/ping`
`/ping`	`/ping`	`/ping`
`~`	`/my_node`	`/my_ns/my_node`
`~/ping`	`/my_node/ping`	`/my_ns/my_node/ping`

Fully-qualified name (FQN)

http://design.ros2.org/articles/topic_and_service_names.html

Command Line for namespace

ros2 run [package] [executable] --ros-args -r __ns:=[/namespace]

For example:

>> ros2 run turtlesim_control controller.py --ros-args -r __ns:=/turtle1

Remapping topic

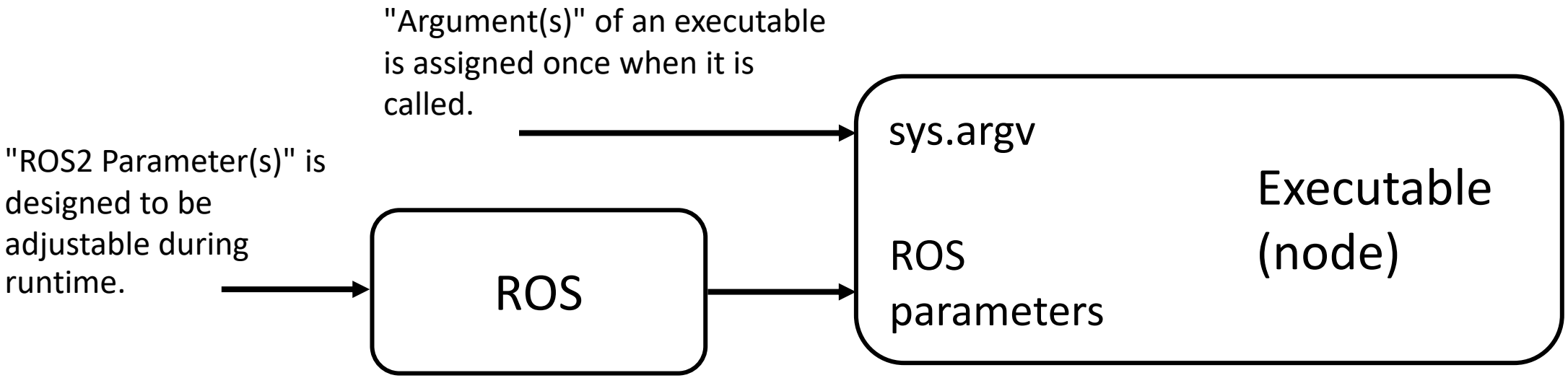
ros2 run [package] [executable] --ros-args -r __ns:=[/namespace]

For example:

>> ros2 run turtlesim_control controller.py --ros-args -r __ns:=/turtle1

Executable Arguments & ROS2 Parameters

Arguments & Parameters of a ROS Node



They are both "constants"

ROS2 Parameters in RCLPY

```
from rclpy.parameter import Parameter

class NewNode(Node):
    def __init__(self):
        super().__init__('my_node')
        self.declare_parameter('length',3.0)
        self.declare_parameter('width',4.0)
        self.declare_parameter('area',12.0)
    def  some_callback(self):
        L = self.get_parameter('length').value
        W = self.get_parameter('width').value
        p = Parameter('area',Parameter.Type.DOUBLE,L*W)
        self.set_parameters([p])
```

Instead of being a standalone value in a ROS network, ROS2 Parameters must be associated with a node.

One can declare parameters in rclpy.Node

Command Line for parameters

ros2 run [package] [executable] --ros-args -p [argument_name]:=[value]

For example:

>> ros2 run turtlesim_control controller.py --ros-args -r __ns:=/turtle1 -p angular_gain:=10.0

Command Line mixing ROS arguments with custom arguments

For example:

```
>> VP_FILE=~/[your_ws]/src/turtlesim_control/via_point/via_point_01.yaml
>> ros2 run turtlesim_control scheduler.py --ros-args -r __ns:=/turtle1 -f $VP_FILE
```

Command Line mixing ROS arguments with custom arguments

```python
import argparse

def main(args=None):

    parser = argparse.ArgumentParser(description='schedule via points')
    parser.add_argument('-f', '--file', help='path to the YAML file of via points')
    parsed_args, remaining_args = parser.parse_known_args(args=args)

    rclpy.init(args=remaining_args)
    file_path = parsed_args.file
```

Launch Script & Launch Action

Launch Files in ROS2

A (ROS2) Launch file allows a user to run a system with multiple ROS nodes or other launch files at once. One can also scheduling the system's behavior as well.

- XML
- YAML
- Python !!!!

General Launch Pipeline

Launch Action

Node

Execute Process

Define your Launch action(s)

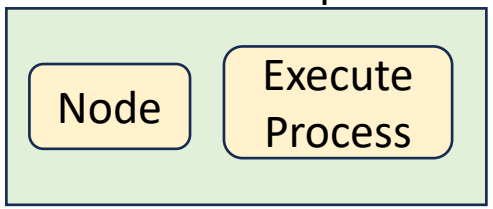
Add each Launch action to the description of your Launch

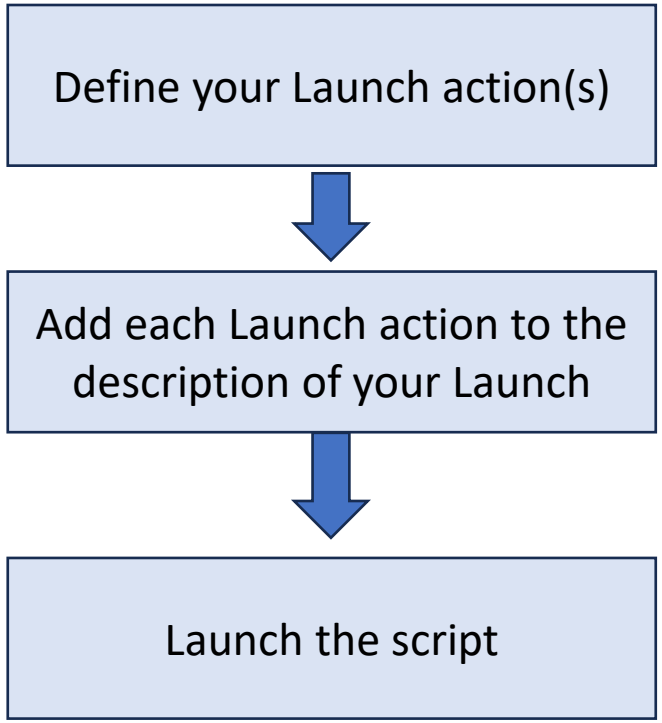
Launch the script

```python
turtlesim = Node(
    package='turtlesim',
    executable='turtlesim_node'
)
controller = Node(
    package='turtlesim_control',
    executable='controller.py'
)
scheduler = Node(
    package='turtlesim_control',
    executable='scheduler.py'
)
```

General Launch Pipeline

Launch Description






```
launch_description = LaunchDescription()
launch_description.add_action(turtlesim)
launch_description.add_action(controller)
launch_description.add_action(scheduler)
```

Not all Launch action will be launched right away.

Launch Script Structure

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    turtlesim = Node(
        package='turtlesim',
        executable='turtlesim_node'
    )
    controller = Node(
        package='turtlesim_control',
        executable='controller.py'
    )
    scheduler = Node(
        package='turtlesim_control',
        executable='scheduler.py'
    )

    launch_description = LaunchDescription()
    launch_description.add_action(turtlesim)
    launch_description.add_action(controller)
    launch_description.add_action(scheduler)
    return launch_description
```

All action must be "added" in this function.

One must not forget to add the action the launch description.

The function must return the Launch Description.

Launch Action: ROS Node

```
from launch_ros.actions import Node

node_action = Node(
    package = 'some_package',
    executable = 'something.py',
    parameters = [
        {'param_1': 1.0},
        {'param_2': 10.0}
    ],
    namespace='my_ns',
    remappings=[
        ('/old_1', '/new_1'),
        ('/old_2', '/new_2'),
    ],
    arguments= ['-f', 'some_argument']
)
```

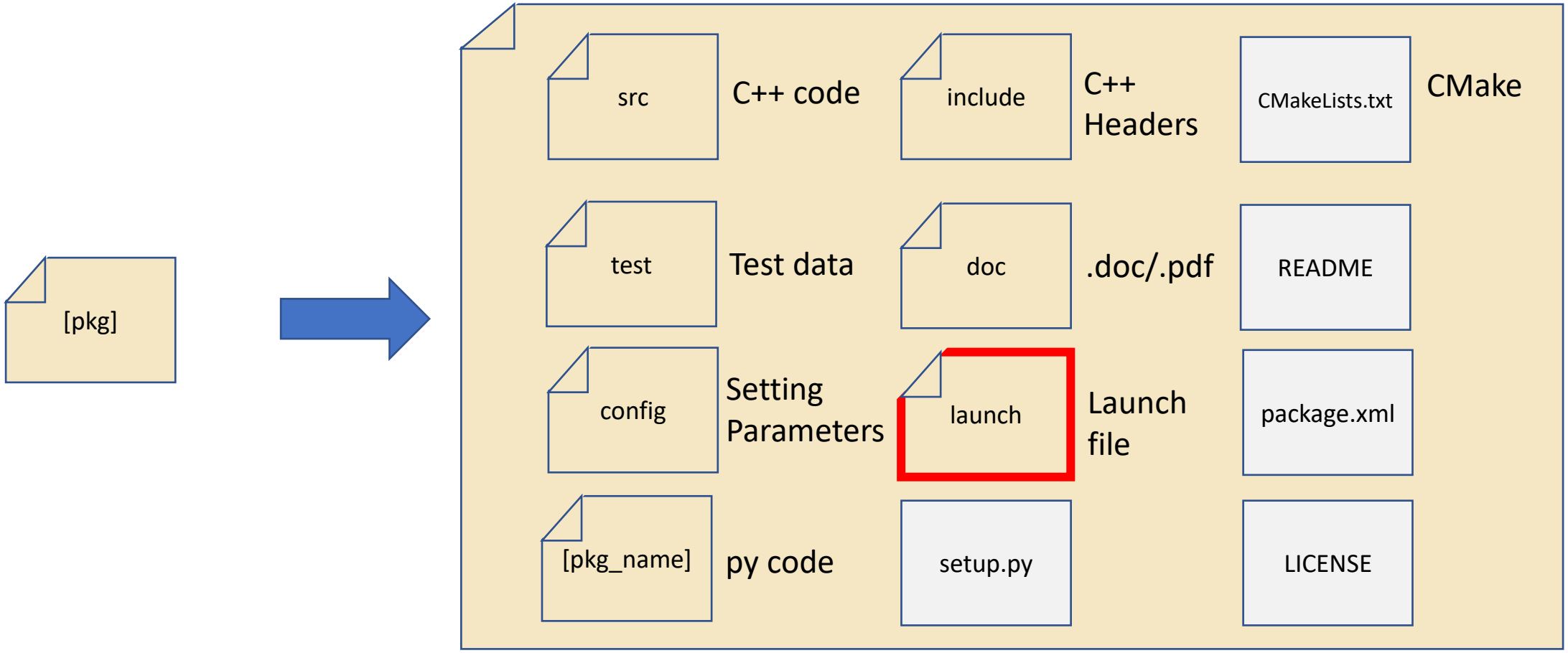
Launch Action: Executing command line

One can call a shell command in a Launch script.

