Annin AR4 Multi-Arm

# Github

<https://github.com/ycheng517/ar4_ros_driver_examples>

Teleoperation using Xbox controller & Multi-Arm Control in Simulation

# gazebo.launch.py

## Overall

This file is a **ROS 2 launch script** written in Python that sets up a sophisticated robotics simulation environment. Think of it as an orchestration script that coordinates multiple software components to create a complete robotic system simulation.

**The Overall Purpose**

This launch file creates a Gazebo simulation environment containing **two identical robotic arms** (AR4 industrial robots) positioned side by side.

**The Key Components**

The script coordinates several essential systems that need to work together for a complete robot simulation:

**Gazebo Simulation Environment**: This is like creating a virtual physics laboratory where the robots exist. The script launches Gazebo (a 3D robot simulator) with an empty world file, providing the physical environment where the robots will operate.

**Robot Description and Visualization**: For each robot arm, the script generates a complete description of the robot's physical structure, joints, and properties. This is like providing Gazebo with detailed blueprints of each robot, including their dimensions, joint limits, and visual appearance.

**Control Systems**: Perhaps most importantly, the script sets up sophisticated control systems for each robot. These include joint state broadcasters (which report the current position of each joint) and trajectory controllers (which can move the robot arms along planned paths).

**The Clever Architecture Choices**

What makes this script particularly elegant is how it handles **dual robot setup**. Instead of writing separate code for each robot, it uses a clever loop that creates identical setups for two different namespaces: "/left" and "/right". Think of namespaces as separate communication channels - this prevents the two robots from interfering with each other's control signals.

The script also demonstrates sophisticated **parameter handling**. It includes a custom class called ControllerConfigSubstitution that dynamically modifies configuration files. This is like having a template system that can customize robot settings on the fly based on which specific robot model you're using.

**The Technical Flow**

Here's how the script orchestrates everything: First, it launches the Gazebo simulation environment. Then, for each robot (left and right), it spawns a complete robotic system including the robot model, state publishers that broadcast joint information, and multiple controllers that can command different parts of the robot (arm joints and gripper).

The script also sets up a **bridge between ROS and Gazebo** communication systems, ensuring that timing information flows correctly between the simulation and the ROS control systems. This synchronization is crucial for realistic simulation behavior.

**Why This Architecture Matters**

This script essentially creates a complete virtual robotics laboratory where you could test coordination algorithms, path planning, or manipulation tasks using two robotic arms working in the same environment.

## ControllerConfigSubstitution vs generate\_launch\_description

*Q: Can you explain the relation between ControllerConfigSubstitution and generate\_launch\_description? Which one is called first? Is any of these called automatically or does one call the other?*

### Execution Order and Timing

Here's the sequence of what actually happens:

**1. generate\_launch\_description() is Called First**

When you run the launch file (e.g., ros2 launch my\_package gazebo.launch.py), the ROS launch system automatically calls generate\_launch\_description(). This is a **required function** that every ROS 2 launch file must have - it's the entry point.

**2. ControllerConfigSubstitution Objects are Created (But Not Executed)**

Inside generate\_launch\_description(), when this line runs:

initial\_joint\_controllers = ControllerConfigSubstitution(

PathJoinSubstitution([

FindPackageShare("annin\_ar4\_driver"), "config",

"controllers.yaml"

]))

This creates a ControllerConfigSubstitution object, but **it doesn't execute its perform() method yet**. Think of it like writing an IOU note - you're promising to do something later, but you're not doing it right now. (In Python we don’t use “new” to create objects)

**3. The Launch Description is Built with Substitution Objects**

The generate\_launch\_description() function continues building nodes that reference these substitution objects:

robot\_description\_content = Command([

*# ... other parameters ...*

"simulation\_controllers:=",

initial\_joint\_controllers, *# This is still just a substitution object*

])

At this point, initial\_joint\_controllers is still just an object waiting to be resolved.

**4. ControllerConfigSubstitution.perform() is Called Later**

The perform() method gets called by the ROS launch system when it actually needs the resolved value. This happens during the **launch execution phase**, which occurs after generate\_launch\_description() has completely finished and returned.

Note: The perform method is a **standard method name** that's part of ROS 2's substitution system architecture. It's not just a naming convention - it's a required method that must be implemented by any class that inherits from the Substitution base class.

