HandyBot - Code

# Github

<https://github.com/ycheng517/tabletop-handybot>

# tabletop\_handybot\_node.py

## Imports

### Code

import rclpy

import tf2\_ros

from cv\_bridge import CvBridge

from geometry\_msgs.msg import Pose, PoseStamped

from sensor\_msgs.msg import Image, PointCloud2

from sensor\_msgs.msg import JointState

from rclpy.callback\_groups import ReentrantCallbackGroup

import json

import os

import time

from functools import cached\_property

from typing import List

import cv2

import numpy as np

import open3d as o3d

import openai

import supervision as sv

import torch

import torchvision

from groundingdino.util.inference import Model

from openai.types.beta import Assistant

from openai.types.beta.threads import RequiredActionFunctionToolCall

from rclpy.duration import Duration

from rclpy.executors import MultiThreadedExecutor

from rclpy.callback\_groups import MutuallyExclusiveCallbackGroup

from rclpy.node import Node

from rclpy.time import Time

from scipy.spatial.transform import Rotation

from segment\_anything import SamPredictor, sam\_model\_registry

from std\_msgs.msg import Int64, String

from pymoveit2 import GripperInterface, MoveIt2

from .openai\_assistant import get\_or\_create\_assistant

from .point\_cloud\_conversion import point\_cloud\_to\_msg

### rclpy

rclpy is the Python client library for **ROS 2** (Robot Operating System 2). It's a set of tools that lets you write ROS 2 nodes and other components using Python.

**Core Functions of rclpy**

rclpy provides the building blocks for creating any ROS 2 application in Python. Its key functions include:

* **Node Management**: It allows you to create, initialize, and destroy **nodes**. A node is a single, executable process in a ROS 2 system that performs a specific task. For example, a robot might have one node for controlling motors and another for reading sensor data.
* **Inter-Node Communication**: rclpy is the key to communication within the ROS 2 network. It enables the creation of:
  + **Publishers and Subscribers**: Used to send and receive messages on named **topics**. This is the most common way for nodes to share data asynchronously.
  + **Clients and Services**: Used for synchronous request-reply communication. A client sends a request to a service, and the service sends a response back.
  + **Actions**: For long-running, goal-oriented tasks, actions provide a way to send a goal, receive feedback as the task progresses, and get a final result.
* **Utilities**: It also includes utility functions for managing ROS 2's lifecycle, like rclpy.init() to start the system and rclpy.spin() to keep a node running and processing messages.

### tf2\_ros

tf2\_ros is a core ROS (Robot Operating System) package that provides the bindings for the **tf2** library. tf2 is a transform library that handles coordinate frames for a robot and its environment. It's a crucial tool in robotics, allowing you to manage multiple coordinate systems and transform data between them.

**Key Concepts**

* **Coordinate Frames**: In robotics, everything is located relative to a coordinate frame. A robot might have a frame for its base, a frame for its gripper, and a camera might have its own frame. The world or map also has a frame. These are all interconnected.
* **Transformations**: A transformation defines the relationship (translation and rotation) between two coordinate frames. tf2 keeps a buffered tree of these transformations over time.
* **Time Travel**: One of tf2's most powerful features is "time travel," which allows you to query the pose of a frame at a past point in time. This is essential for applications where sensor data needs to be synchronized with the robot's historical position.

**Functionality**

The tf2\_ros package provides the ROS bindings for the tf2 library, enabling ROS nodes to participate in the transform tree. It includes:

* **Broadcasters**: Nodes that publish transformations. For example, a node controlling a robot's wheels would publish the transform from the base frame to the wheel frames.
* **Listeners**: Nodes that listen for and buffer transformations. A listener can then query the buffer to find the relationship between any two frames at a specific time.

### cv\_bridge => CvBridge

cv\_bridge library is a crucial tool in the **ROS** (Robot Operating System) ecosystem, as it provides a bridge between two different data formats:

1. **ROS image messages**: The standard format for images within ROS.
2. **OpenCV image data**: The format used by the widely-used **OpenCV** library (cv2), which represents images as NumPy arrays.

CvBridge makes it possible to convert images from one format to the other, allowing you to process images from ROS topics using powerful computer vision algorithms in OpenCV.

**How it Works**

The CvBridge class has two primary methods:

* **imgmsg\_to\_cv2()**: This method takes a ROS Image or CompressedImage message and converts it into an OpenCV image, which is a **NumPy array**. This is what you would use to get an image from a ROS topic and then apply computer vision algorithms to it.
* **cv2\_to\_imgmsg()**: This method takes an OpenCV image (NumPy array) and converts it into a ROS Image message. This is used when you want to publish an image that has been processed by OpenCV back onto a ROS topic for other nodes to use.

### geometry\_msgs.msg => Pose, PoseStamped

from geometry\_msgs.msg import Pose, PoseStamped is a Python statement used in the **ROS (Robot Operating System)** framework. It imports two message types, **Pose** and **PoseStamped**, from the geometry\_msgs package. These messages are fundamental for representing and communicating positional data in robotics. 🤖

**Pose Message**

The Pose message defines the position and orientation of an object in 3D space. It contains two main components:

* **Point position**: A Point message that specifies the object's X, Y, and Z coordinates.
* **Quaternion orientation**: A Quaternion message that defines the object's orientation in terms of its rotation around the X, Y, and Z axes. This is a more robust way to represent 3D rotations than Euler angles.

A Pose message is purely a geometric primitive — it describes a location and orientation but doesn't include any information about the coordinate frame it's relative to or a timestamp.

**PoseStamped Message**

The PoseStamped message is an enhanced version of Pose. It adds two important pieces of information:

* **std\_msgs/Header header**: This header contains a stamp (a timestamp) and a frame\_id (the name of the coordinate frame to which the pose is relative).
* **Pose pose**: This is the actual Pose message containing the position and orientation.

The PoseStamped message is the standard for communicating pose data in ROS, as it provides a complete and unambiguous description of where an object is located, when the measurement was taken, and in what coordinate system. This is crucial for applications that involve multiple sensors or robot parts moving over time.

In short, Pose is the raw geometric data, while PoseStamped provides the essential context (time and coordinate frame) needed for a robotic system to make sense of that data.

### sensor\_msgs.msg => Image, PointCloud2

from sensor\_msgs.msg import Image, PointCloud2 is a Python statement used in the **ROS (Robot Operating System)** framework. It imports two standard message types from the sensor\_msgs package, which are crucial for handling sensor data in robotics.

**Image Message**

The Image message is the standard ROS format for representing a camera or image sensor feed. It encapsulates all the necessary information about an image, including:

* **std\_msgs/Header**: This contains a timestamp and a frame\_id, which specifies the coordinate frame the image was captured from.
* **height and width**: The dimensions of the image in pixels.
* **encoding**: The format of the image data (e.g., rgb8 for 8-bit color, mono8 for 8-bit grayscale).
* **data**: The raw image data, which is a one-dimensional array of bytes.

This message is used by nodes that publish or subscribe to camera feeds. For instance, a node controlling a camera on a robot would publish Image messages, and another node performing image processing would subscribe to them.

**PointCloud2 Message**

The PointCloud2 message is the standard ROS format for representing a **point cloud**. A point cloud is a set of data points in a 3D coordinate system, typically generated by sensors like LiDAR or 3D cameras. It contains:

* **std\_msgs/Header**: Similar to the Image message, it includes a timestamp and frame\_id.
* **height and width**: These dimensions can either represent an organized (like an image) or unorganized point cloud.
* **fields**: A description of the data fields for each point (e.g., x, y, z for coordinates, rgb for color).
* **data**: The raw point data.

The PointCloud2 message is essential for applications that require a 3D understanding of the environment, such as object recognition, mapping, and navigation.

In short, Image is for 2D visual data, while PointCloud2 is for 3D spatial data. Both are fundamental for any robot that perceives its environment using cameras or 3D sensors.

### sensor\_msgs.msg => JointState

from sensor\_msgs.msg import JointState is a Python statement used in the **ROS (Robot Operating System)** framework. It imports the **JointState** message type from the sensor\_msgs package. This message is the standard way to communicate the state of a robot's joints, such as a robotic arm or a wheeled robot.