```
from launch.actions import ExecuteProcess

spawn_turtle2 = ExecuteProcess(
    cmd = [['ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 2.0,
theta: 0.0, name: \'turtle2\'}"']],
    shell=True
)
```

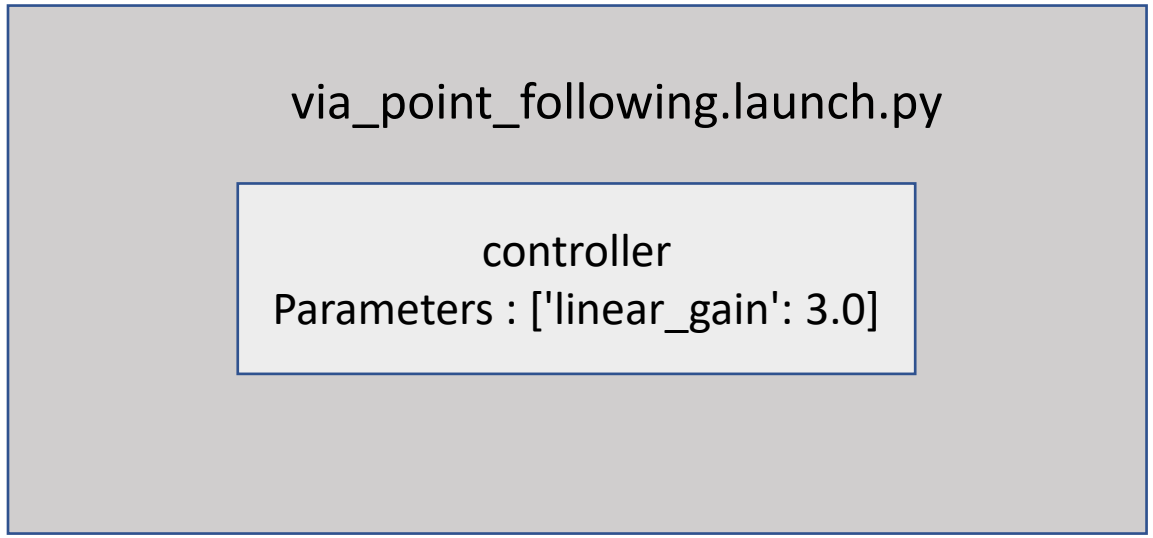
Package Layout: Launch

Don’t forget to add ‘launch’ directory to CMakeLists.txt

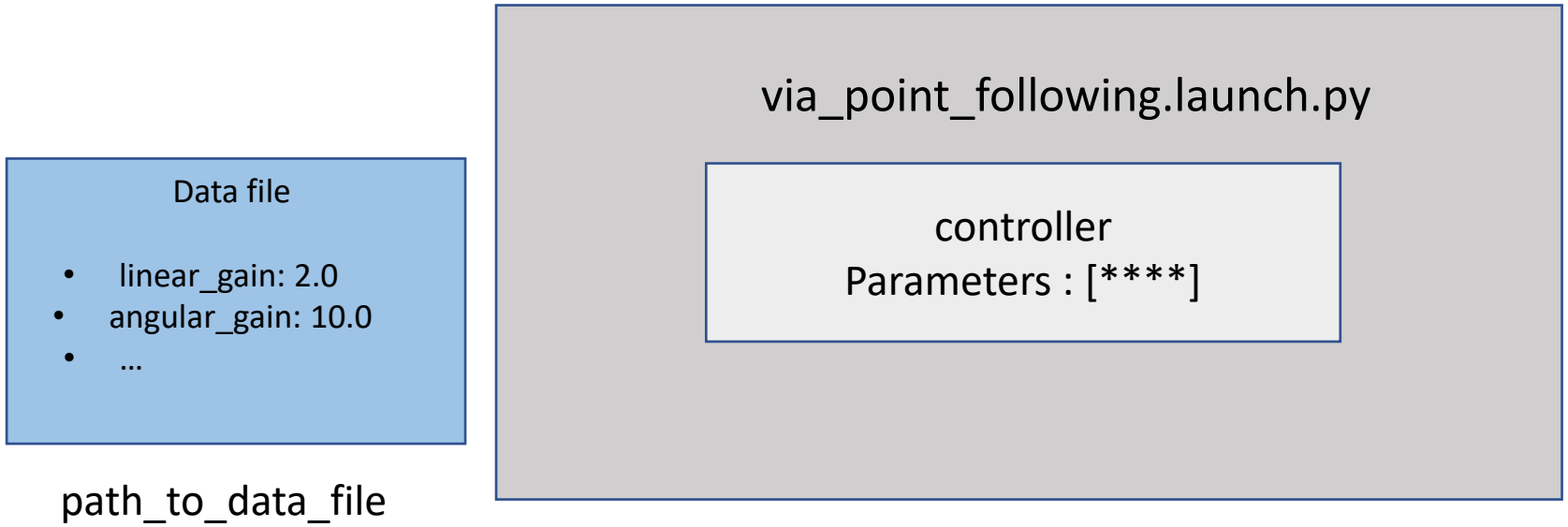


Data Deserialization

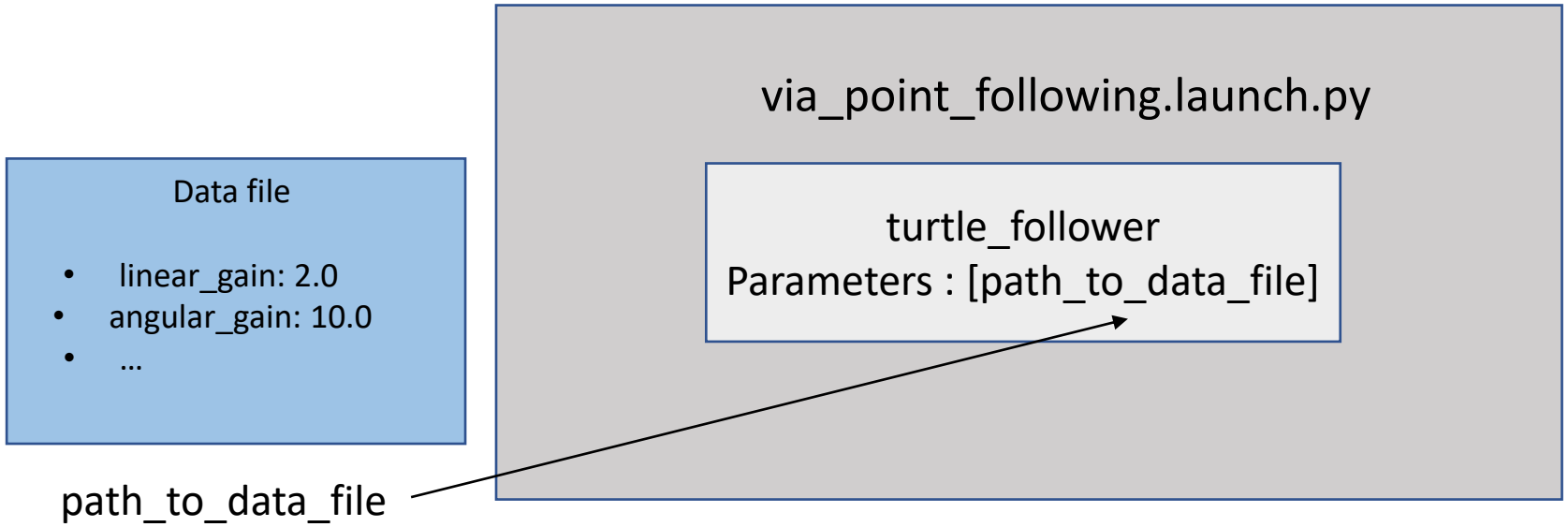
Passing Parameters Manually



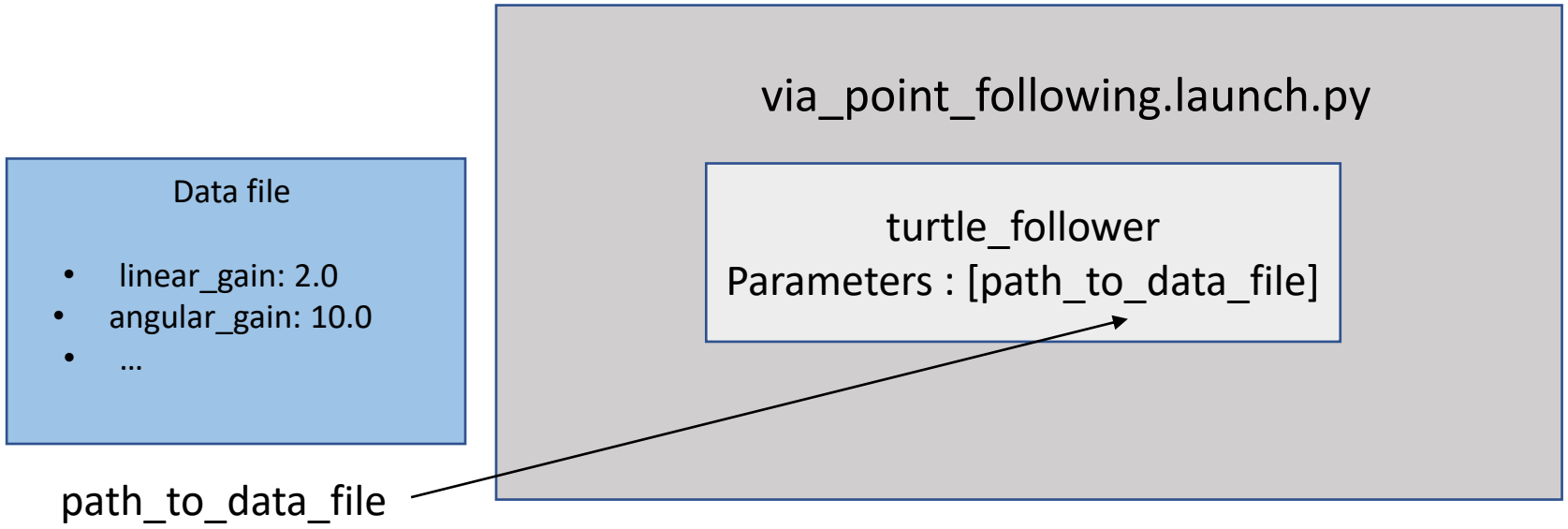
Automatic Deserialization from YAML file



Automatic Deserialization from YAML file

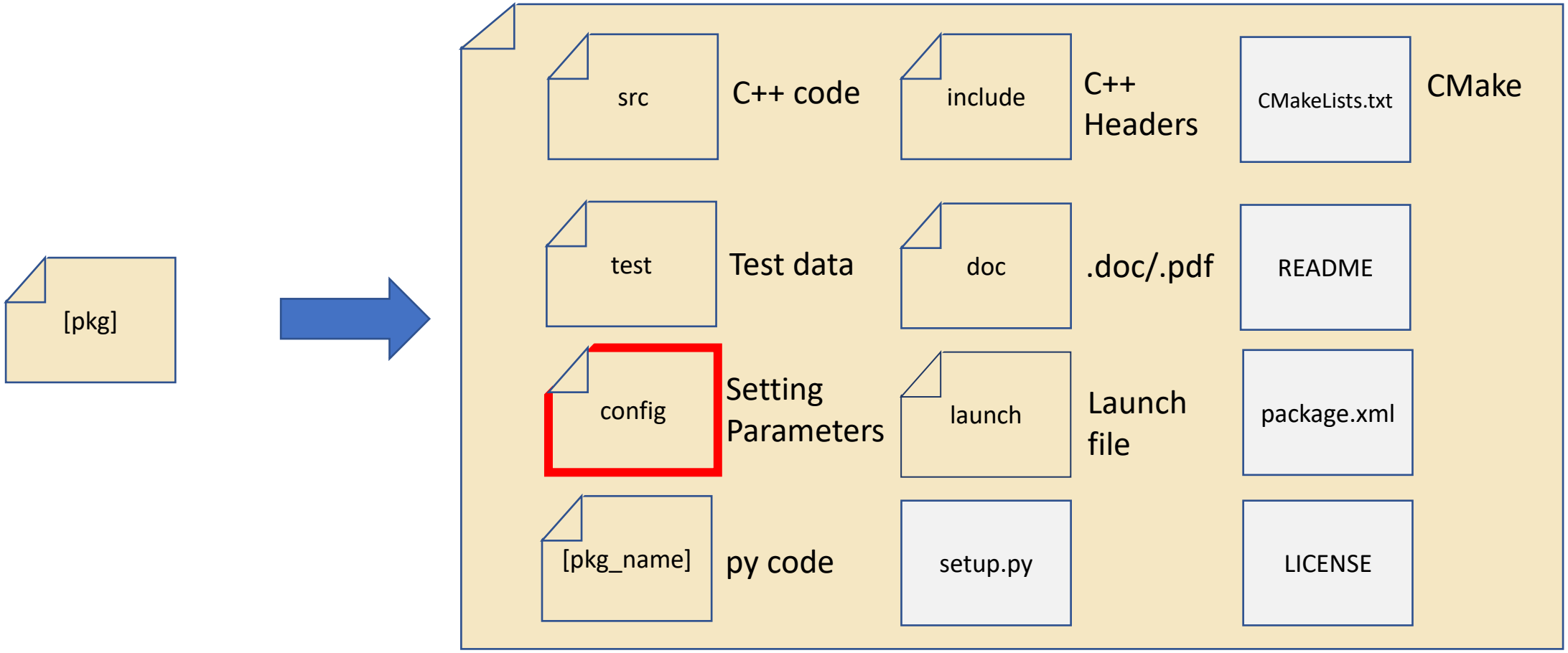


Automatic Deserialization from YAML file



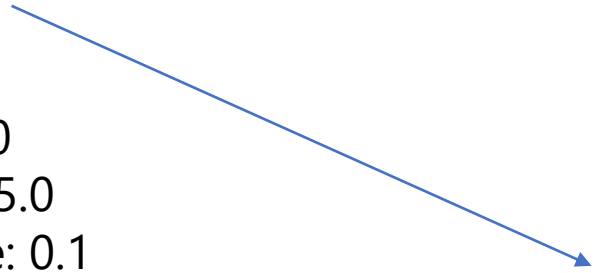
Where is the file ?
How can we make sure that the Launch script load the correct data file regardless of the computer it was installed on?

Package Layout



YAML Format for ROS2 Parameters