### The Launch System's Two-Phase Process

Think of it like this:

**Phase 1: Planning** (when generate\_launch\_description() runs)

* Create a blueprint of what needs to happen
* Set up substitution objects as "placeholders" for values that will be determined later
* Build the complete launch description structure

**Phase 2: Execution** (when the launch system processes the description)

* Resolve all substitutions by calling their perform() methods
* Start nodes with the resolved values
* Execute the actual system

## Imports

### Code

import os

import tempfile

from ament\_index\_python.packages import get\_package\_share\_directory

from launch import LaunchDescription

from launch.actions import DeclareLaunchArgument

from launch.actions import IncludeLaunchDescription

from launch.launch\_description\_sources import PythonLaunchDescriptionSource

from launch.substitution import Substitution

from launch.substitutions import (

    Command,

    FindExecutable,

    PathJoinSubstitution,

    LaunchConfiguration,

)

from launch\_ros.actions import Node

from launch\_ros.substitutions import FindPackageShare

### Explanation

**tempfile** - is used to create temporary files and directories.

### get\_package\_share\_directory

from ament\_index\_python.packages import get\_package\_share\_directory is from ROS 2.

This function's primary purpose is to find the installation path of the share directory for a given ROS 2 package.

The share directory is a standard location in ROS 2 packages where files that need to be shared at runtime are stored. This can include:

* Configuration files (.yaml)
* Launch files (.launch.py)
* URDF (Unified Robot Description Format) files for robot models
* Rviz configuration files

### launch => LaunchDescription

LaunchDescription is a class in the **ROS 2 launch framework** used to define and manage a collection of nodes, processes, and other launch files to be executed together. It's the central container for a ROS 2 launch file.

**What It Does**

When you write a Python launch file in ROS 2, you create an instance of LaunchDescription and fill it with various launch actions. This object then defines a complete system that ROS 2 can start, stop, and manage. Think of it as a blueprint for your application.

For example, a LaunchDescription might contain:

* **Node actions**: To start individual ROS 2 nodes, specifying their package, executable name, and any arguments.
* **GroupAction**: To group multiple actions together, often to apply a common set of parameters or rules.
* **IncludeLaunchDescription**: To embed another launch file, allowing you to create modular and reusable launch configurations.
* **ExecuteProcess**: To run a standard command-line program that isn't a ROS 2 node.

By using LaunchDescription, you can launch an entire robotic system—including a robot's driver, a navigation stack, and a user interface—with a single command.

**Why Use It?**

The LaunchDescription class and the ROS 2 launch framework offer several advantages:

* **Reproducibility**: It ensures that everyone on a team can launch the same complex system with identical configurations, reducing setup errors.
* **Modularity**: You can break down a large system into smaller, reusable launch files.
* **Parameter Management**: It allows you to set parameters for your nodes in a centralized location, making it easy to tune your system without recompiling code.
* **Process Management**: It automatically handles the startup and shutdown of all the defined nodes and processes.

### launch.actions => DeclareLaunchArgument

from launch.actions import DeclareLaunchArgument is a Python statement used in ROS 2 launch files to import the DeclareLaunchArgument class. This class allows you to define command-line arguments for your launch file, making it more flexible and reusable.

**What DeclareLaunchArgument Does**

DeclareLaunchArgument creates a variable that can be set from the command line when the launch file is executed. This variable can then be used within the launch file to modify the behavior of nodes or other actions. It's essentially a way to create a user-configurable setting for your launch script.

The constructor for DeclareLaunchArgument takes several key arguments:

* name: The name of the launch argument (e.g., 'robot\_name'). This is how you will refer to it both on the command line and within your launch file.
* default\_value: The default value the argument will have if the user doesn't provide one.
* description: A helpful description of what the argument does. This is shown when a user requests help for the launch file.

### launch.actions => IncludeLaunchDescription

The IncludeLaunchDescription class allows you to **embed one launch file inside another**, creating modular and reusable launch configurations.

### PythonLaunchDescriptionSource

from launch.launch\_description\_sources import PythonLaunchDescriptionSource used in ROS 2 launch files to import the PythonLaunchDescriptionSource class. This class is essential for loading and processing other launch files that are written as Python scripts.

PythonLaunchDescriptionSource acts as a **loader** or **parser** for a Python-based launch file.

### launch.substitution => Substitution

from launch.substitution import Substitution is a Python statement used in ROS 2 launch files to import the Substitution base class. This class is part of the **ROS 2 launch framework's substitution system**, which allows you to create dynamic values that are resolved at runtime.

**What Is a Substitution?**

A **Substitution** is a way to represent a value that isn't a simple string or number but instead depends on some context that's only available when the launch file is actually running. Examples of these contexts include:

* The value of a launch argument.
* The name of the current package.
* The value of an environment variable.