**What is a JointState Message?**

The JointState message contains all the essential information about the joints of a robot at a single point in time. It's used by nodes that publish joint data (like motor controllers or simulators) and by nodes that need to read that data (like a robot's inverse kinematics solver or a visualization tool).

The message has several key fields:

* **std\_msgs/Header**: This provides a timestamp to indicate when the joint state was recorded and a frame\_id (though JointState is typically not tied to a specific frame).
* **name**: A list of strings, where each string is the name of a joint in the robot (e.g., 'joint1', 'gripper\_joint').
* **position**: A list of numbers representing the joint positions. This is the most common field used and can be in radians for revolute joints or meters for prismatic joints.
* **velocity**: An optional list of numbers representing the joint velocities (e.g., radians per second).
* **effort**: An optional list of numbers representing the effort or torque applied to each joint.

**Common Use Cases**

The JointState message is fundamental to robotics applications. For example:

* A motor driver node would periodically publish JointState messages containing the current position of the robot's joints.
* A visualization tool like **RViz** subscribes to JointState messages to display the robot's model in its current configuration.
* A motion planning or control node would subscribe to JointState messages to get feedback on the robot's current pose and plan the next movement.

In essence, JointState is the standard "snapshot" of a robot's joint configuration, allowing various components of a ROS system to share a common understanding of the robot's physical state.

### rclpy.callback\_groups => ReentrantCallbackGroup

The statement from rclpy.callback\_groups import ReentrantCallbackGroup imports a specific class, ReentrantCallbackGroup, from the rclpy.callback\_groups module in ROS 2. This class is a type of callback group that allows for **reentrant execution**, meaning multiple callbacks can run concurrently on the same executor.

**What Is a Callback Group?**

In ROS 2, a **callback group** manages the execution of callbacks (functions that respond to events like new messages, timer expirations, or service requests) within a node. By default, a single-threaded executor processes callbacks sequentially. This can be problematic if a callback takes a long time to execute, as it can block other callbacks from running, leading to delays.

**The Problem with Non-Reentrant Groups**

By default, callback groups in ROS 2 are **non-reentrant**. This means that when one callback from the group is being processed by the executor, no other callback from that same group can run at the same time. If a long-running callback is part of a non-reentrant group, it will effectively "lock" the group, preventing any other callbacks in that group from executing until the first one is finished.

**How ReentrantCallbackGroup Helps**

The ReentrantCallbackGroup solves this problem by allowing multiple callbacks within the group to execute **in parallel**. It tells the executor that it's safe to run these callbacks concurrently, even if the executor is single-threaded, as long as the callbacks themselves don't interfere with each other. This is particularly useful for tasks that involve network I/O or waiting for external resources, as the executor can switch to another callback while one is waiting, improving responsiveness and throughput.

For example, imagine a node that subscribes to two different topics: one for sensor data and another for configuration updates. If both callbacks are in a non-reentrant group and the sensor data callback is slow, the configuration update callback will have to wait. By placing them in a ReentrantCallbackGroup, the executor can handle both simultaneously.

**When to Use a ReentrantCallbackGroup**

Use a ReentrantCallbackGroup when:

* You have callbacks that don't share or modify shared resources (i.e., they are thread-safe).
* Your callbacks might take a significant amount of time to execute.
* You need to improve the parallelism and responsiveness of your ROS 2 node.

If your callbacks access shared data, you must use proper synchronization mechanisms like **mutexes** to prevent race conditions. The ReentrantCallbackGroup doesn't automatically make your code thread-safe; it only provides the ability for concurrency.

### os

The **'os'** module provides a way of using operating system-dependent functionality like reading or writing to the file system. It allows you to interact with the underlying operating system that your Python script is running on.

**Key Functionality**

The **'os'** module provides a wide range of functions for interacting with the file system and managing system processes. Some of its most common uses include:

* **File and Directory Operations**: You can create, delete, rename, and check for the existence of files and folders. For example, os.mkdir('new\_folder') creates a new directory, and os.remove('file.txt') deletes a file.
* **Path Manipulation**: It offers functions to join, split, and normalize file paths, making your code portable across different operating systems (like Windows, macOS, and Linux). For instance, os.path.join('folder', 'file.txt') correctly combines a directory and filename.
* **Environment Variables**: You can access and modify system **environment variables**, which are key-value pairs that store information about the system and user environment. os.environ provides a dictionary-like object to access these variables.
* **Executing System Commands**: The os.system() function allows you to execute shell commands directly from your Python script, such as os.system('ls') on Linux or os.system('dir') on Windows to list files.

The **'os'** module is fundamental for any Python program that needs to interact with the operating system, providing a consistent interface to perform low-level tasks without needing to write platform-specific code

### time

The 'time' module provides various functions for handling time-related tasks, such as measuring elapsed time and pausing your program's execution.

**Key Functions**

The 'time' module is widely used for:

* **Pausing Execution**: The most common function is time.sleep(). It's used to halt your program for a specified number of seconds. For example, time.sleep(5) will make the program wait for 5 seconds before continuing. This is useful for things like managing API request rates or simulating delays.
* **Getting the Current Time**: The time.time() function returns the current time as a floating-point number, which represents the number of seconds that have passed since the "epoch." The epoch is a fixed point in time, which is January 1, 1970, at 00:00:00 (UTC) on most systems.
* **Measuring Performance**: By getting the time before and after a block of code, you can measure how long it takes to execute. This is a simple and effective way to **benchmark** your code and find performance bottlenecks.

**Time vs. Datetime**

While the 'time' module is great for measuring time and delays, the built-in **'datetime'** module is typically preferred for more complex tasks involving dates and times, such as:

* Formatting dates and times into human-readable strings (e.g., "August 14, 2025").
* Performing calendar-related calculations (e.g., finding the date in two weeks).

Both modules work with time, but they serve different purposes. The 'time' module focuses more on the **concept of time itself** and simple operations, while the 'datetime' module is geared toward **calendar-based operations**.

### functools => cached\_property

from functools import cached\_property is a decorator in Python that is used to cache the result of a method on an instance. It's part of the functools module, which provides higher-order functions and operations on callable objects.

**How It Works**

When a method is decorated with @cached\_property, it works as follows:

1. **First Access**: The first time you access the method like a property (without parentheses, e.g., instance.method\_name), the method's code is executed and its return value is calculated.
2. **Caching**: This return value is then stored in the instance's \_\_dict\_\_ under the same name as the method.
3. **Subsequent Access**: Any subsequent access to the same property on that instance will not re-run the method. Instead, it will immediately return the cached value.

**Why Use cached\_property?**

cached\_property is particularly useful for optimizing a class when:

* **Calculations are expensive**: The method involves a resource-intensive operation, such as a complex calculation, a database query, or a network request. Caching the result prevents this expensive operation from being performed multiple times.
* **The result is constant for the instance's lifetime**: The value of the property is dependent only on the state of the instance and does not change after its initial calculation.

### cv2

**'cv2'** module, which is the main module for **OpenCV** (Open Source Computer Vision Library). This library is a powerful and widely-used tool for computer vision, machine learning, and image processing.

**Key Capabilities of OpenCV**

Once you've imported cv2, you gain access to thousands of functions that allow you to perform a vast range of tasks, including:

* **Image and Video Manipulation**: You can read, write, display, and resize images and videos. For instance, cv2.imread('image.jpg') reads an image from a file, and cv2.imshow('window', image) displays it in a window.
* **Feature Detection**: OpenCV can identify and describe unique features in an image, such as corners and edges. This is crucial for tasks like object recognition and image stitching.
* **Object Detection and Tracking**: It's commonly used to detect and track specific objects in a video stream, like faces, hands, or cars. Face detection, for example, is a classic application using OpenCV's pre-trained models.
* **Video Analysis**: You can perform various operations on video streams, such as background subtraction, motion detection, and optical flow analysis.
* **Image Processing**: OpenCV provides functions for filtering, enhancing, and transforming images. This includes operations like blurring, thresholding, and color space conversions.

### numpy

NumPy, which stands for Numerical Python, is a fundamental library for scientific computing and is widely used for working with arrays and matrices.