```
turtle1:
    controller:
        ros__parameters:
            linear_gain: 1.0
            angular_gain: 5.0
            goal_tolerance: 0.1

turtle2/controller:
        ros__parameters:
            linear_gain: 0.75
            angular_gain: 7.5
            goal_tolerance: 1.0
```



rclpy automatically combines all keys "above" the keyword "ros__parameters" to obtain the associated node's name.

i.e. : /turtle1/via_point_follower

get_package_share_directory

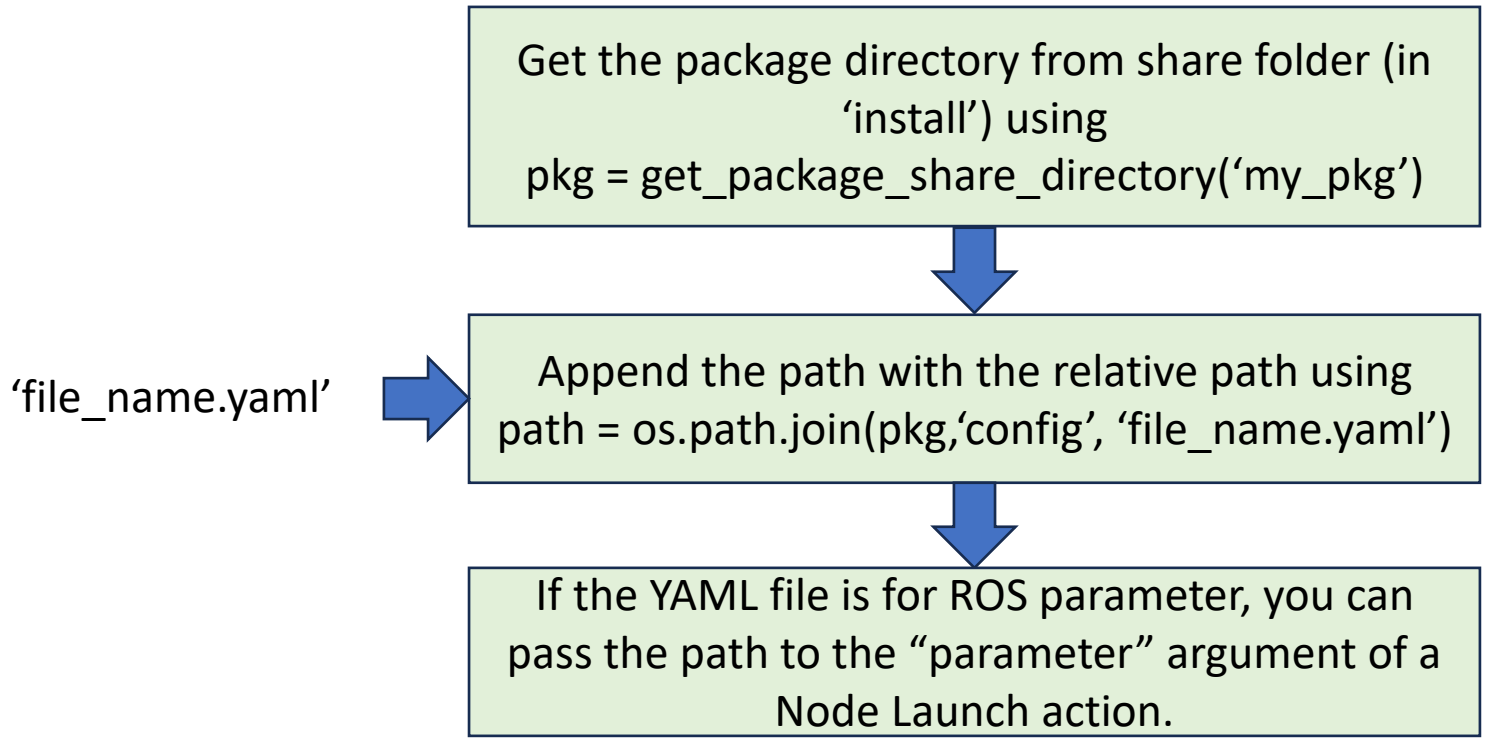
To get the full path to the file, one must get the path to the "installed" package [NOT the one in src].

"get_package_share_directory" allows the software to automatically return the full path to the path with the given name.

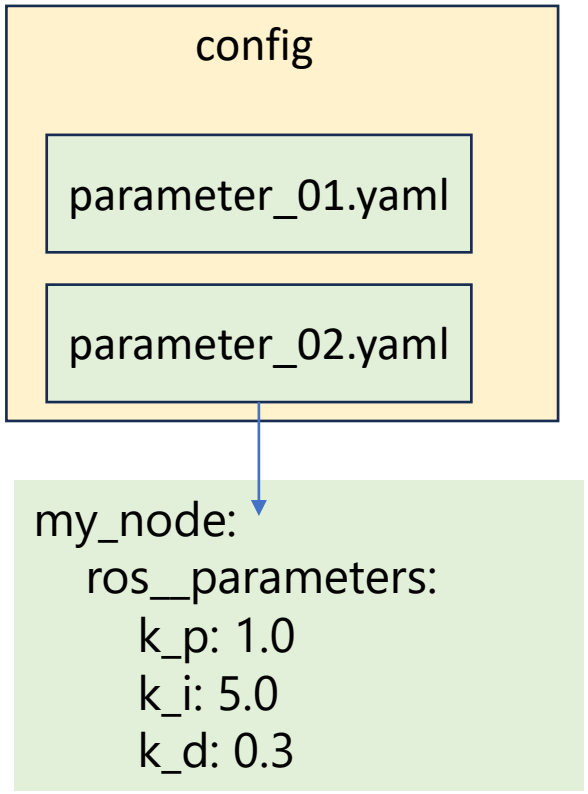
"os.path.join" append the string together as an appropriate path (with /).