Instead of a hardcoded value, you use a Substitution object. When the launch system executes a node or an action, it looks for these Substitution objects and resolves them to their final string value.

### FindPackageShare

from launch\_ros.substitutions import FindPackageShare is a Python statement used in ROS 2 launch files to import the FindPackageShare class. This class is a **substitution** that finds the absolute path to a package's share directory at runtime.

## generate\_launch\_description

### Code

def generate\_launch\_description():

    ar\_model\_arg = DeclareLaunchArgument("ar\_model",

                                         default\_value="mk3",

                                         choices=["mk1", "mk2", "mk3"],

                                         description="Model of AR4")

    ar\_model\_config = LaunchConfiguration("ar\_model")

    # Gazebo nodes

    world = os.path.join(get\_package\_share\_directory('annin\_ar4\_gazebo'),

                         'worlds', 'empty.world')

    gazebo = IncludeLaunchDescription(PythonLaunchDescriptionSource(

        [FindPackageShare("ros\_gz\_sim"), "/launch", "/gz\_sim.launch.py"]),

                                      launch\_arguments={

                                          'gz\_args': f'-r -v 4 {world}',

                                          'on\_exit\_shutdown': 'True'

                                      }.items())

stuff\_to\_spawn = []

    for namespace in ["/left", "/right"]:

        initial\_joint\_controllers = ControllerConfigSubstitution(

            PathJoinSubstitution([

                FindPackageShare("annin\_ar4\_driver"), "config",

                "controllers.yaml"

            ]))

        robot\_description\_content = Command([

            PathJoinSubstitution([FindExecutable(name="xacro")]),

            " ",

            PathJoinSubstitution([

                FindPackageShare("annin\_ar4\_description"),

                "urdf",

                "ar\_gazebo.urdf.xacro",

            ]),

            " ",

            "ar\_model:=",

            ar\_model\_config,

            " ",

            f"namespace:={namespace}",

            " ",

            "simulation\_controllers:=",

            initial\_joint\_controllers,

        ])

        robot\_description = {"robot\_description": robot\_description\_content}

        robot\_state\_publisher\_node = Node(

            package="robot\_state\_publisher",

            executable="robot\_state\_publisher",

            output="both",

            parameters=[

                robot\_description,

                {

                    "use\_sim\_time": True

                },

            ],

            namespace=namespace,

            remappings=[("/tf", "tf"), ("/tf\_static", "tf\_static")],

        )

        joint\_state\_broadcaster\_spawner = Node(

            package="controller\_manager",

            executable="spawner",

            arguments=[

                "joint\_state\_broadcaster", "-c",

                f"{namespace}/controller\_manager"

            ],

            namespace=namespace,

        )

        # There may be other controllers of the joints, but this is the initially-started one

        initial\_joint\_controller\_spawner\_started = Node(

            package="controller\_manager",

            executable="spawner",

            arguments=[

                "joint\_trajectory\_controller",

                "--param-file",

                initial\_joint\_controllers,

                "-c",

                f"{namespace}/controller\_manager",

            ],

        )

        gripper\_joint\_controller\_spawner\_started = Node(

            package="controller\_manager",

            executable="spawner",

            arguments=[

                "gripper\_controller",

                "--param-file",

                initial\_joint\_controllers,

                "-c",

                f"{namespace}/controller\_manager",

            ],

        )

        x\_pos = "-0.5" if namespace == "/left" else "0.5"

        gazebo\_spawn\_robot = Node(

            package="ros\_gz\_sim",

            executable="create",

            arguments=[

                "-name", f"{namespace}\_arm", "-topic",

                f"{namespace}/robot\_description", "-x", x\_pos, "-y", "0", "-z",

                "0"

            ],

            output="screen",

            namespace=namespace,

        )

        stuff\_to\_spawn.extend([

            robot\_state\_publisher\_node,

            joint\_state\_broadcaster\_spawner,

            initial\_joint\_controller\_spawner\_started,

            gripper\_joint\_controller\_spawner\_started,

            gazebo\_spawn\_robot,

        ])

# end of for

    gazebo\_bridge = Node(

        package='ros\_gz\_bridge',

        executable='parameter\_bridge',

        arguments=["/clock@rosgraph\_msgs/msg/Clock[ignition.msgs.Clock"],

        output='screen',

    )

    stuff\_to\_spawn.append(gazebo\_bridge)

    return LaunchDescription([

        ar\_model\_arg,

        gazebo,

        \*stuff\_to\_spawn,

    ])

### Overview

The generate\_launch\_description function is the heart of this launch file - it's like a master blueprint that constructs an entire multi-robot simulation environment.