**Why is NumPy so important?**

The core of the NumPy library is the **ndarray** (N-dimensional array) object. These arrays are more efficient than Python's built-in lists for numerical operations because they are:

* **Faster**: NumPy arrays are stored in a contiguous block of memory, which makes them faster to process.
* **More Compact**: They use less memory than standard Python lists for storing the same data.
* **Vectorized**: NumPy allows you to perform operations on entire arrays at once without having to write explicit loops. This is a powerful feature called **vectorization**, which dramatically simplifies and speeds up numerical computations.

### open3d

Open3d is an open-source library that provides a comprehensive set of tools for working with 3D models, point clouds, and other geometric data types.

**Key Capabilities**

Open3D is highly versatile and is used for a variety of tasks in computer vision, robotics, and scientific computing. Its main functionalities include:

* **3D Data Structures**: It offers efficient data structures for representing 3D data, such as PointCloud for point clouds, TriangleMesh for meshes, and VoxelGrid for volumetric data. This makes it easy to load, save, and manipulate 3D models.
* **3D Reconstruction and Registration**: The library includes algorithms for creating 3D models from 2D images or depth sensors. It's also used for **registration**, which is the process of aligning multiple 3D models or point clouds into a single, cohesive model.
* **Visualization**: Open3D provides powerful and easy-to-use visualization tools. You can render 3D data, create interactive viewers, and animate 3D scenes directly within your Python script.
* **Geometric Processing**: It has functions for a wide range of geometric operations, including filtering, segmentation, and meshing. For example, you can remove noise from a point cloud or simplify a complex mesh.
* **Computer Vision and Machine Learning**: The library integrates with other popular libraries like NumPy and PyTorch, making it a great tool for building pipelines that combine 3D data with machine learning models for tasks like object detection and classification in 3D space.

### supervision

**Supervision** library, a lightweight and open-source toolkit designed to assist developers in building computer vision projects. It provides a variety of utilities to simplify common tasks in object detection, segmentation, and classification, making it easier to prototype and visualize results.

**Key Features and Use Cases**

Supervision library is a helpful tool for anyone working with computer vision models, especially those built using popular frameworks like **YOLO**, **Hugging Face**, and **Ultralytics**. Its main features include:

* **Visualization**: Supervision's core strength is its visualization capabilities. It allows you to easily draw bounding boxes, masks, and labels on images. For example, after running an object detection model, you can use the library to draw the detected objects on the original image with just a few lines of code.
* **Utilities for Annotations**: It provides a standardized way to work with different annotation formats. This simplifies the process of converting between various formats and makes it easier to handle data for training or evaluating models.
* **Data Handling**: The library includes tools to load and save detection results, making it easy to store and retrieve model outputs.
* **Model Integration**: Supervision is designed to be model-agnostic, meaning it can work with the output of almost any computer vision model. This flexibility allows developers to quickly integrate its visualization and utility functions into their existing projects.

In short, import supervision is the entry point to a library that streamlines the development workflow for computer vision applications by providing tools that make it simple to visualize, manage, and process the output of detection and segmentation models.

The **Supervision** library was created by **Piotr Skalski**, who is also the Open-Source Lead at Roboflow. The project is a collaborative effort with the open-source community.

**Unique and Remarkable Features**

Supervision is not a new computer vision framework but rather a toolkit designed to fill a gap in the ecosystem. What makes it unique is its focus on streamlining the development workflow by providing a simple, unified, and model-agnostic interface for common computer vision tasks. Instead of requiring developers to write boilerplate code for things like drawing bounding boxes, converting annotation formats, or managing detections, Supervision provides a set of reusable tools.

Here are some of its remarkable features:

* **Ease of Use**: It has a simple and intuitive API that lets developers quickly visualize model predictions, such as drawing detections or masks on an image, with just a few lines of code. This significantly speeds up prototyping and debugging.
* **Model Agnostic**: It is designed to work with the output of any computer vision model, whether it's from popular libraries like Ultralytics' YOLOv8, Hugging Face, or a custom-trained model. This flexibility allows it to be easily integrated into any existing project.
* **Comprehensive Utilities**: The library includes a wide range of utilities beyond just visualization. These tools handle tasks like non-max suppression (NMS), calculating Intersection over Union (IoU),

### torch

**PyTorch** library, an open-source machine learning framework. It's used for building and training neural networks and is particularly popular for deep learning research and development.

**Key Features of PyTorch**

PyTorch is a powerful and flexible library that stands out for several reasons:

* **Dynamic Computation Graphs**: Unlike some other frameworks, PyTorch uses a dynamic graph. This means the graph is built on the fly as the code is executed. This makes it easier to debug, as you can use standard Python debugging tools, and more flexible for models with variable-length inputs.
* **Tensors**: The fundamental data structure in PyTorch is the **tensor**, which is a multi-dimensional array similar to a NumPy array. Tensors can be used to store data, and they are optimized for numerical operations. What's unique about them is that they can also run on GPUs (Graphics Processing Units), which massively accelerates the training of large neural networks.
* **Autograd**: PyTorch's torch.autograd module automatically computes the gradients of a tensor. This is a core feature that makes backpropagation and training neural networks much simpler, as you don't have to manually calculate derivatives.
* **Deep Learning Modules**: PyTorch provides a rich set of modules in torch.nn for building neural network layers (e.g., convolutional layers, recurrent layers), loss functions, and optimizers.

In essence, import torch is the first step to using PyTorch to build and train machine learning models, from simple linear regressions to complex deep neural networks.

### torchvision

**TorchVision** library is a key part of the PyTorch ecosystem. It's a collection of popular datasets, model architectures, and image transformations specifically designed for computer vision.

**What's in the TorchVision Library?**

The library is a powerful tool for computer vision developers and researchers, as it provides a standardized way to handle common tasks. It is divided into several main components:

* **torchvision.datasets**: This module provides access to a large number of publicly available and pre-processed datasets, such as **MNIST**, **CIFAR-10**, and **ImageNet**. You can load these datasets with a single line of code, which is incredibly useful for training and testing models.
* **torchvision.models**: This module contains pre-trained models for image classification, object detection, and other vision tasks. These models have been trained on large datasets like ImageNet, and you can use them directly or fine-tune them for your specific problem. Examples include popular architectures like **ResNet**, **VGG**, and **MobileNet**.
* **torchvision.transforms**: This module offers a set of common image transformations. These are essential for data augmentation, a technique used to expand the training dataset by applying random changes to images (e.g., cropping, flipping, and rotating). These transformations help improve a model's robustness and generalization.
* **torchvision.utils**: This module provides utility functions, such as make\_grid, which helps you visualize a batch of images by arranging them in a grid.

In short, while torch is the foundation for deep learning, torchvision provides the specialized tools and data you need to get started quickly and efficiently with computer vision tasks.

### groundingdino.util.inference => Model

from groundingdino.util.inference import Model is a Python statement that imports the **Model** class from the **GroundingDINO** library. This class is used to load and run the GroundingDINO model, a powerful vision-language model for **open-set object detection**.

**How GroundingDINO Works**

GroundingDINO is a remarkable model because it can detect objects in an image based on a **text prompt** without needing to be trained on those specific objects.

Here's a breakdown of how it works and what the Model class facilitates:

* **Vision Transformer (DINO)**: The "DINO" part of the name refers to the vision transformer backbone that the model uses to understand the visual features of an image.
* **Text Encoder**: The model also has a text encoder that understands a text prompt (e.g., "a red car").
* **Feature Fusion**: These two components are fused, allowing the model to ground the text prompt to the visual features. The model then generates bounding boxes for the objects described in the text.
* **The Model Class**: When you import the Model class, you are importing the core functionality to use this process. You would typically initialize it with the path to the model's weights and configuration file. Once initialized, you can use the predict\_with\_prompts method to perform inference, providing it with an image and a text prompt to detect objects.

In essence, from groundingdino.util.inference import Model gives you the key to unlock the power of GroundingDINO, allowing you to perform highly flexible object detection by simply describing the objects you want to find.

### scipy.spatial.transform => Rotation

from scipy.spatial.transform import Rotation is a Python statement that imports the **Rotation** class from the **SciPy** library. This class is a powerful tool for representing and manipulating 3D rotations in a robust and computationally efficient way.