```
from ament_index_python.packages import get_package_share_directory
import os

control_pkg = get_package_share_directory('turtlesim_control')
full_path = os.path.join(control_pkg,'config','turtle_parameters.yaml')
```

General steps for deserializing YAML file

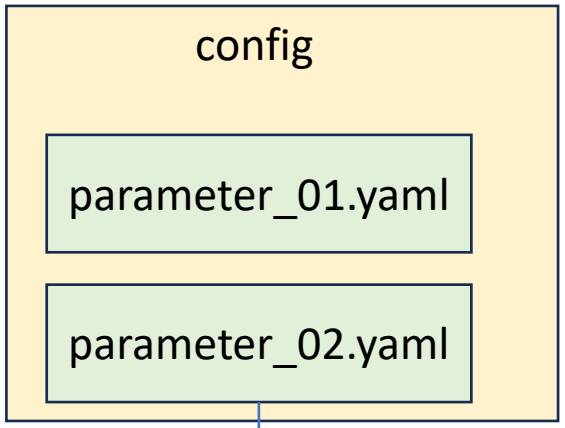


Example



Package: my_pkg
Executable: something.py
Node name: my_node

Example




```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

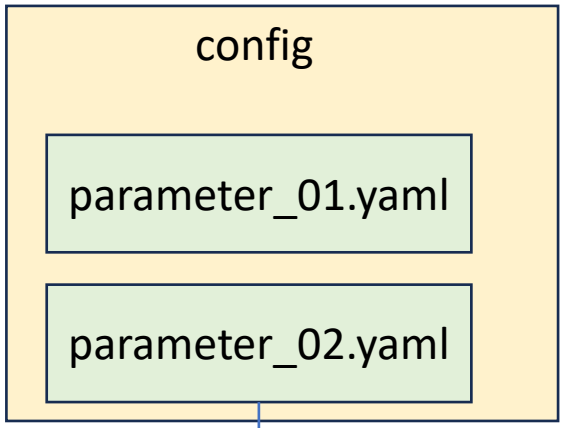
```
from launch import LaunchDescription


def generate_launch_description()
```

```
    launch_description = LaunchDescription()

    return launch_description
```

Package: my_pkg
Executable: something.py
Node name: my_node

Data Deserialization

Example

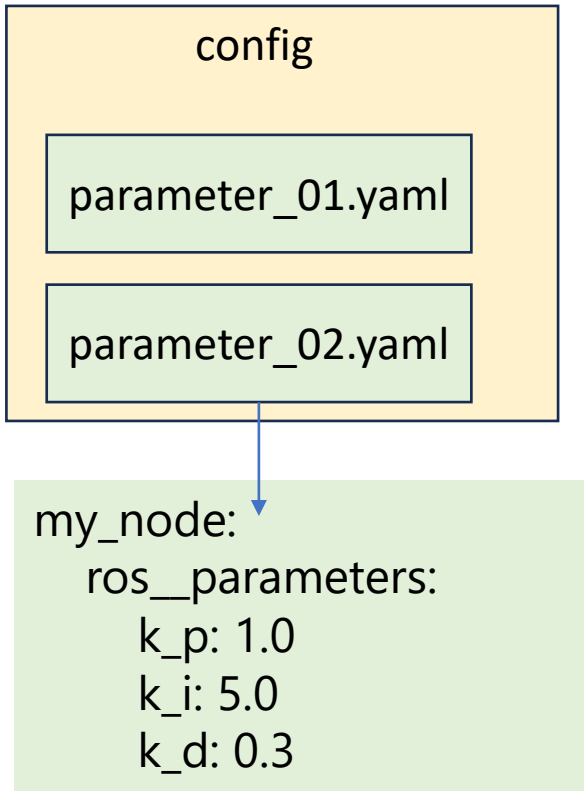



```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description()



    config_path = …




    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        parameters = [config_path]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(my_node_action)

    return launch_description
```

Package: my_pkg
Executable: something.py
Node name: my_node

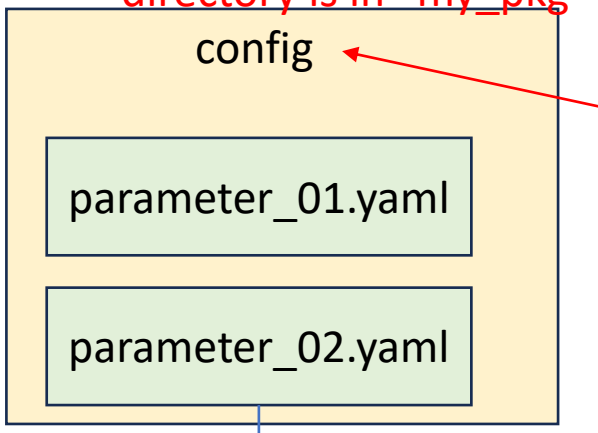
Example




```
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description()


    my_pkg = get_package_share_directory('my_pkg')

    config_path = os.path.join(my_pkg,'config','parameter_01.yaml')




    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        parameters = [config_path]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(my_node_action)

    return launch_description
```

Example

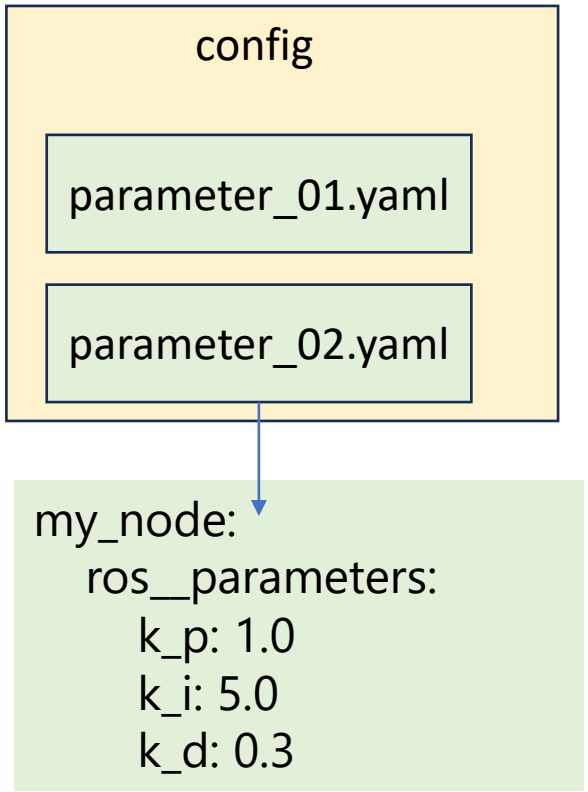
Assume that this config directory is in “my_pkg”




```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

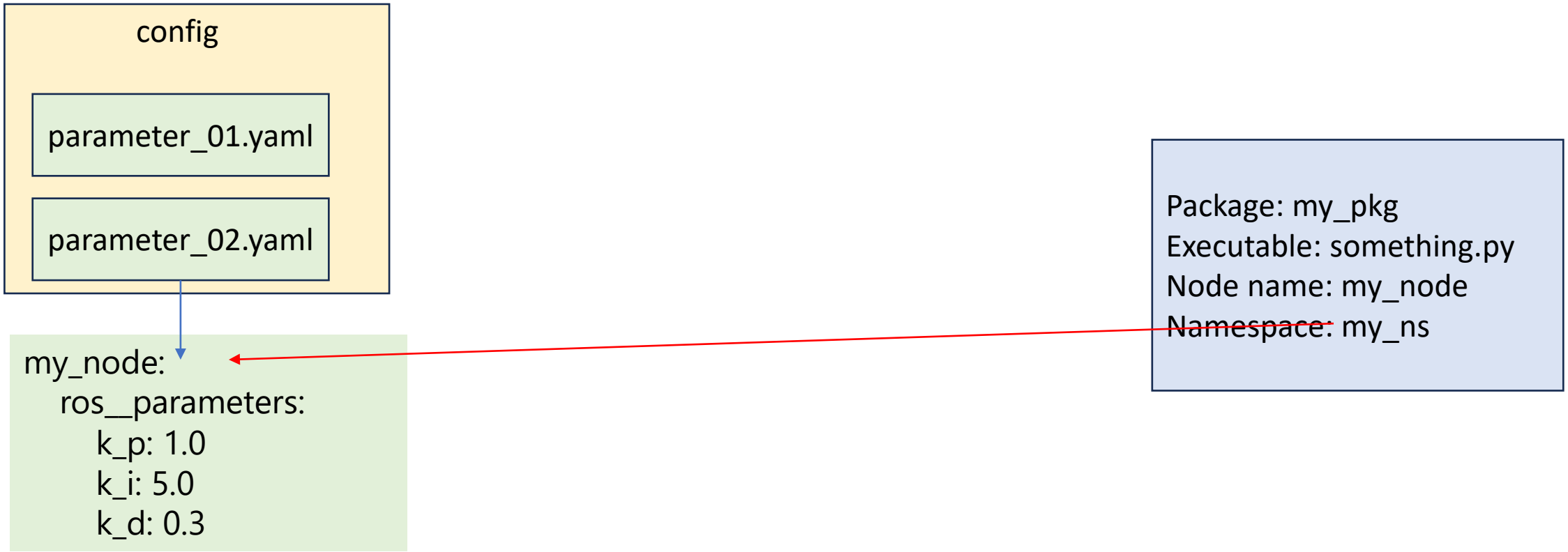
```
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description()


    my_pkg = get_package_share_directory('my_pkg')

    config_path = os.path.join(my_pkg,'config','parameter_01.yaml')