### 1. Launch Arguments: Making the System Configurable

ar\_model\_arg = DeclareLaunchArgument("ar\_model",

default\_value="mk3",

choices=["mk1", "mk2", "mk3"],

description="Model of AR4")

ar\_model\_config = LaunchConfiguration("ar\_model")

The function begins by establishing user-configurable parameters. Think of DeclareLaunchArgument as creating a command-line option that users can specify when launching the simulation. The choices parameter acts like a validation system - users can only select from the three supported AR4 robot models. This prevents runtime errors from invalid model specifications.

The LaunchConfiguration creates a reference to this argument that can be used throughout the launch file. It's like creating a variable that gets its value from the user's choice, but can be referenced multiple times in different contexts.

### 2. Setting Up the Simulation World

world = os.path.join(get\_package\_share\_directory('annin\_ar4\_gazebo'),

'worlds', 'empty.world')

gazebo = IncludeLaunchDescription(PythonLaunchDescriptionSource(

[FindPackageShare("ros\_gz\_sim"), "/launch", "/gz\_sim.launch.py"]),

launch\_arguments={

'gz\_args': f'-r -v 4 {world}',

'on\_exit\_shutdown': 'True'

}.items())

This section constructs the virtual environment where our robots will live. The world file path construction demonstrates robust file handling - it locates the package directory regardless of where ROS is installed on the system, then constructs a path to an empty simulation world.

The Gazebo launch inclusion is particularly sophisticated. Rather than reinventing the wheel, it leverages an existing, well-tested Gazebo launch file from the ros\_gz\_sim package. The launch arguments passed to Gazebo include -r (auto-start the simulation), -v 4 (verbose logging level 4), and the world file path. The on\_exit\_shutdown: 'True' ensures that when this launch file terminates, Gazebo shuts down cleanly rather than leaving zombie processes.

IncludeLaunchDescription is a powerful composition mechanism that lets you embed entire launch files within other launch files. Think of it as being able to "copy and paste" a complete launch configuration from another file directly into your current launch description.

**The Core Concept: Launch File Composition**

When you use IncludeLaunchDescription, you're essentially saying "take everything defined in that other launch file and make it part of my current launch file."

Note: The empty.world file is typically provided by the annin\_ar4\_gazebo package. We can install them using:

sudo apt-get install ros-jazzy-annin-ar4-description ros-jazzy-annin-ar4-gazebo ros-jazzy-annin-ar4-moveit-config

### 3. The Multi-Robot Loop: Elegant Code Reuse

stuff\_to\_spawn = []

for namespace in ["/left", "/right"]:

It uses a loop to create identical configurations for two different robot instances. The stuff\_to\_spawn list acts as a collection basket for all the nodes that need to be launched.

The namespace approach is crucial for multi-robot systems. By giving each robot its own namespace ("/left" and "/right"), all of their ROS communications become isolated. It's like giving each robot its own radio frequency so their control signals don't interfere with each other.

### 4. Dynamic Robot Description Generation

initial\_joint\_controllers = ControllerConfigSubstitution(

PathJoinSubstitution([

FindPackageShare("annin\_ar4\_driver"), "config",

"controllers.yaml"

]))

robot\_description\_content = Command([

PathJoinSubstitution([FindExecutable(name="xacro")]),

" ",

PathJoinSubstitution([

FindPackageShare("annin\_ar4\_description"),

"urdf",

"ar\_gazebo.urdf.xacro",

]),

" ",

"ar\_model:=",

ar\_model\_config,

" ",

f"namespace:={namespace}",

" ",

"simulation\_controllers:=",

initial\_joint\_controllers,

])

This section showcases the power of ROS 2's substitution system. The initial\_joint\_controllers uses our custom substitution class to create namespace-specific controller configurations. This is that clever template system we discussed earlier in action.

The robot\_description\_content construction builds a command line that will execute the xacro tool (a macro processor for robot descriptions) with specific parameters. Think of xacro as a robot description compiler that takes template files and generates complete robot models.

The command construction demonstrates how multiple dynamic values get combined: the robot model selection from the launch argument, the current namespace from the loop, and the dynamically generated controller configuration. This creates a unique robot description for each arm while sharing the same underlying templates.