**How the Rotation Class Works**

The Rotation class provides a unified interface for working with 3D rotations, regardless of the underlying representation. You can use it to:

* **Create Rotations**: You can initialize a Rotation object from various common rotation representations, including:
  + **Quaternions**: A 4-element vector that is a common and efficient way to represent 3D rotations.
  + **Rotation Matrices**: A 3x3 matrix that describes a rotation.
  + **Euler Angles**: A set of three angles (e.g., yaw, pitch, and roll) that are intuitive but can suffer from issues like Gimbal lock.
  + **Rotation Vectors**: A 3-element vector where the direction specifies the axis of rotation and the magnitude specifies the angle.
* **Convert Between Representations**: Once you have a Rotation object, you can easily convert it to any of the other supported representations. For example, you can create a rotation from a set of Euler angles and then get its equivalent quaternion or rotation matrix.
* **Apply Rotations**: The Rotation class allows you to apply a rotation to a 3D vector or a set of vectors. This is particularly useful for transforming points or coordinates from one frame of reference to another.
* **Combine and Interpolate Rotations**: You can multiply two Rotation objects to combine their effects, or you can interpolate between two rotations to create smooth animations or trajectories.

In essence, from scipy.spatial.transform import Rotation gives you a convenient and reliable way to handle all your 3D rotation needs without having to manually write complex mathematical code for each representation. This makes it an essential tool for robotics, computer graphics, and physics simulations.

### segment\_anything => SamPredictor, sam\_model\_registry

from segment\_anything import SamPredictor, sam\_model\_registry is a Python statement that imports key components of the **Segment Anything Model (SAM)**, a powerful foundation model for image segmentation. This allows a user to programmatically load and use the SAM model for tasks like creating image masks or segmenting objects.

**Key Components**

* **sam\_model\_registry**: This is a registry or a collection of different versions of the SAM model. It simplifies the process of loading a specific model variant (e.g., vit\_h, vit\_l, vit\_b), which have varying sizes and performance characteristics. You can select the appropriate model based on your needs, with larger models generally being more accurate but slower.
* **SamPredictor**: This is the main class used for running inference with the loaded SAM model. It takes an image as input, prepares it for the model, and then uses a set of user-provided **prompts** to generate segmentation masks. These prompts can be points, bounding boxes, or text, allowing the user to specify what they want to segment.

**How It Works**

SAM is unique because it is "promptable." Unlike traditional segmentation models that are trained on a fixed set of classes, SAM can generate a segmentation mask for an object based on a simple input prompt.

The typical workflow using these components is:

1. Load the desired SAM model from the registry using sam\_model\_registry.
2. Initialize the SamPredictor class with the loaded model.
3. Set the image you want to segment in the SamPredictor. This involves an initial one-time processing step for the image.
4. Provide a prompt (e.g., a coordinate point on an object) to the predictor.
5. The predictor then returns the segmentation mask for the object, which can be visualized or further processed.

In short, SamPredictor and sam\_model\_registry are the necessary classes to load the Segment Anything Model and interact with it programmatically to perform image segmentation with user-defined prompts.

*But does Sam works with 2D or 3D ?*

The original Segment Anything Model (SAM) was designed and trained for **2D images**. It cannot directly process 3D data such as point clouds or volumetric scans. However, the success of SAM has led to a lot of research and development in adapting it to 3D, resulting in several notable approaches.

**Adapting SAM for 3D**

Since SAM's core architecture works on 2D images, methods for 3D use typically fall into two categories:

* **2D-to-3D Conversion**: This is the most common approach. It involves using the 2D SAM model to segment individual slices of a 3D volume (like a medical scan or a sequence of video frames) and then combining these 2D masks into a 3D representation. This method is often used for medical imaging, where each slice can be treated as a separate 2D image.
* **Modified SAM Architectures**: Researchers have created new models that either modify SAM's architecture or use it as a component within a larger 3D framework. For example, **SAM3D** and **SAM-Med3D** are models that adapt SAM's principles to natively handle 3D inputs. These adaptations often involve changing the model's encoder to accept volumetric data.

**The Rise of SAM 2**

Meta AI has since released **Segment Anything 2 (SAM 2)**, which is designed to handle both images and **videos**. Because a video can be seen as a sequence of 2D images over time, SAM 2's new capabilities for video segmentation can be seamlessly transferred to 3D segmentation tasks. This represents a more fundamental approach to 3D capability than the earlier 2D-to-3D conversion methods.

The video below offers a complete guide to using the original SAM for various applications. [SAM - Segment Anything Model by Meta AI: Complete Guide](https://www.youtube.com/watch?v=D-D6ZmadzPE) This video is a great resource for getting started with SAM's fundamental 2D capabilities.

### pymoveit2 => GripperInterface, MoveIt2

from pymoveit2 import GripperInterface, MoveIt2 is a Python statement used in the **ROS 2** (Robot Operating System 2) framework. It imports two essential classes from the pymoveit2 library that simplify motion planning and control for robotic arms and grippers.

**MoveIt2 Class**

The **MoveIt2** class is the core of the library. It provides a high-level, Pythonic interface to **MoveIt 2**, the most widely used motion planning framework in ROS 2. Instead of writing complex ROS 2 nodes to handle motion planning, this class lets you send simple commands to a robotic arm.

Key functionalities of MoveIt2 include:

* **Move to Pose**: Command the robot to move its end-effector to a specific position and orientation in 3D space.
* **Move to Joints**: Command the robot to move its joints to a specific configuration (e.g., specific joint angles).
* **Move to Named Pose**: Use pre-defined, named poses (like "home" or "ready") that are configured in the robot's URDF file.
* **Collision Avoidance**: MoveIt2 automatically plans a path that avoids collisions with the robot's body, the environment, and itself.

**GripperInterface Class**

The **GripperInterface** class provides a similar high-level interface specifically for controlling a robot's gripper. It simplifies the process of opening and closing a gripper, which is often a separate action from the main arm movement.

Key functionalities of GripperInterface include:

* **Open Gripper**: Command the gripper to open to a specified position.
* **Close Gripper**: Command the gripper to close to a specified position, often to grip an object.
* **Control Velocity**: Set the speed at which the gripper opens or closes.

In essence, pymoveit2 acts as a simplified Python wrapper around the powerful but complex ROS 2 MoveIt framework. This makes it easier for developers to quickly prototype and control robotic arms and grippers without diving into the complexities of ROS 2 message passing and service calls.

## Initializations

### Code

DEVICE = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# GroundingDINO config and checkpoint

GSA\_PATH = "./Grounded-Segment-Anything/Grounded-Segment-Anything"

GROUNDING\_DINO\_CONFIG\_PATH = os.path.join(

    GSA\_PATH, "GroundingDINO/groundingdino/config/GroundingDINO\_SwinT\_OGC.py")

GROUNDING\_DINO\_CHECKPOINT\_PATH = os.path.join(GSA\_PATH,

                              "groundingdino\_swint\_ogc.pth")

# Segment-Anything checkpoint

SAM\_ENCODER\_VERSION = "vit\_h"

SAM\_CHECKPOINT\_PATH = os.path.join(GSA\_PATH, "sam\_vit\_h\_4b8939.pth")

# Predict classes and hyper-param for GroundingDINO

BOX\_THRESHOLD = 0.4

TEXT\_THRESHOLD = 0.25

NMS\_THRESHOLD = 0.8

### DEVICE = torch.device

DEVICE = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

code snippet you provided determines which device, either a GPU or a CPU, to use for running PyTorch operations.

**Breakdown of the Code**

The line DEVICE = torch.device("cuda" if torch.cuda.is\_available() else "cpu") sets the **DEVICE** variable. It's an **if-else** statement in a single line.

* **torch.cuda.is\_available()** 🤖 checks if a **CUDA-enabled GPU** is present and can be used by PyTorch. **CUDA (Compute Unified Device Architecture)** is a parallel computing platform and programming model created by NVIDIA for general-purpose computing on GPUs.
* If this check returns **True**, the condition is met, and **DEVICE** is assigned torch.device("cuda"). This means PyTorch will run its tensors and models on the GPU, which is **significantly faster** for deep learning tasks.
* If the check returns **False**, a CUDA-enabled GPU is not available. The **else** part of the statement is executed, and **DEVICE** is assigned torch.device("cpu"). This tells PyTorch to use the computer's CPU, which is the default fallback option.