    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        parameters = [config_path]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(my_node_action)

    return launch_description
```

Example: namespace ?



Package: my_pkg
Executable: something.py
Node name: my_node
Namespace: my_ns

Example: namespace ?

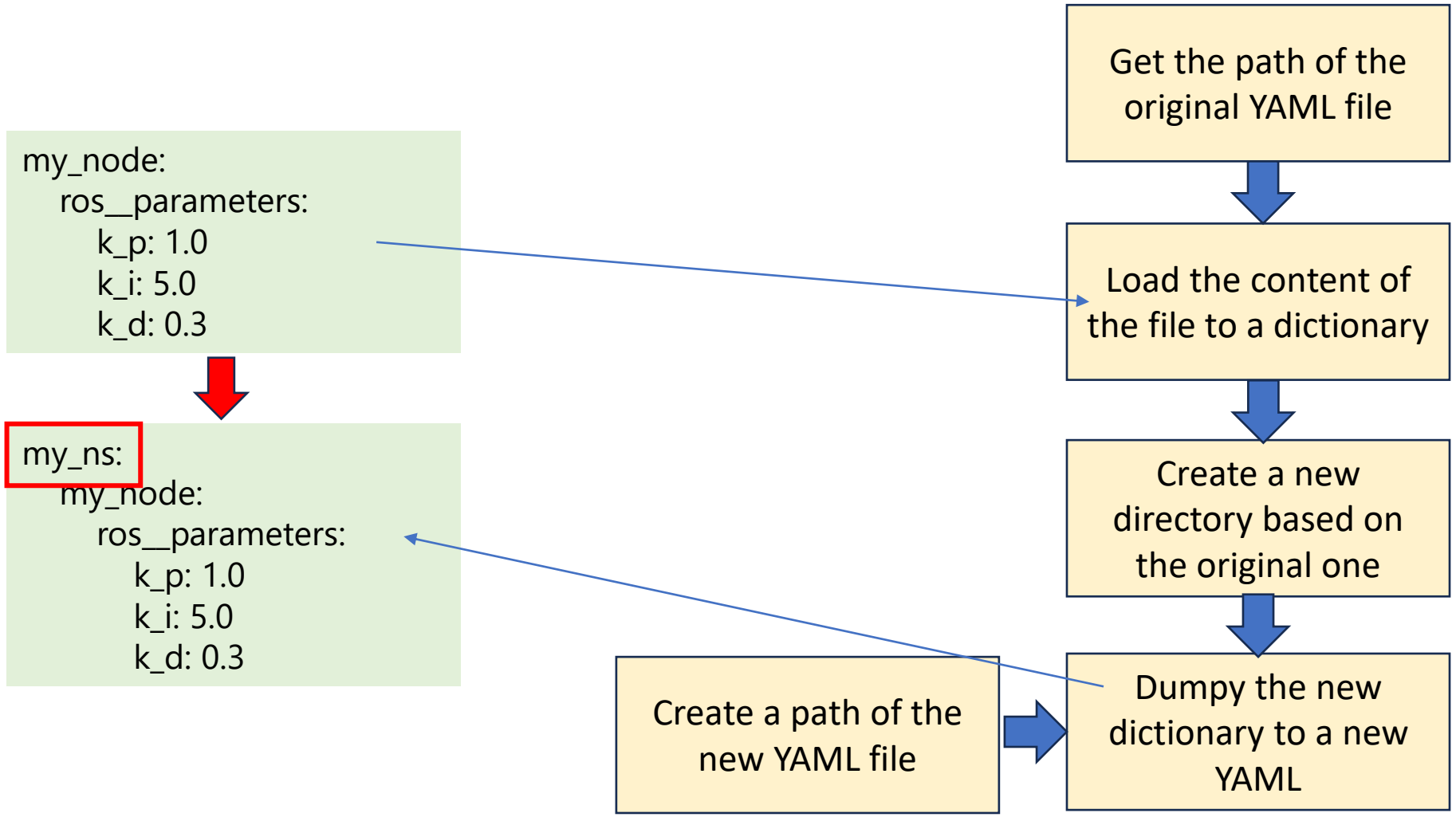


Modifying namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

Programmatically namespace in YAML file



Programmatically namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

```
import yaml

# deserializing a YAML file

with open(path,'r') as file:
    data = yaml.load(file,loader=yaml.SafeLoader)
```

Programmatically namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

```
import yaml

# deserializing a YAML file

with open(path,'r') as file:
    data = yaml.load(file,loader=yaml.SafeLoader)

# creating new content

new_data = {'my_ns': data}
```

Programmatically namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

```
import yaml

# deserializing a YAML file

with open(path,'r') as file:
    data = yaml.load(file,loader=yaml.SafeLoader)

# creating new content

new_data = {'my_ns': data}

# serializing to a new YAML file

with open(new_path,'w') as file:
    yaml.dump(new_data,file)
```

Programmatically namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

```
import yaml

function modify_config_namespace(path:str,new_path,:str,namespace:str) -> None

    with open(path,'r') as file:
        data = yaml.load(file,loader=yaml.SafeLoader)

    new_data = {namespace: data}

    with open(new_path,'w') as file:
        yaml.dump(new_data,file)
```

Programmatically namespace in YAML file

```
my_node:
    ros__parameters:
        k_p: 1.0
        k_i: 5.0
        k_d: 0.3
```

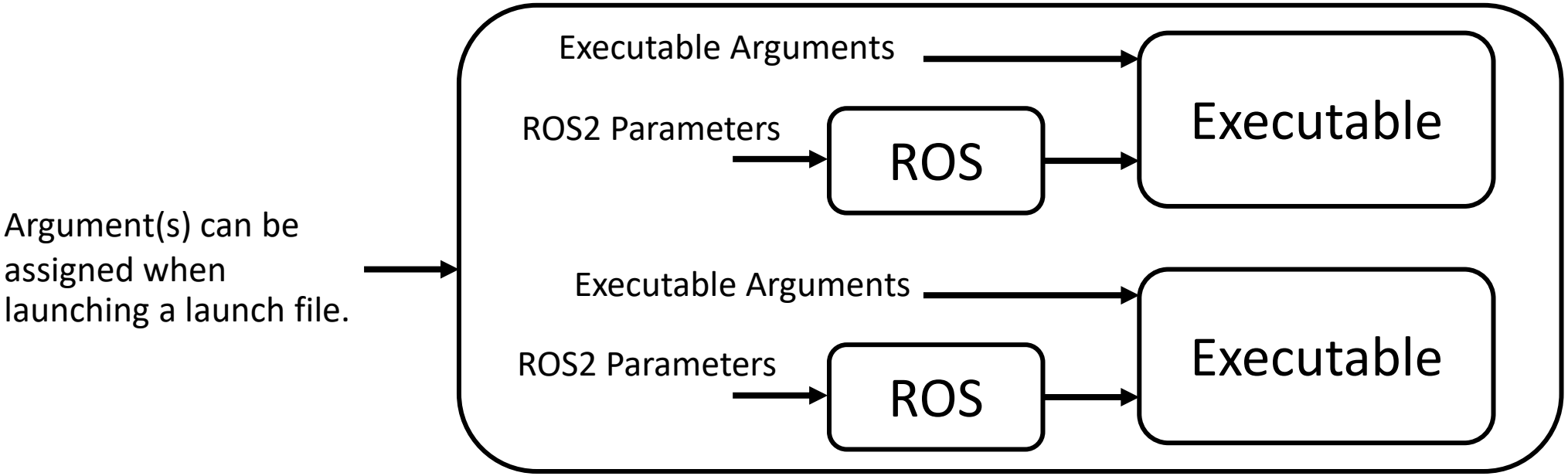
```
my_ns:
    my_node:
        ros__parameters:
            k_p: 1.0
            k_i: 5.0
            k_d: 0.3
```

```
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description()
    namespace = 'my_ns'

    my_pkg = get_package_share_directory('my_pkg')

    config_path = os.path.join(my_pkg,'config','parameter_01.yaml')

    new_config_path = os.path.join(my_pkg,'config','parameter_01_'+namespace+'.yaml')

    modify_config_namespace(config_path,new_config_path,namespace)

    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        namespace = namespace,
        parameters = [config_path]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(my_node_action)