### 5. State Publishing and Monitoring

robot\_state\_publisher\_node = Node(

package="robot\_state\_publisher",

executable="robot\_state\_publisher",

output="both",

parameters=[

robot\_description,

{

"use\_sim\_time": True

},

],

namespace=namespace,

remappings=[("/tf", "tf"), ("/tf\_static", "tf\_static")],

)

The robot **state publisher** is like the robot's "nervous system" - it continuously broadcasts the current state of all joints to the rest of the ROS system. The use\_sim\_time: True parameter is crucial for simulation environments, ensuring all timing is synchronized with the simulation clock rather than real-world time.

The remappings handle coordinate frame transformations. In robotics, every part of a robot exists in a coordinate system relative to other parts. The tf (transform) system manages these relationships, and the remappings ensure that transform data flows correctly between namespaces.

### 6. Controller System Architecture

joint\_state\_broadcaster\_spawner = Node(

package="controller\_manager",

executable="spawner",

arguments=[

"joint\_state\_broadcaster", "-c",

f"{namespace}/controller\_manager"

],

namespace=namespace,

)

initial\_joint\_controller\_spawner\_started = Node(

package="controller\_manager",

executable="spawner",

arguments=[

"joint\_trajectory\_controller",

"--param-file",

initial\_joint\_controllers,

"-c",

f"{namespace}/controller\_manager",

],

)

gripper\_joint\_controller\_spawner\_started = Node(

package="controller\_manager",

executable="spawner",

arguments=[

"gripper\_controller",

"--param-file",

initial\_joint\_controllers,

"-c",

f"{namespace}/controller\_manager",

],

)

This section reveals the sophisticated control architecture. Think of the controller\_manager as a central dispatch system that coordinates different types of controllers. Each controller has a specific job:

* The **joint\_state\_broadcaster** monitors and reports joint positions - like having sensors throughout the robot that report "where am I right now?"
* The **joint\_trajectory\_controller** handles coordinated movement of the main arm joints, planning smooth paths between positions.
* The **gripper\_controller** manages the end-effector separately, allowing fine control over grasping operations.

The -c argument specifies which controller manager to connect to, using the namespace-specific path. This ensures each robot's controllers connect to the correct control system.

**Where the Controller Actually Lives**

The joint\_trajectory\_controller isn't explicitly defined in this launch file. Instead, it's referenced by name when the spawner tries to load it

**The Configuration File Connection**

The real controller configuration comes from that controllers.yaml file that gets processed by our ControllerConfigSubstitution. This YAML file would contain something like:

controller\_manager:

ros\_\_parameters:

update\_rate: 100

joint\_trajectory\_controller:

type: joint\_trajectory\_controller/JointTrajectoryController

joints:

- joint\_1

- joint\_2

- joint\_3

- joint\_4

- joint\_5

- joint\_6

*# ... other parameters*

Ulisses’ Note: We can find this yaml file at:

C:\Dropbox\Code Robotics Books\ar4\_ros\_driver-main\annin\_ar4\_driver\config\

and

C:\Dropbox\Code Robotics Books\ar4\_ros\_driver-main\annin\_ar4\_moveit\_config\config

**How the Controller Gets Loaded**

Here's the actual sequence:

1. **Controller Definition**: The joint\_trajectory\_controller is defined as a controller type in the ROS control framework, not in this specific launch file
2. **Configuration Loading**: The --param-file argument tells the spawner to load controller parameters from the processed controllers.yaml file
3. **Controller Instantiation**: The spawner looks up "joint\_trajectory\_controller" in that configuration file and creates an instance based on those parameters
4. **Registration**: The controller gets registered with the controller manager for that namespace

**Why This Approach?**

This separation is actually elegant design:

* **Configuration vs. Code**: The controller behavior is defined in configuration files, making it easy to modify without changing launch files
* **Reusability**: The same controller type can be used with different configurations for different robots

### 7. Physical Robot Placement

x\_pos = "-0.5" if namespace == "/left" else "0.5"

gazebo\_spawn\_robot = Node(

package="ros\_gz\_sim",

executable="create",

arguments=[

"-name", f"{namespace}\_arm", "-topic",

f"{namespace}/robot\_description", "-x", x\_pos, "-y", "0", "-z",

"0"

],

output="screen",

namespace=namespace,

)

This demonstrates spatial reasoning in code. The conditional placement puts the left robot at x=-0.5 and the right robot at x=0.5, creating a symmetric workspace. The spawn command tells Gazebo to create a physical robot instance using the robot description we generated earlier.