This is a common and **essential best practice** in PyTorch for creating portable code that can run on systems with or without a GPU, without requiring manual changes. The **DEVICE** variable is then used in later code to move tensors and models to the appropriate device (e.g., model.to(DEVICE) or data.to(DEVICE)).

The **torch.device** function in PyTorch is a constructor that creates an object representing the device on which a torch.Tensor or torch.nn.Module is going to be allocated. Think of it as a **device manager** that tells PyTorch where to perform its computations.

**How It Works**

The primary purpose of **torch.device** is to specify a location for tensors and models. This location can be either:

* **CPU**: The central processing unit of your computer. This is the default device and is represented by 'cpu'.
* **GPU**: A graphics processing unit, specifically one with CUDA support, which is represented by 'cuda'. If you have multiple GPUs, you can specify a particular one by its index, such as 'cuda:0' for the first GPU or 'cuda:1' for the second.

Using **torch.device** makes your code flexible and portable. Instead of hardcoding device names like 'cuda:0', you can define a single **DEVICE** variable and use it throughout your code. This allows your program to seamlessly switch between running on a GPU or a CPU without changing any other lines of code, depending on what hardware is available.

## def segment - Prompting SAM with detected boxes

### Code

# Prompting SAM with detected boxes

def segment(sam\_predictor: SamPredictor, image: np.ndarray,

xyxy: np.ndarray) -> np.ndarray:

sam\_predictor.set\_image(image)

result\_masks = []

for box in xyxy:

masks, scores, \_ = sam\_predictor.predict(box=box,

multimask\_output=True)

index = np.argmax(scores)

result\_masks.append(masks[index])

return np.array(result\_masks)

### Explanation

segment, uses a **Segment Anything Model (SAM)** to perform image segmentation based on bounding boxes. It takes a SAM predictor, an image, and an array of bounding boxes as input, and returns an array of the best segmentation masks for each box.

This function bridges object detection and segmentation. *GroundingDINO* provides bounding boxes (xyxy coordinates), and this function uses SAM to create precise pixel-level masks for each detected object. The multimask\_output=True generates several mask options, and we select the most confident one.

**Explanation of the Function**

The segment function processes each bounding box one by one to generate segmentation masks. Here's a breakdown of the steps:

* **Setting the Image**: sam\_predictor.set\_image(image) prepares the SAM model for inference by providing it with the image to be segmented. This step is crucial for the model to process the subsequent prompts (the bounding boxes).
* **Iterating Through Bounding Boxes**: The for box in xyxy: loop processes each bounding box provided in the xyxy NumPy array. The format xyxy indicates that each box is represented by its top-left coordinates (x1​,y1​) and its bottom-right coordinates (x2​,y2​).
* **Predicting Masks**: sam\_predictor.predict() is the core of the function. It takes a bounding box as a prompt and uses the SAM model to predict a set of potential segmentation masks. The multimask\_output=True argument instructs the model to generate multiple, plausible masks for the given box, rather than just a single one. This is particularly useful when there's ambiguity, such as a box containing multiple objects or parts of an object. The function returns three values:
  + masks: A NumPy array containing the generated segmentation masks.
  + scores: An array of confidence scores, where each score corresponds to one of the predicted masks. A higher score indicates a higher probability that the mask correctly identifies the object.
  + \_: The function returns a third value, which is typically not used in this context, hence the underscore.
* **Selecting the Best Mask**: index = np.argmax(scores) finds the index of the highest score in the scores array. This identifies the most confident and likely correct mask among the predicted set.
* **Collecting Results**: result\_masks.append(masks[index]) adds the best-scoring mask to the result\_masks list. After the loop finishes, this list contains the single best segmentation mask for each input bounding box.
* **Returning the Output**: return np.array(result\_masks) converts the list of selected masks into a single NumPy array and returns it. This array represents the final output, with each element corresponding to the segmentation mask for an input bounding box.

## class TabletopHandyBotNode(Node):

class TabletopHandyBotNode(Node):

    """Main ROS 2 node for Tabletop HandyBot."""

TabletopHandyBotNode inherits from ROS 2's Node class and serves as the main controller for a vision-guided robotic manipulation system.

## \_\_init\_\_ method

### Parameters

    # TODO: make args rosparams

    def \_\_init\_\_(

            self,

            annotate: bool = False, # Enable visual debugging annotations

            publish\_point\_cloud: bool = False, # Publish point clouds for visualization

            assistant\_id: str = "", # OpenAI Assistant ID

            # Adjust these offsets to your needs:

            offset\_x: float = 0.015, # Gripper calibration offsets

            offset\_y: float = -0.015,

            offset\_z: float = 0.08,  # accounts for the height of the gripper

    ):

### ROS 2 Node Initialization

        super().\_\_init\_\_("tabletop\_handybot\_node")

        self.logger = self.get\_logger()

### Basic Component Setup

        self.cv\_bridge = CvBridge() *# Converts between ROS and OpenCV images*

        self.gripper\_joint\_name = "gripper\_joint"

        callback\_group = ReentrantCallbackGroup()

**CvBridge**: Converts between ROS Image messages and OpenCV arrays

**ReentrantCallbackGroup**: Allows callbacks to execute concurrently (important for multi-threaded operations)

### Robot Arm Configuration

        # Create MoveIt 2 interface

        self.arm\_joint\_names = [

            "joint\_1", "joint\_2", "joint\_3", "joint\_4", "joint\_5", "joint\_6"

        ]

        self.moveit2 = MoveIt2(

            node=self,

            joint\_names=self.arm\_joint\_names,

            base\_link\_name="base\_link", # Robot's reference frame

            end\_effector\_name="link\_6", # Last link of arm

            group\_name="ar\_manipulator", # MoveIt planning group

            callback\_group=callback\_group,

        )

        self.moveit2.planner\_id = "RRTConnectkConfigDefault" *# Motion planning algorithm*

**MoveIt2** is the motion planning framework that:

* Plans collision-free paths
* Handles inverse kinematics
* Executes smooth trajectories

### Gripper Interface Setup

        self.gripper\_interface = GripperInterface(

            node=self,

            gripper\_joint\_names=["gripper\_jaw1\_joint"],

            open\_gripper\_joint\_positions=[-0.012], # Fully open position

            closed\_gripper\_joint\_positions=[0.0], # Fully closed position

            gripper\_group\_name="ar\_gripper",

            callback\_group=callback\_group,

            gripper\_command\_action\_name="/gripper\_controller/gripper\_cmd",

        )

This configures a parallel gripper where:

* Negative values = open gripper
* Zero = closed gripper
* Uses ROS action interface for control

Above we had: from pymoveit2 import GripperInterface

### Transform System

    self.tf\_buffer = tf2\_ros.Buffer()

    self.tf\_listener = tf2\_ros.TransformListener(self.tf\_buffer, self)

**TF (Transform) System** maintains coordinate relationships between:

* Camera frame → Robot base frame
* End effector → Base frame
* Essential for converting 2D image coordinates to 3D robot positions

### Computer Vision Models

        # GroundingDINO for object detection

self.grounding\_dino\_model = Model(

            model\_config\_path=GROUNDING\_DINO\_CONFIG\_PATH,

            model\_checkpoint\_path=GROUNDING\_DINO\_CHECKPOINT\_PATH,

        )

# Segment Anything Model (SAM) for segmentation

        self.sam = sam\_model\_registry[SAM\_ENCODER\_VERSION](

            checkpoint=SAM\_CHECKPOINT\_PATH)

        self.sam.to(device=DEVICE)

        self.sam\_predictor = SamPredictor(self.sam)

### OpenAI Integration

        self.openai = openai.OpenAI()

        self.assistant: Assistant = get\_or\_create\_assistant(

            self.openai, assistant\_id)

        self.logger.info(f"Loaded assistant with ID: {self.assistant.id}")

Sets up AI assistant that can:

* Process natural language commands
* Call robot functions as tools
* Manage multi-step task execution

### State Variables

        self.annotate = annotate

        self.publish\_point\_cloud = publish\_point\_cloud

        self.n\_frames\_processed = 0

        self.\_last\_depth\_msg = None *# Latest depth image*

        self.\_last\_rgb\_msg = None *# Latest RGB image*

        self.arm\_joint\_state: JointState | None = None *# Current joint positions*

        self.\_last\_detections: sv.Detections | None = None *# Latest object detections*

        self.\_object\_in\_gripper: bool = False *# Gripper state tracking*

        self.gripper\_squeeze\_factor = 0.5 *# Gripper force scaling*

**Offset storage** for gripper calibration:

        self.offset\_x = offset\_x

        self.offset\_y = offset\_y

        self.offset\_z = offset\_z

These compensate for mechanical differences between camera coordinates and actual grasp points.