    return launch_description
```

Data Deserialization

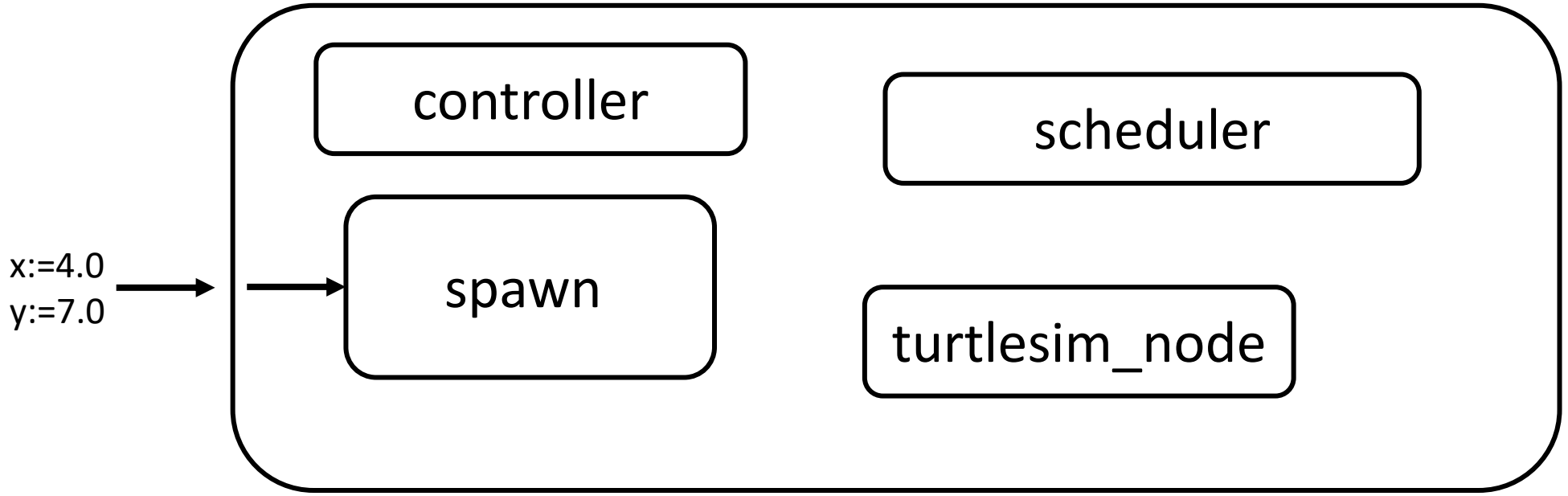
Launch Argument & Launch Configuration

Launch Arguments



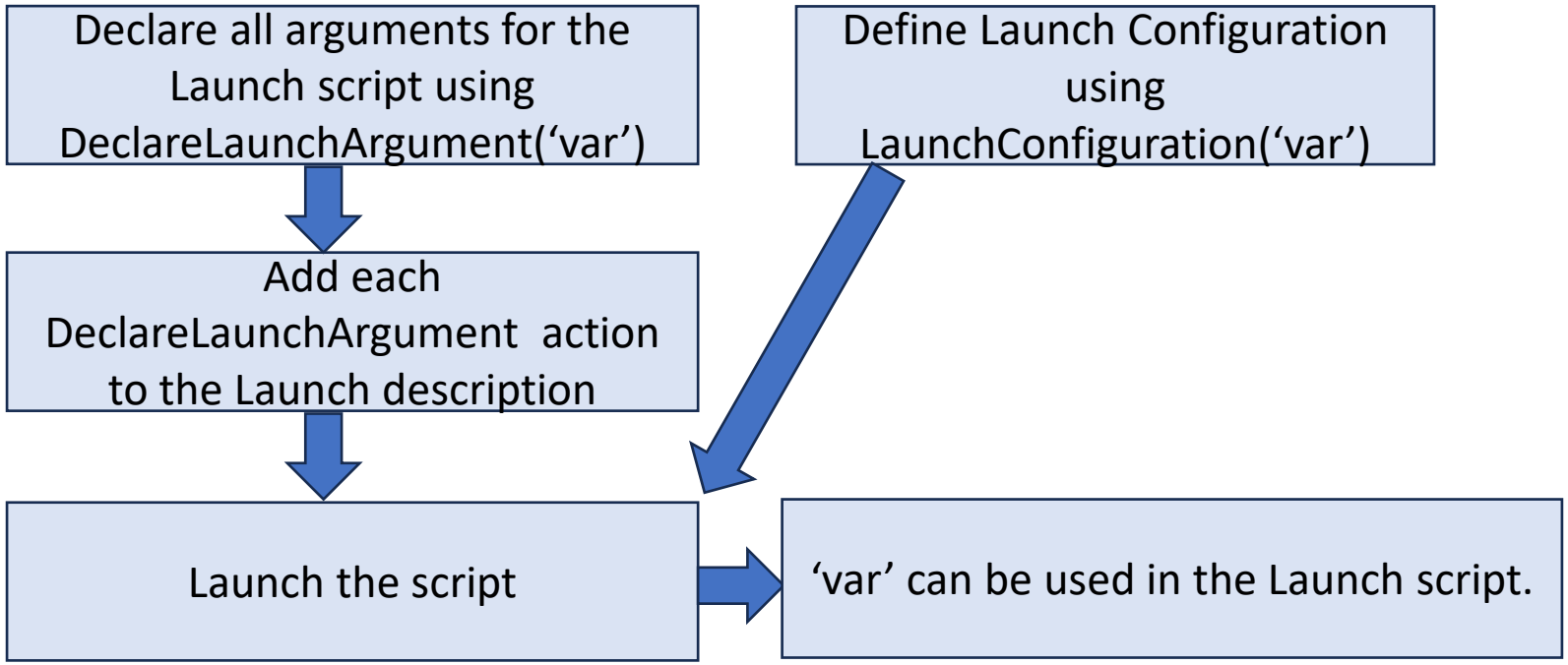
There is **NO** such thing as "Launch parameters"

Example: Launching with initial spawn location



>> ros2 launch turtlsim_control via_point_following.launch.py x:=4.0 y:=7.0

General Pipeline for adding arguments



Adding Launch Arugments to the Launch file

```
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

x_launch_arg = DeclareLaunchArgument('x', default_value='2.0')
y_launch_arg = DeclareLaunchArgument('y', default_value='2.0')
x = LaunchConfiguration('x')
y = LaunchConfiguration('y')

cmd = LaunchConfiguration('cmd',default=[
    'ros2 service call /spawn turtlesim/srv/Spawn "{x: ',
    x,
    ', y: ',
    y,
    ', theta: 0.0, name: \'',turtle_name,'\'}"'
])
spawn_turtle = ExecuteProcess(cmd = [[cmd]], shell=True)
```

DeclareLaunchArgument adds an ability to assign argument of the given name when launching.

LaunchConfiguration allows the Launch file to store and substitute value that is assigned to the variable.

Example:

Launch Argument:
- k_p : (2.0)

```
from launch import LaunchDescription

from launch_ros.actions import Node

from ament_index_python.packages import get_package_share_directory

def generate_launch_description()



    namespace = 'my_ns'

    my_pkg = get_package_share_directory('my_pkg')

    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        namespace = namespace
        parameters = [{'k_p': ...}]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(my_node_action)

    return launch_description
```

Example:

Launch Argument:
- k_p : (2.0)

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

from ament_index_python.packages import get_package_share_directory

def generate_launch_description()

    k_p_launch_arg = DeclareLaunchArgument('k_p', default_value='2.0’)


    namespace = 'my_ns'

    my_pkg = get_package_share_directory('my_pkg')

    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py’,
        namespace = namespace
        parameters = [{'k_p': ...}]
    )

    launch_description = LaunchDescription()
    launch_description.add_action(k_p_launch_arg)
    launch_description.add_action(my_node_action)

    return launch_description
```

Creation order matters !!
Arguments must be declared before they are referred to.

Example:

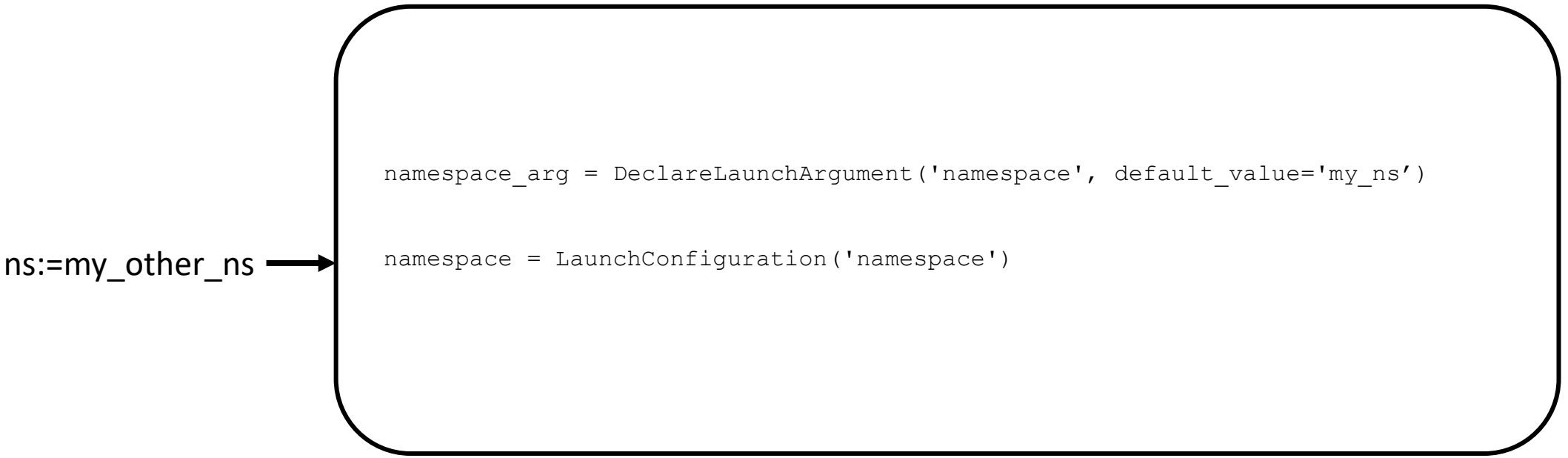
Launch Argument:
- k_p : (2.0)

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node
from launch.substitution import LaunchConfiguration
from ament_index_python.packages import get_package_share_directory

def generate_launch_description()

    k_p_launch_arg = DeclareLaunchArgument('k_p', default_value='2.0’)
    k_p = LaunchConfiguration('k_p')

    namespace = 'my_ns'

    my_pkg = get_package_share_directory('my_pkg')

    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py’,
        namespace = namespace
        parameters = [{'k_p': k_p}]
    )

    launch_description = LaunchDescription()
    launch_description.add_action(k_p_launch_arg)
    launch_description.add_action(my_node_action)

    return launch_description
```

What if ?

What do we have to do ?

What if ?



Add both Launch arguments and Launch configurations

What if ?

ns:=my_other_ns

```
# add launch arguments
. . .

. . .

modify_config_namespace(config_path,new_config_path,namespace)

my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        namespace = namespace,
        parameters = [config_path]
    )
. . .
```

These are not Python string. They are Launch configuration, which cannot be used as 'key' to the directory.

Modify the parameter file with the namespace

What if ?

These are not Python string. They are Launch configuration, which cannot be used as 'key' to the directory.

ns:=my_other_ns →

```
# add launch arguments
. . .

. . .

modify_config_namespace(config_path,new_config_path,namespace)

my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        namespace = namespace,
        parameters = [config_path]
    )
. . .
```

Can we just turn Launch configuration to a string ?

What if ?

ns:=my_other_ns

```
# add launch arguments
. . .