The topic specification (f"{namespace}/robot\_description") ensures each robot gets spawned using its own customized description, maintaining the namespace isolation throughout the entire system.

### 8. Append (extend) data to stuff\_to\_spawn

stuff\_to\_spawn.extend([

            robot\_state\_publisher\_node,

            joint\_state\_broadcaster\_spawner,

            initial\_joint\_controller\_spawner\_started,

            gripper\_joint\_controller\_spawner\_started,

            gazebo\_spawn\_robot,

        ])

stuff\_to\_spawn will contain all the joints for both arms (because it’s inside the arms loop)

### 9. System Integration and Timing

gazebo\_bridge = Node(

package='ros\_gz\_bridge',

executable='parameter\_bridge',

arguments=["/clock@rosgraph\_msgs/msg/Clock[ignition.msgs.Clock"],

output='screen',

)

The final bridge component solves a critical timing synchronization problem. ROS and Gazebo use different internal clock systems, and this bridge ensures they stay synchronized. Think of it as a translator that converts timing information between two different time-keeping systems.

### 10. The Final Assembly

return LaunchDescription([

ar\_model\_arg,

gazebo,

\*stuff\_to\_spawn,

])

The return statement demonstrates Python's unpacking operator (\*) elegantly combining all the components we've built. The launch argument comes first, followed by the Gazebo environment, then all the robot-specific nodes we collected in our loop.

This function ultimately creates a complete, coordinated system where two robots can operate independently while sharing the same simulation environment - a sophisticated achievement that would require hundreds of lines of code if written without this modular, substitution-based architecture.

#### Q: How the unpacking operator works?

**What's Actually Happening**

return LaunchDescription([

ar\_model\_arg,

gazebo,

\*stuff\_to\_spawn,

])

Let's break this down step by step. At the point of the return statement, stuff\_to\_spawn is a list containing multiple Node objects. Let's say it looks something like this:

stuff\_to\_spawn = [

robot\_state\_publisher\_node\_left,

joint\_state\_broadcaster\_spawner\_left,

initial\_joint\_controller\_spawner\_left,

gripper\_joint\_controller\_spawner\_left,

gazebo\_spawn\_robot\_left,

robot\_state\_publisher\_node\_right,

joint\_state\_broadcaster\_spawner\_right,

initial\_joint\_controller\_spawner\_right,

gripper\_joint\_controller\_spawner\_right,

gazebo\_spawn\_robot\_right,

gazebo\_bridge,

]

**Without the Unpacking Operator**

If we didn't use the \* operator, we'd be creating a list that looks like this:

return LaunchDescription([

ar\_model\_arg,

gazebo,

stuff\_to\_spawn, *# This would be a nested list!*

])

This would create a structure like:

[

ar\_model\_arg,

gazebo,

[node1, node2, node3, node4, node5, ...] *# Nested list - not what we want!*

]

The LaunchDescription expects a flat list of launch actions, not a list containing another list.

**With the Unpacking Operator**

The \* operator "unpacks" or "spreads" the contents of stuff\_to\_spawn directly into the surrounding list:

return LaunchDescription([

ar\_model\_arg,

gazebo,

\*stuff\_to\_spawn, *# This unpacks to: node1, node2, node3, node4, node5, ...*

])

This creates a flat structure like:

[

ar\_model\_arg,

gazebo,

robot\_state\_publisher\_node\_left,

joint\_state\_broadcaster\_spawner\_left,

initial\_joint\_controller\_spawner\_left,

gripper\_joint\_controller\_spawner\_left,

gazebo\_spawn\_robot\_left,

robot\_state\_publisher\_node\_right,

joint\_state\_broadcaster\_spawner\_right,

initial\_joint\_controller\_spawner\_right,

gripper\_joint\_controller\_spawner\_right,

gazebo\_spawn\_robot\_right,

gazebo\_bridge,

]

## ControllerConfigSubstitution

### Code

class ControllerConfigSubstitution(Substitution):

    """Substitution that fills out tf\_prefix in controllers.yaml."""

    def \_\_init\_\_(self,

                 file\_path: Substitution,

                 tf\_prefix: Substitution | str = ""):

        super().\_\_init\_\_()

        self.\_file\_path = file\_path

        self.\_tf\_prefix = tf\_prefix

    def perform(self, context):

        # Evaluate the file path and namespace substitutions

        file\_path\_val = self.\_file\_path.perform(context)

        if isinstance(self.\_tf\_prefix, str):

            tf\_prefix\_val = self.\_tf\_prefix

        else:

            tf\_prefix\_val = self.\_tf\_prefix.perform(context)

        with open(file\_path\_val, "r") as f:

            content = f.read()

        content = content.replace('$(var tf\_prefix)', tf\_prefix\_val)

        temp\_file = tempfile.NamedTemporaryFile(delete=False, suffix=".yaml")

        temp\_file.write(content.encode("utf-8"))

        temp\_file.close()

        return temp\_file.name

### Overview

The ControllerConfigSubstitution class demonstrates dynamic configuration file processing.