### ROS Subscriptions & Optional publishers

#### ROS Sensor Data

        self.image\_sub = self.create\_subscription(Image,

                                                  "/camera/color/image\_raw",

                                                  self.image\_callback, 10)

        self.depth\_sub = self.create\_subscription(

            Image, "/camera/aligned\_depth\_to\_color/image\_raw",

            self.depth\_callback, 10)

        self.joint\_states\_sub = self.create\_subscription(

            JointState, "/joint\_states", self.joint\_states\_callback, 10)

In robotics, **joint states are indeed sensor data** - they represent the current physical state of the robot's joints as measured by internal sensors.

header *# Timestamp*

name[] *# Joint names ["joint\_1", "joint\_2", ...]*

position[] *# Current joint angles/positions (from encoders)*

velocity[] *# Current joint velocities (computed or measured)*

effort[] *# Current joint forces/torques (from force sensors)*

Types of Joint Sensors

* **Encoders** → Measure joint position/angle
* **Velocity sensors** → Measure rotational/linear speed
* **Force/torque sensors** → Measure mechanical loads
* **Current sensors** → Monitor motor current (indicates effort)

#### Optional Publishers

        if self.publish\_point\_cloud:

            self.point\_cloud\_pub = self.create\_publisher(

                PointCloud2, "/point\_cloud", 2)

#### ROS Command Interfaces

        self.prompt\_sub = self.create\_subscription(

            String, "/prompt", self.start, 10,

            callback\_group=MutuallyExclusiveCallbackGroup())

        self.save\_images\_sub = self.create\_subscription(

            String, "/save\_images", self.save\_images, 10)

# Direct control commands

        self.detect\_objects\_sub = self.create\_subscription(

            String, "/detect\_objects", self.detect\_objects\_cb, 10)

        self.release\_at\_sub = self.create\_subscription(Int64, "/release\_above",

                                                   self.release\_above\_cb, 10)

        self.pick\_object\_sub = self.create\_subscription(

            Int64, "/pick\_object", self.pick\_object\_cb, 10)

        self.logger.info("Tabletop HandyBot node initialized.")

**Key Design Decision**: The /prompt subscription uses MutuallyExclusiveCallbackGroup() because natural language processing can take significant time, and we don't want it to block other operations.

*Q: About the imporantance of “self.create\_subscription( String, "/prompt", self.start, 10”*

**What it does:**

* Creates a ROS subscriber that listens to the /prompt topic
* Any message published to /prompt will trigger the self.start method
* Queue size of 10 messages (if messages come faster than processing)

**Why it's critical:**

* This is the **primary interface** for commanding the robot
* Without this, there's no way to send natural language commands to the system
* It's the entry point that converts ROS messages into robot actions

## start method

### Code

    def start(self, msg: String):

        if not self.\_last\_rgb\_msg or not self.\_last\_depth\_msg:

            return

        rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

        depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

        self.\_last\_detections = None

        self.logger.info(f"Processing: {msg.data}")

        self.logger.info(

            f"Initial Joint states: {self.arm\_joint\_state.position}")

        thread = self.openai.beta.threads.create()

        message = self.openai.beta.threads.messages.create(

            thread\_id=thread.id,

            role="user",

            content=msg.data,

        )

        self.logger.info(str(message))

        run = self.openai.beta.threads.runs.create\_and\_poll(

            thread\_id=thread.id,

            assistant\_id=self.assistant.id,

        )

        done = False

        while not done:

            if run.status == "completed":

                messages = self.openai.beta.threads.messages.list(

                    thread\_id=thread.id)

                self.logger.info(str(messages))

                done = True

                break

            else:

                self.logger.info(run.status)

            tool\_outputs = []

            for tool\_call in run.required\_action.submit\_tool\_outputs.tool\_calls:

                self.logger.info(f"tool\_call: {tool\_call}")

                if tool\_call.type == "function":

                    self.handle\_tool\_call(tool\_call, rgb\_image, depth\_image,

                                          tool\_outputs)

            self.logger.info(f"tool\_outputs: {tool\_outputs}")

            if tool\_outputs:

                try:

                    run = self.openai.beta.threads.runs.submit\_tool\_outputs\_and\_poll(

                        thread\_id=thread.id,

                        run\_id=run.id,

                        tool\_outputs=tool\_outputs)

                    self.logger.info("Tool outputs submitted successfully.")

                except Exception as e:  # pylint: disable=broad-except

                    self.logger.error(f"Failed to submit tool outputs: {e}")

            else:

                self.logger.info("No tool outputs to submit.")

        self.go\_home()

        self.logger.info("Task completed.")

### Section 1: Initial Setup and Validation

if not self.\_last\_rgb\_msg or not self.\_last\_depth\_msg:

return

rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

self.\_last\_detections = None

**Breakdown:**

1. **Sensor Data Check**: Ensures both RGB and depth images are available before proceeding
2. **Image Conversion**: Converts ROS Image messages to OpenCV numpy arrays using CvBridge
3. **Detection Reset**: Clears any previous object detections to start fresh

*Q: What is the point of line "depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)"? Why do we need this conversion?*

The conversion on that line is necessary because of the different data formats used by ROS and OpenCV/computer vision libraries.

**The Format Difference**

**ROS Format (sensor\_msgs/Image):**

* This is ROS's standardized message format for images
* Contains metadata like encoding, timestamp, frame\_id
* Data is packaged in a ROS-specific structure
* Designed for inter-process communication in ROS systems

**OpenCV Format (np.ndarray):**

* This is a NumPy array that OpenCV and most computer vision libraries expect
* Raw pixel data in memory
* No ROS-specific metadata
* Direct access to pixel values for processing

**Why The Conversion Is Needed**

Looking at how depth\_image is used in the code:

1. **In handle\_tool\_call**: Passed to pick\_object() and release\_above() methods
2. **In those methods**: Used for 3D point cloud generation and spatial reasoning
3. **Computer Vision Processing**: Libraries like Open3D expect NumPy arrays, not ROS messages

For example, in pick\_object():

masked\_depth\_image[mask] = depth\_image[mask] *# Array indexing*

masked\_depth\_image /= 1000.0 *# Mathematical operations*

pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

o3d.geometry.Image(masked\_depth\_image), *# Open3D expects numpy array*

...

)

**What CvBridge Does**

CvBridge handles:

* **Format Translation**: Converts ROS image encoding to OpenCV format
* **Memory Layout**: Ensures proper array structure
* **Data Type Conversion**: Handles different pixel formats (uint16 for depth, uint8 for RGB, etc.)
* **Error Handling**: Manages encoding mismatches

### Only one image conversion?

*Q: But why is this conversation in start method? Does this mean this only runs once and only does one convertion? I would think that we would need to do this multiple times.*

You're absolutely right to question this! You've identified a potential issue with the current design.

**The Current Design Problem**

Looking at the start method, the conversion happens **once** at the beginning:

def start(self, msg: String):

*# ... validation ...*

rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

*# ... rest of method uses these same images ...*

This means:

* The images are captured at the **start** of the command
* The same static images are used throughout the **entire** execution sequence
* Even if the robot moves objects around, it's still using the original images

**Why This Could Be Problematic**

Consider this scenario:

1. User says: "Pick up the red cup and put it next to the blue plate"
2. start() captures images showing cup in original position
3. Assistant calls detect\_objects("cup,plate") - uses original images ✓
4. Assistant calls pick\_object(0) - robot picks up cup, cup moves
5. Assistant calls detect\_objects("plate") - **still uses original images!** ❌
6. Assistant calls move\_above\_object\_and\_release(0) - uses outdated plate location ❌

**Why It Might Still Work**

The current design might work because:

1. **Static Scene Assumption**: The background objects (like the plate) don't move
2. **Single Detection Phase**: The assistant might detect all objects upfront, then just reference them by index
3. **Simple Tasks**: For basic pick-and-place, you might not need updated perception

**Better Design Would Be**

You're right that ideally, the conversions should happen **per tool call**:

def handle\_tool\_call(self, tool\_call, tool\_outputs):

*# Convert fresh images for each tool call*

rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

if tool\_call.function.name == "detect\_objects":

*# Use fresh images*

self.\_last\_detections = self.detect\_objects(rgb\_image, classes)

This would ensure each tool call uses the most recent sensor data, allowing the robot to adapt to changes in the scene caused by its own actions.