. . .

modify_config_namespace(config_path,new_config_path,namespace)

my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py',
        namespace = namespace,
        parameters = [config_path]
    )
. . .
```

These are not Python string. They are Launch configuration, which cannot be used as 'key' to the directory.

… Not really… There is no function to convert LaunchConfiguration directly to string.

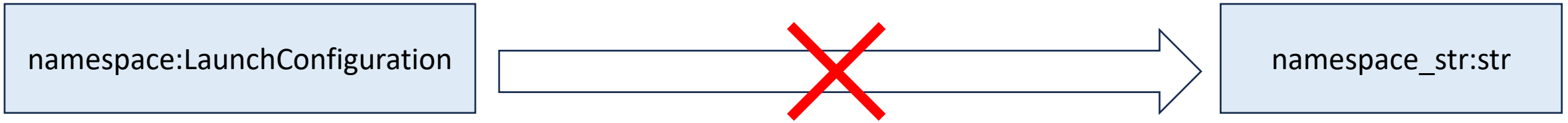
~~namespace.to_string()~~, ~~namespace.to_str()~~ , ~~str(namespace)~~

Launch Action: OpaqueFunction

OpaqueFunction is a class defined in launch.actions. One can associate an OpaqueFunction with another function, which can access a Launch context (LaunchContext class from launch).

One method of the class LaunchContext allows us to perform the substitution on any Launch configuration. This will convert the Launch configuration to Python string. However, this has to be done in the function. We will refer to this as “render_function”.

Launch Context



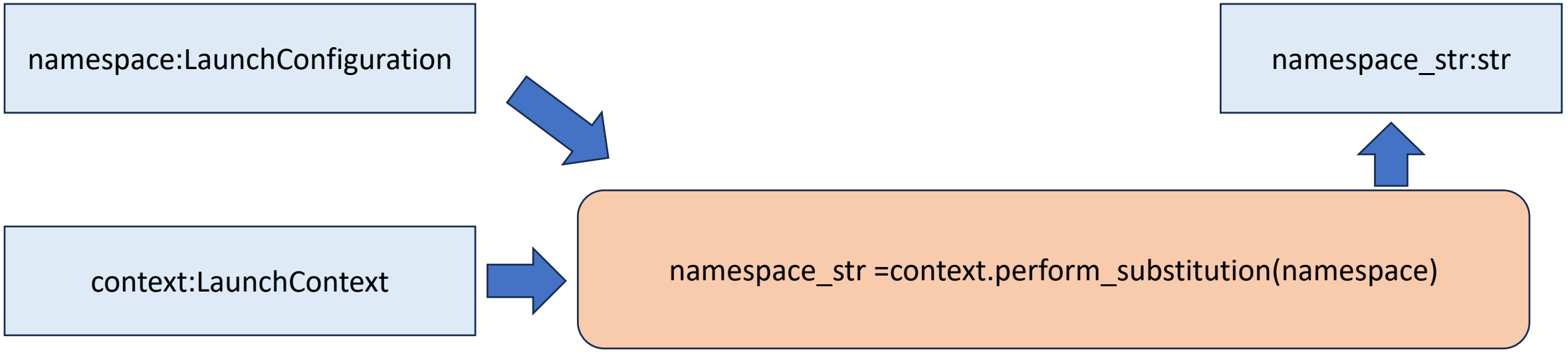
Launch Context

namespace:LaunchConfiguration

namespace_str:str

context:LaunchContext

Launch Context



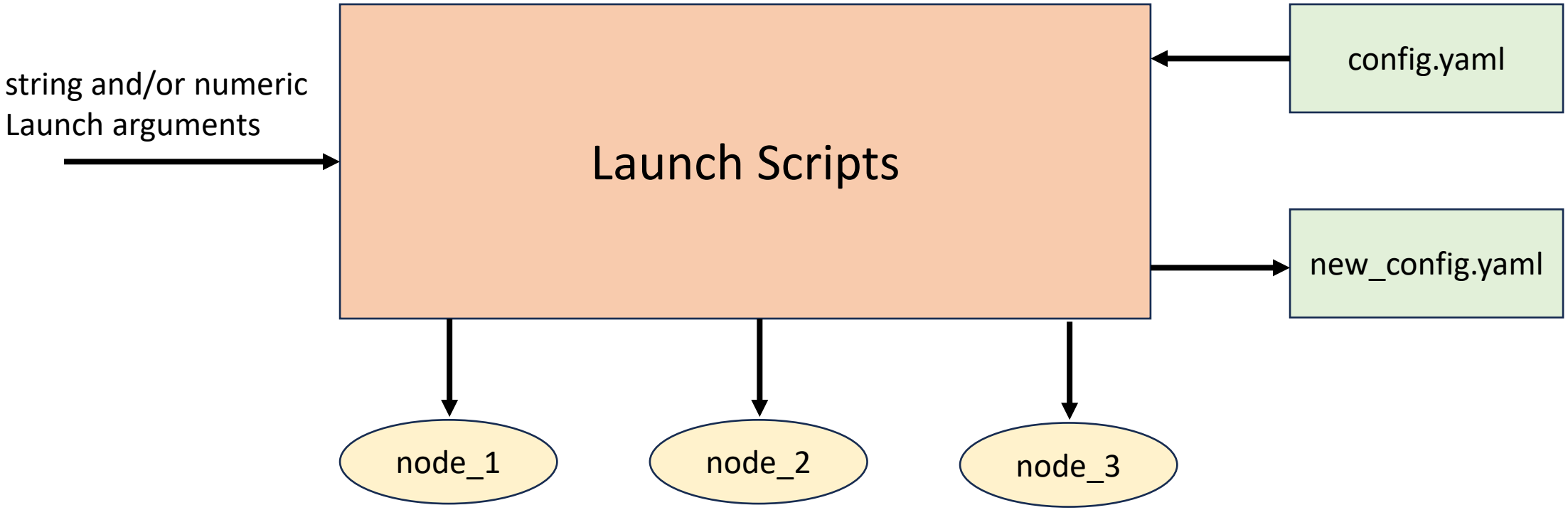
Launch Context

```
from launch import LaunchDescription, LaunchContext
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node
from launch.substitution import LaunchConfiguration
from ament_index_python.packages import get_package_share_directory

def render_namespace(context:LaunchContext,launch_description:LaunchDescription,namespace:LaunchConfiguration) -> None
    namespace_str = context.perform_substitution(namespace)

    my_pkg = get_package_share_directory('my_pkg’)
    config_path = os.path.join(my_pkg,'config','parameter_01.yaml')
    new_config_path = os.path.join(my_pkg,'config','parameter_01_'+namespace_str+'.yaml')

    modify_config_namespace(config_path,new_config_path,namespace_str)

    my_node_action = Node(
        package = 'my_pkg',
        executable = 'something.py’,
        namespace = namespace_str,
        parameters = [config_path]
    )

    launch_description.add_action(my_node_action)
```

OpaqueFunction

```
from launch import LaunchDescription, LaunchContext
from launch.actions import DeclareLaunchArgument, OpaqueFunction
from launch_ros.actions import Node
from launch.substitution import LaunchConfiguration
from ament_index_python.packages import get_package_share_directory

def render_namespace(context:LaunchContext,launch_description:LaunchDescription,namespace:LaunchConfiguration) -> None
    ...

def generate_launch_description()

    namespace_launch_arg = DeclareLaunchArgument('namespace', default_value='my_ns’)
    namespace = LaunchConfiguration('namespace')

    launch_description = LaunchDescription()
    launch_description.add_action(namespace_launch_arg)

    opaque_function = OpaqueFunction(
        function=render_namespace,
        args=[launch_description,namespace]
    )

    launch_description.add_action(opaque_function)

    return launch_description
```

What do we have now?



Scheduling in Launch File

Running shell command using ExecuteProcess

```
from launch.actions import ExecuteProcess

spawn_turtle2 = ExecuteProcess(
        cmd = [['ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0,
y: 2.0, theta: 0.0, name: \'turtle2\'}"']],
        shell=True
)

launch_description.add_action(spawn_turtle2)
```

Event Handlers

The state of launched processes is monitored by the Launch system and can be used to trigger "events".

One can schedule Launch actions to execute when one of the following event occurs.

- OnProcessStart
- OnProcessIO
- OnExecutionComplete
- OnProcessExit
- OnShutdown

OnProcessStart

```
from launch.actions import LogInfo, RegisterEventHandler
from launch.event_handlers import OnProcessStart

turtlesim_node = Node(
    ...
)
spawn_turtle = ExecuteProcess(
    ...
)

event_handler =  RegisterEventHandler(
    OnProcessStart(
        target_action=turtlesim_node,
        on_start=[
            LogInfo(msg='Turtlesim started, spawning turtle'),
            spawn_turtle
        ]
    )
)

launch_description.add_action(event_handler)
```

OnExecutionComplete

```
from launch.actions import LogInfo, RegisterEventHandler
from launch.event_handlers import OnExecutionComplete

turtlesim_node = Node(
    ...
)
spawn_turtle = ExecuteProcess(
    ...
)

event_handler =  RegisterEventHandler(
    OnExecutionComplete(
        target_action= spawn_turtle,
        on_completion=[
            controller_node
        ]
    )
)

launch_description.add_action(event_handler)
```

OnProcessExit

```
from launch.actions import EmitEvent, LogInfo, RegisterEventHandler
from launch.event_handlers import OnProcessExit
from launch.events import Shutdown

turtlesim_node = Node(
        ...
)
spawn_turtle = ExecuteProcess(
        ...
)

event_handler =  RegisterEventHandler(
    OnProcessExit(
        target_action=turtlesim_node,
        on_exit=[
            LogInfo(msg='closed the turtlesim window’)),
            EmitEvent(event=Shutdown(reason='Window closed’))
        ]
    )
)

launch_description.add_action(event_handler)
```

Launch files in a Launch file

Calling another Launch file in a Launch file