**The Problem It Solves**

Imagine you have a robot configuration template that needs to work with multiple robots, each having different namespaces or prefixes. Rather than maintaining separate configuration files for each robot, this class creates a "smart template" that can fill in robot-specific details at runtime.

**Class Architecture and Inheritance**

class ControllerConfigSubstitution(Substitution):

This class inherits from ROS 2's Substitution base class, which means it plugs into the launch system's dynamic value resolution framework. Think of substitutions as "placeholder values" that get calculated when the launch file actually runs, rather than when it's written.

### The Constructor: Setting Up the Template

def \_\_init\_\_(self,

file\_path: Substitution,

tf\_prefix: Substitution | str = ""):

super().\_\_init\_\_()

self.\_file\_path = file\_path

self.\_tf\_prefix = tf\_prefix

The constructor accepts two key parameters. The file\_path parameter points to the template configuration file - typically a YAML file containing controller settings with placeholder variables. Notice that this can be a Substitution object itself, meaning the file path could be dynamically determined based on other launch parameters.

The tf\_prefix parameter, bu using the type annotation Substitution | str = "" tells us it can accept either a static string or another substitution that gets resolved dynamically. This flexibility is crucial because you might want to hardcode a prefix in some situations, while in others you might want it determined by launch arguments.

### The Core Logic: Dynamic File Processing

def perform(self, context):

*# Evaluate the file path and namespace substitutions*

file\_path\_val = self.\_file\_path.perform(context)

if isinstance(self.\_tf\_prefix, str):

tf\_prefix\_val = self.\_tf\_prefix

else:

tf\_prefix\_val = self.\_tf\_prefix.perform(context)

The perform method is where the magic happens. This method gets called by the ROS launch system when it needs the actual value of this substitution. The context parameter contains all the runtime information needed to resolve other substitutions.

The first line resolves the file path. If self.\_file\_path was itself a substitution (like PathJoinSubstitution), calling perform(context) on it returns the actual file path string.

The conditional logic for tf\_prefix checks whether the prefix is already a plain string or needs to be resolved as a substitution. This dual-mode handling makes the class more flexible and prevents runtime errors.

### File Processing and Template Replacement

with open(file\_path\_val, "r") as f:

content = f.read()

content = content.replace('$(var tf\_prefix)', tf\_prefix\_val)

Here's where the actual template processing occurs. The code reads the entire configuration file into memory, then performs a simple but powerful string replacement. The pattern $(var tf\_prefix) is a ROS convention for variable substitution within configuration files.

This approach is elegant because it leverages familiar templating concepts. Rather than requiring complex XML processing or specialized parsing, it uses straightforward string replacement that's easy to understand and debug.

### Temporary File Creation and Management

temp\_file = tempfile.NamedTemporaryFile(delete=False, suffix=".yaml")

temp\_file.write(content.encode("utf-8"))

temp\_file.close()

return temp\_file.name

This final section demonstrates sophisticated temporary file handling. The delete=False parameter is crucial - it tells Python not to automatically delete the file when the file handle closes. This is necessary because other ROS processes will need to read this file later.

The suffix=".yaml" ensures the temporary file has the correct extension, which can be important for tools that determine file type based on extension.

The explicit encoding to UTF-8 ensures consistent text handling across different systems and locales. The file is immediately closed after writing, which is good practice for releasing system resources.

Finally, returning temp\_file.name provides the path to the newly created, customized configuration file.