So you've correctly identified that the current design takes a "snapshot" approach rather than a "live perception" approach, which could be limiting for more complex manipulation tasks.

### Will start run again?

*Q: But if the user says a second command, will "start" method run again?*

Yes, absolutely! The start method will run again for each new command. Let me explain how this works:

**How Commands Trigger the start Method**

Looking at the ROS subscription setup in the \_\_init\_\_ method:

self.prompt\_sub = self.create\_subscription(

String, "/prompt", self.start, 10,

callback\_group=MutuallyExclusiveCallbackGroup())

This means:

* **Every time** a message is published to the /prompt topic, the start method gets called
* Each call to start is a **separate execution** with fresh sensor data
* The MutuallyExclusiveCallbackGroup ensures commands don't run simultaneously

**Command Execution Flow**

**First Command:**

1. User/system publishes "Pick up the red cup" to /prompt topic
2. start() method runs:
   * Captures current RGB/depth images
   * Creates new OpenAI thread
   * Executes the task
   * Robot goes home

**Second Command:**

1. User/system publishes "Put the cup on the plate" to /prompt topic
2. start() method runs **again**:
   * Captures **new** RGB/depth images (showing cup in gripper from previous command)
   * Creates **another new** OpenAI thread
   * Executes this new task
   * Robot goes home again

**Key Points**

**Fresh Data Per Command:**

* Each start() call gets the latest self.\_last\_rgb\_msg and self.\_last\_depth\_msg
* These are continuously updated by the camera callbacks
* So each command sees the current state of the world

### Section 2: Logging and OpenAI Thread Creation

self.logger.info(f"Processing: {msg.data}")

self.logger.info(f"Initial Joint states: {self.arm\_joint\_state.position}")

thread = self.openai.beta.threads.create()

message = self.openai.beta.threads.messages.create(

thread\_id=thread.id,

role="user",

content=msg.data,

)

self.logger.info(str(message))

**Process:**

1. **Task Logging**: Records the incoming command for debugging
2. **State Logging**: Captures initial robot arm joint positions
3. **Thread Creation**: Creates a new OpenAI conversation thread (isolated context)
4. **Message Creation**: Adds the user's natural language command to the thread
5. **Message Logging**: Records the created message object

### Section 3: Assistant Run Initialization

run = self.openai.beta.threads.runs.create\_and\_poll(

thread\_id=thread.id,

assistant\_id=self.assistant.id,

)

**Function:**

* **Run Creation**: Starts the OpenAI Assistant processing the thread
* **Polling**: create\_and\_poll automatically handles the asynchronous nature of the API
* **Assistant Binding**: Uses the pre-configured assistant with robot manipulation tools

### Section 4: Main Loop

#### 4.1 - Loop start & Status check

done = False

while not done:

if run.status == "completed":

messages = self.openai.beta.threads.messages.list(

thread\_id=thread.id)

self.logger.info(str(messages))

done = True

break

else:

self.logger.info(run.status)

**Loop Logic:**

1. **Status Check**: Continuously monitors the assistant's processing status
2. **Completion Handler**: When finished, retrieves all conversation messages
3. **Status Logging**: Reports intermediate statuses (e.g., "requires\_action", "in\_progress")
4. **Loop Exit**: Sets done flag and breaks when assistant completes

#### 4.2 - Tool Call Processing

tool\_outputs = []

for tool\_call in run.required\_action.submit\_tool\_outputs.tool\_calls:

self.logger.info(f"tool\_call: {tool\_call}")

if tool\_call.type == "function":

self.handle\_tool\_call(tool\_call, rgb\_image, depth\_image, tool\_outputs)

**Process:**

1. **Output Initialization**: Creates empty list for tool responses
2. **Tool Call Iteration**: Loops through each function call requested by the assistant
3. **Tool Call Logging**: Records each tool call for debugging
4. **Function Execution**: Delegates to handle\_tool\_call method for actual robot operations
5. **Response Accumulation**: handle\_tool\_call appends responses to tool\_outputs

#### 4.3 - Tool Output Submission

self.logger.info(f"tool\_outputs: {tool\_outputs}")

if tool\_outputs:

try:

run = self.openai.beta.threads.runs.submit\_tool\_outputs\_and\_poll(

thread\_id=thread.id,

run\_id=run.id,

tool\_outputs=tool\_outputs)

self.logger.info("Tool outputs submitted successfully.")

except Exception as e: *# pylint: disable=broad-except*

self.logger.error(f"Failed to submit tool outputs: {e}")

else:

self.logger.info("No tool outputs to submit.")

**Error Handling and Submission:**

1. **Output Logging**: Records all tool responses before submission
2. **Conditional Submission**: Only submits if there are actual outputs
3. **API Call**: Sends tool results back to OpenAI and continues polling
4. **Exception Handling**: Catches and logs any API communication errors
5. **Success Confirmation**: Logs successful submission
6. **Empty Case**: Handles scenario where no tools were called

### Section 5: Task Completion

self.go\_home()

self.logger.info("Task completed.")

**Cleanup:**

1. **Robot Reset**: Returns robot arm to home position
2. **Completion Logging**: Records task completion

## execution\_success method

### Code

def execution\_success(self, tool\_call: RequiredActionFunctionToolCall,

                          tool\_outputs: List[dict]) -> None:

        output\_dict = {"success": True}

        tool\_outputs.append({

            "tool\_call\_id": tool\_call.id,

            "output": json.dumps(output\_dict),

        })

### Explanation

This code is part of the **OpenAI Assistant API integration** for handling tool execution results. Let me break it down:

**Purpose**

This method reports successful completion of a robot function call back to the OpenAI Assistant.

**Parameters**

* **tool\_call**: A RequiredActionFunctionToolCall object from OpenAI's API representing a function the AI assistant wants to execute
* **tool\_outputs**: A list that accumulates responses to send back to the assistant

**How It Works**

**1. Creates Success Response**

output\_dict = {"success": True}

Creates a simple dictionary indicating the operation completed successfully.

**2. Formats for OpenAI API**

tool\_outputs.append({

"tool\_call\_id": tool\_call.id, *# Links response to specific tool call*

"output": json.dumps(output\_dict), *# Serializes response as JSON string*

})

The OpenAI Assistant API requires responses in this specific format:

* **tool\_call\_id**: Matches the ID of the original function call
* **output**: JSON string containing the result

**Why This Pattern Exists**

1. **AI Feedback Loop**: The assistant needs to know if its commands succeeded or failed
2. **Error Handling**: Failed operations can trigger alternative strategies
3. **Task Continuation**: Success signals allow the AI to proceed with next steps
4. **Debugging**: Provides clear status reporting for complex multi-step tasks

### Explanation 2

*Q:But does this project deal with multi-step tasks? I thought commands were just a single-step task (at least from OpenAI assistant point of view)*

A: Let me clarify what's actually happening in this system.

**Single vs Multi-Step Tasks**

From the **OpenAI Assistant's perspective**, each individual tool call is indeed a single step. However, the system handles **multi-step tasks** in two ways:

**1. AI-Orchestrated Multi-Step Tasks**

The OpenAI Assistant can break down complex commands into multiple sequential tool calls within a single conversation:

# User input: "Pick up the red cup and place it on the blue plate"

# AI Assistant might generate this sequence:

# Step 1: detect\_objects("red cup, blue plate")

# Step 2: pick\_object(0) # red cup index

# Step 3: move\_above\_object\_and\_release(1) # blue plate index

Looking at the conversation loop in start():

while not done:

if run.status == "completed":

done = True

break

*# Handle current batch of tool calls*

for tool\_call in run.required\_action.submit\_tool\_outputs.tool\_calls:

self.handle\_tool\_call(tool\_call, ...)