```
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

turtlesim_spawn_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(turtlesim_control_pkg,'launch','turtlesim_spawn.launch.py')
    ),
    launch_arguments = {
        'x': x,
        'y': y,
        'name':namespace
    }.items()
)
```

Get full path of other Launch file

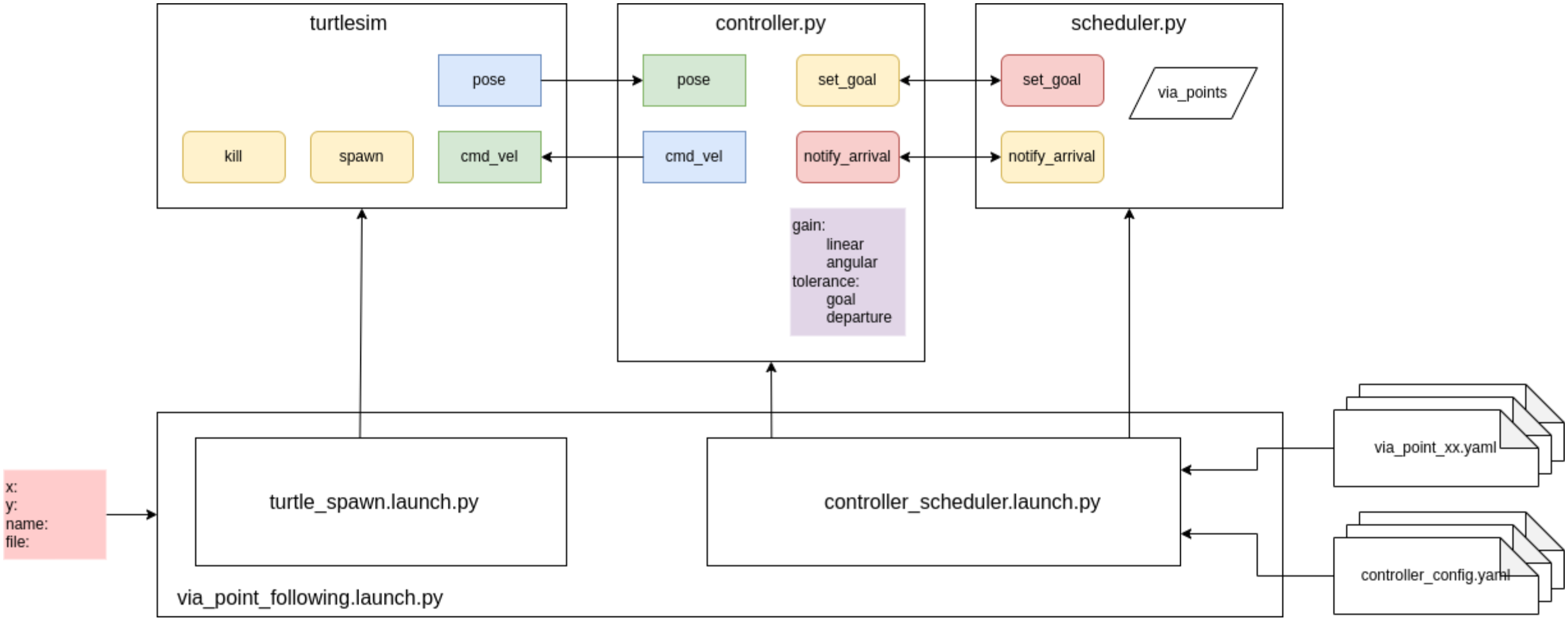
One can also pass along the arguments to other Launch file.

Summary

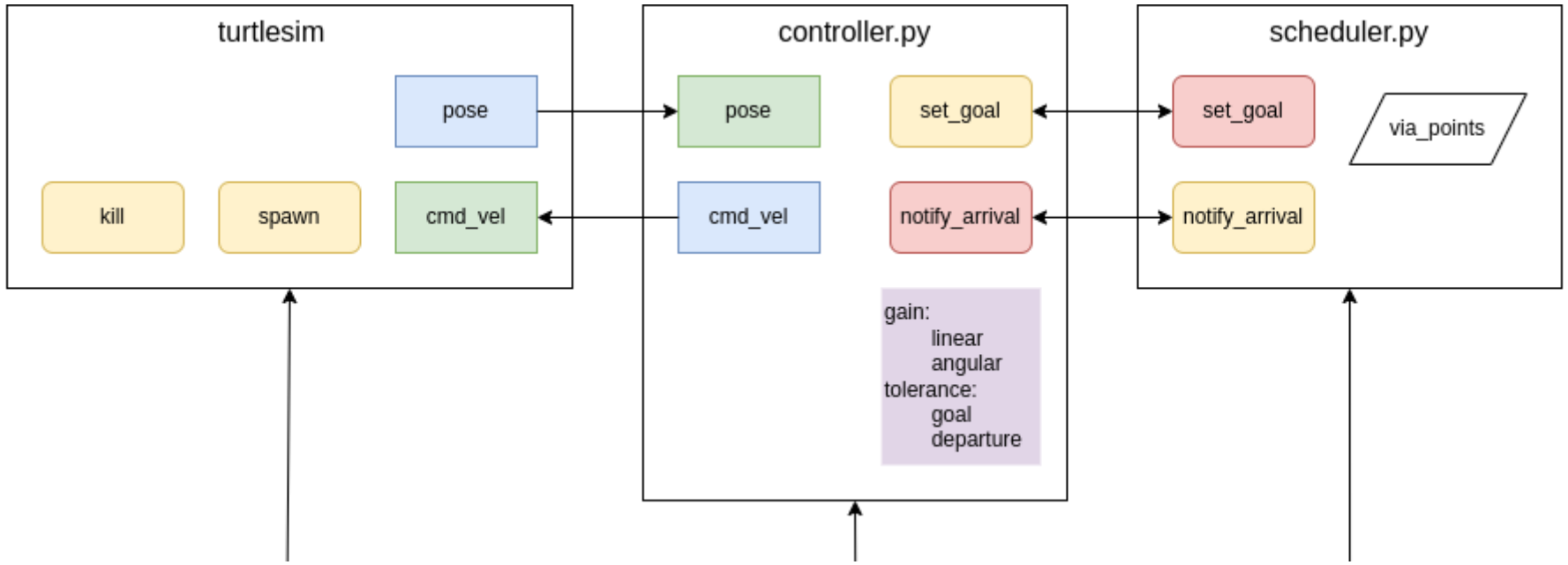
- package structure (review)
- namespace
- parameters
- python argument/ argprase
- launch scripts & launch action
- launch arguments & launch configuration
- deserialization w/ YAML
- opaque function
- scheduling
- Launch in Launch

Exercise

Exercise



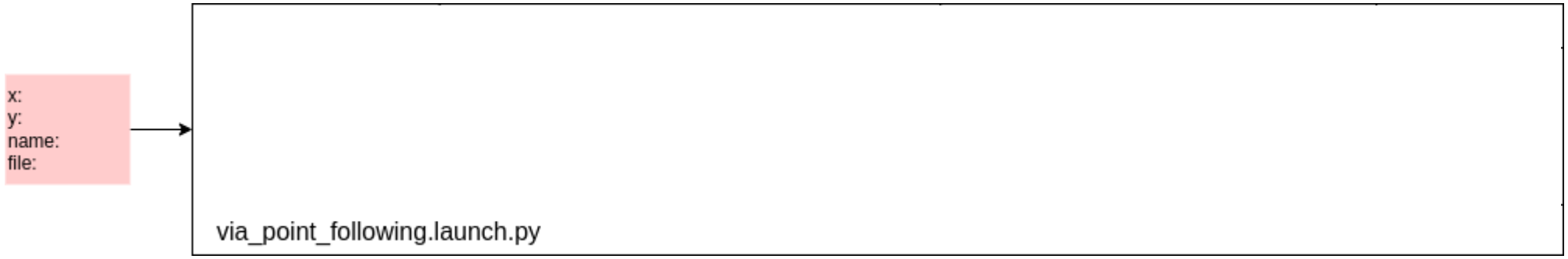
Goal 1: Launch multiple nodes



Launch turtlesim_node, controller.py, and scheduler.py with proper namespace, and executable arguments.

Exercise

Goal 2: Accepting Launch Arguments

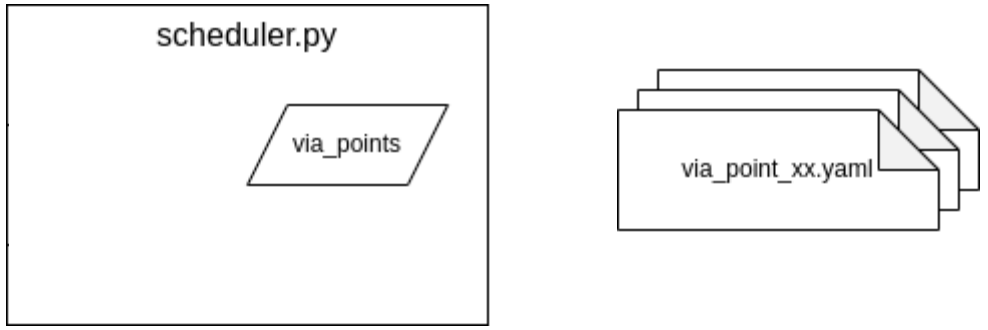


The Launch script must accept 4 optional arguments:

- x: spawn location in x-direction
- y: spawn location in y-direction
- name: name of the turtle
- file: path of the via point file (relative to the ‘via point’ directory of the package)

Exercise

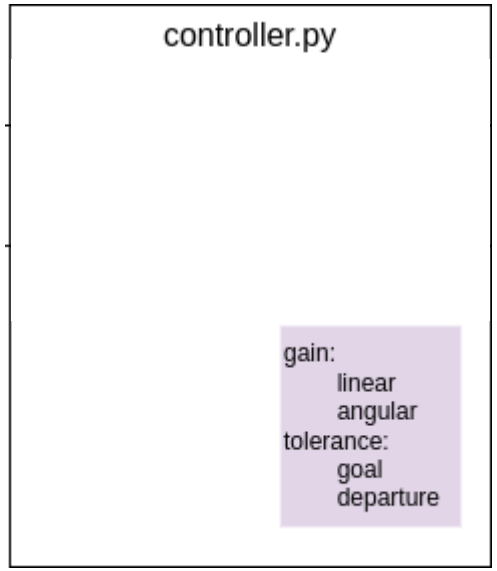
Goal 3: Pass parameters via a YAML file

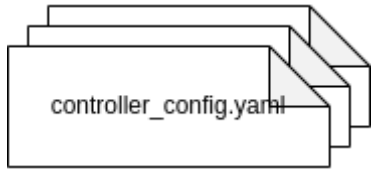


Pass the full path to the via point file in the "via_point" directory of the package based on the given (relative) file name to the scheduler

Exercise

Goal 4: Creating a new YAML file





Given a “base” configuration file for a generic controller node, create a new configuration file for a fully-qualified node and update the parameters of the node using the file.

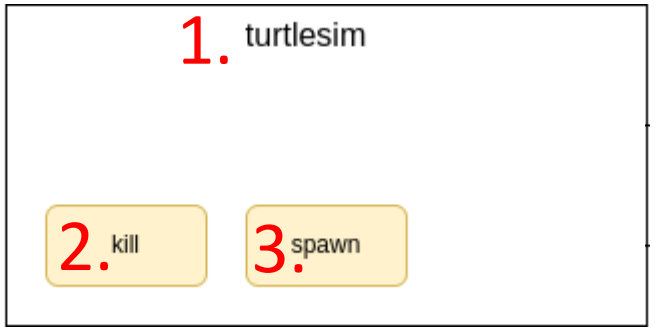
The base configuration conists of 'linear_gain','angular_gain',and 'tolerance’

The base configuration file is named “controller_config.yaml” as should look like this. [The parameters in the diagram are incorrect.]

```
controller:
   ros__parameters:
      linear_gain:1.0
      angular_gain:5.0
      tolerance:0.2
```

Exercise

Goal 5: Scheduling processes



When turtlesim_node starts, kill any existing turtle in turtlesim, then spawn a new one with the given name and location.

Exercise

Example Repo- Branch: launch-solution

https://github.com/kittinook/FRA501/tree/launch-solution

Download & add “turtlesim_control” & “turtlesim_interfaces” to the “src” directory of your workspace. Then build these packages.

Follow the instruction on README.md