## Equivalent XML

<?xml version="1.0"?>

<launch>

    <!-- Model argument -->

    <arg name="ar\_model" default="mk3" description="Model of AR4">

        <allowed\_values>mk1 mk2 mk3</allowed\_values>

    </arg>

    <!-- Include Gazebo -->

    <let name="world\_path" value="$(find-pkg-share annin\_ar4\_gazebo)/worlds/empty.world"/>

    <include file="$(find-pkg-share ros\_gz\_sim)/launch/gz\_sim.launch.py">

        <arg name="gz\_args" value="-r -v 4 $(var world\_path)"/>

        <arg name="on\_exit\_shutdown" value="true"/>

    </include>

    <!-- Left arm -->

    <group ns="/left">

        <let name="tf\_prefix" value="/left"/>

        <let name="x\_pos" value="-0.5"/>

        <node pkg="robot\_state\_publisher" exec="robot\_state\_publisher" output="both">

            <param name="robot\_description" value="$(command 'xacro $(find-pkg-share annin\_ar4\_description)/urdf/ar\_gazebo.urdf.xacro ar\_model:=$(var ar\_model) namespace:=/left simulation\_controllers:=$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml')"/>

            <param name="use\_sim\_time" value="true"/>

            <remap from="/tf" to="tf"/>

            <remap from="/tf\_static" to="tf\_static"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="joint\_state\_broadcaster\_spawner">

            <arg name="controller\_name" value="joint\_state\_broadcaster"/>

            <arg name="controller\_manager" value="/left/controller\_manager"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="trajectory\_controller\_spawner">

            <arg name="controller\_name" value="joint\_trajectory\_controller"/>

            <arg name="controller\_manager" value="/left/controller\_manager"/>

            <arg name="params\_file" value="$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="gripper\_controller\_spawner">

            <arg name="controller\_name" value="gripper\_controller"/>

            <arg name="controller\_manager" value="/left/controller\_manager"/>

            <arg name="params\_file" value="$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml"/>

        </node>

        <node pkg="ros\_gz\_sim" exec="create" output="screen">

            <arg name="name" value="/left\_arm"/>

            <arg name="topic" value="/left/robot\_description"/>

            <arg name="x" value="-0.5"/>

            <arg name="y" value="0"/>

            <arg name="z" value="0"/>

        </node>

    </group>

    <!-- Right arm -->

    <group ns="/right">

        <let name="tf\_prefix" value="/right"/>

        <let name="x\_pos" value="0.5"/>

        <node pkg="robot\_state\_publisher" exec="robot\_state\_publisher" output="both">

            <param name="robot\_description" value="$(command 'xacro $(find-pkg-share annin\_ar4\_description)/urdf/ar\_gazebo.urdf.xacro ar\_model:=$(var ar\_model) namespace:=/right simulation\_controllers:=$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml')"/>

            <param name="use\_sim\_time" value="true"/>

            <remap from="/tf" to="tf"/>

            <remap from="/tf\_static" to="tf\_static"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="joint\_state\_broadcaster\_spawner">

            <arg name="controller\_name" value="joint\_state\_broadcaster"/>

            <arg name="controller\_manager" value="/right/controller\_manager"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="trajectory\_controller\_spawner">

            <arg name="controller\_name" value="joint\_trajectory\_controller"/>

            <arg name="controller\_manager" value="/right/controller\_manager"/>

            <arg name="params\_file" value="$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml"/>

        </node>

        <node pkg="controller\_manager" exec="spawner" name="gripper\_controller\_spawner">

            <arg name="controller\_name" value="gripper\_controller"/>

            <arg name="controller\_manager" value="/right/controller\_manager"/>

            <arg name="params\_file" value="$(find-pkg-share annin\_ar4\_driver)/config/controllers.yaml"/>

        </node>

        <node pkg="ros\_gz\_sim" exec="create" output="screen">

            <arg name="name" value="/right\_arm"/>

            <arg name="topic" value="/right/robot\_description"/>

            <arg name="x" value="0.5"/>

            <arg name="y" value="0"/>

            <arg name="z" value="0"/>

        </node>

    </group>

    <!-- Bridge for clock -->

    <node pkg="ros\_gz\_bridge" exec="parameter\_bridge" output="screen">

        <arg name="arguments" value="/clock@rosgraph\_msgs/msg/Clock[ignition.msgs.Clock"/>

    </node>

</launch>

## Comments XML

A few important notes about the XML version:

1. It doesn't have the custom [ControllerConfigSubstitution](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) functionality that the Python version has. You might need to handle the tf\_prefix substitution in controllers.yaml differently.
2. The XML format is more declarative but less flexible than Python when it comes to programmatic generation of launch configuration.
3. The command substitution syntax is slightly different in XML format.
4. The namespacing is handled through group tags with the ns attribute.
5. While this XML version should work for most cases, some complex substitutions and conditional logic might be easier to handle in the Python version.

The main advantage of using XML is that it's more readable and familiar to those coming from ROS 1. However, the Python format offers more flexibility and programming capabilities, which is why it's becoming more common in ROS 2.