*# Submit responses and get next batch of tool calls*

run = self.openai.beta.threads.runs.submit\_tool\_outputs\_and\_poll(...)

*# ↑ This loop continues until the AI decides the task is complete*

**2. Robot-Level Multi-Step Operations**

Individual tool calls themselves involve multiple physical steps:

def pick\_object(self, object\_index, ...):

*# Multiple physical steps within one "tool call":*

self.gripper\_interface.open() *# Step 1: Open gripper*

msg.position.z += 0.05 *# Step 2: Move above object*

self.move\_to(msg)

msg.position.z -= 0.05 *# Step 3: Descend to object*

self.move\_to(msg)

self.gripper\_interface.move\_to\_position(...) *# Step 4: Close gripper*

msg.position.z += 0.12 *# Step 5: Lift object*

self.move\_to(msg)

*# But from AI's perspective, this is just one successful "pick\_object" call*

self.execution\_success(tool\_call, tool\_outputs)

**Why the Success/Failure Reporting Matters**

Even for "single-step" tool calls, the feedback is crucial because:

**1. Conditional Logic**

The AI needs to know if a step failed to adjust its strategy:

*# If detection fails, AI might try different object descriptions*

*# If picking fails, AI might try a different approach*

*# If placement fails, AI might try a different location*

**2. State Management**

The AI tracks what has been accomplished:

if tool\_call.function.name == "pick\_object":

if self.\_object\_in\_gripper: *# State check*

self.execution\_failure(tool\_call, tool\_outputs,

"Gripper is already holding an object.")

*# AI now knows it needs to release first*

**3. Error Recovery**

Failed operations can trigger alternative strategies within the same conversation.

**So You're Right, But...**

You're correct that individual tool calls are single steps from the AI's perspective. However:

* **The conversation itself is multi-step** (AI can make multiple tool calls)
* **Each tool call can fail**, requiring the AI to adapt
* **The success/failure reporting enables intelligent task orchestration**

The execution\_success/failure methods aren't about multi-step tasks per se, but about giving the AI reliable feedback to make intelligent decisions about what to do next within a potentially complex task sequence.

### Orchestration

*Q: Which system is doing this orchestration?*

The **orchestration** is being done by the **OpenAI Assistant** (GPT model), not the robot code. Let me clarify the division of responsibilities:

**OpenAI Assistant = The Orchestrator**

The OpenAI Assistant (running on OpenAI's servers) is responsible for:

1. **Breaking down complex commands** into sequential tool calls
2. **Deciding what to do next** based on success/failure feedback
3. **Managing the overall task strategy**
4. **Adapting when things go wrong**

**How the Orchestration Works**

**1. Assistant Configuration**

The OpenAI Assistant is configured with available tools:

# In openai\_assistant.py (referenced but not shown in the code)

tools = [

{

"type": "function",

"function": {

"name": "detect\_objects",

"description": "Detect objects in the scene",

"parameters": {...}

}

},

{

"type": "function",

"function": {

"name": "pick\_object",

"description": "Pick up an object by index",

"parameters": {...}

}

},

# ... more tools

]

**2. AI Makes Decisions**

When you say "Pick up the red cup and put it on the blue plate", the **GPT model** decides:

* "First I need to detect objects"
* "Then I need to pick the red cup"
* "Then I need to place it above the blue plate"

**3. Robot Code is Just the Executor**

The robot code (TabletopHandyBotNode) is purely reactive:

def start(self, msg: String):

*# Send user command to OpenAI*

thread = self.openai.beta.threads.create()

message = self.openai.beta.threads.messages.create(

thread\_id=thread.id,

role="user",

content=msg.data, *# "Pick up red cup and put it on blue plate"*

)

*# Let OpenAI Assistant decide what to do*

run = self.openai.beta.threads.runs.create\_and\_poll(

thread\_id=thread.id,

assistant\_id=self.assistant.id,

)

*# Robot just executes whatever the AI decides*

while not done:

for tool\_call in run.required\_action.submit\_tool\_outputs.tool\_calls:

*# AI decided to call these functions - robot just executes*

self.handle\_tool\_call(tool\_call, ...)

**The Conversation Flow**

Here's what actually happens:

User: "Pick up the red cup and put it on the blue plate"

↓

OpenAI Assistant: "I need to detect objects first"

↓ (calls tool)

Robot: detect\_objects("red cup, blue plate") → success

↓ (reports back)

OpenAI Assistant: "Great! I see a red cup at index 0 and blue plate at index 1. Now I'll pick up the cup"

↓ (calls tool)

Robot: pick\_object(0) → success

↓ (reports back)

OpenAI Assistant: "Perfect! Now I'll place it above the blue plate"

↓ (calls tool)

Robot: move\_above\_object\_and\_release(1) → success

↓ (reports back)

OpenAI Assistant: "Task completed successfully!"

**Robot's Role = Dumb Executor**

The robot code has **no orchestration logic**. It just:

1. **Waits** for tool calls from the AI
2. **Executes** the requested function
3. **Reports** success or failure back
4. **Waits** for the next instruction

**Why This Architecture?**

**Advantages:**

* **AI handles complex reasoning** (what to do when things fail, alternative strategies)
* **Robot code stays simple** (just implements basic actions)
* **Easy to extend** (add new capabilities by adding tools)
* **Natural language interface** (AI translates human intent to robot actions)

**The orchestration intelligence lives entirely in the OpenAI Assistant**, making this a classic example of **AI-driven robotics** where the language model serves as the "brain" and the robot serves as the "body."

### Agent?

*Q: So this Open AI Assistent is like an Agent? How does it work internally? Is something like LangChain used?*

A: Yes, the **OpenAI Assistant is indeed an agent** - but it's quite different from LangChain-based agents. Let me explain how it works:

The OpenAI Assistant exhibits classic **agentic behavior**:

* **Planning**: Breaks down complex tasks into steps
* **Tool Usage**: Calls functions to interact with the world
* **Reasoning**: Makes decisions based on feedback
* **Memory**: Maintains conversation context across multiple interactions

**How OpenAI Assistants Work Internally**

**1. Native OpenAI Implementation (Not LangChain)**

The Assistants API enables developers to easily build powerful AI assistants within their apps. This API removes the need to manage conversation history and adds access to OpenAI-hosted tools like Code Interpreter and File Search. [Assistants API (v2) FAQ | OpenAI Help Center](https://help.openai.com/en/articles/8550641-assistants-api-v2-faq)

OpenAI built their own agentic framework rather than using LangChain:

# This is OpenAI's native assistant, not LangChain

self.assistant: Assistant = get\_or\_create\_assistant(self.openai, assistant\_id)

**2. Key Components**

**Assistant Definition:**

* **Instructions**: System prompt defining the assistant's role
* **Tools**: Available functions it can call
* **Model**: Which GPT model powers the reasoning

**Thread Management:**

* **Threads**: Conversation contexts (like chat sessions)
* **Messages**: User inputs and assistant responses
* **Runs**: Execution instances that process messages

**3. Internal Architecture**

User Message → Thread → Run → Model Reasoning → Tool Calls → Execution → Response

**The Run Loop:**

1. **Message Processing**: GPT model reads conversation history
2. **Planning**: Model decides what tools to call and in what order
3. **Tool Calling**: Generates structured function calls
4. **Execution Wait**: Pauses while external system executes tools
5. **Feedback Integration**: Receives tool results and continues reasoning
6. **Response Generation**: Produces final answer or next tool calls

**4. Comparison with LangChain Agents**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **OpenAI Assistants** | **LangChain Agents** |
| **Hosting** | OpenAI's servers | Your infrastructure |
| **State Management** | Automatic | Manual implementation |
| **Tool Definition** | JSON schema | Python functions + decorators |
| **Memory** | Built-in thread persistence | Custom memory classes |
| **Reasoning** | Native GPT integration | ReAct/Plan-and-Execute patterns |

**5. How the Robot Code Uses It**

Looking at the code structure:

*# Create conversation thread*

thread = self.openai.beta.threads.create()

*# Send user message*

message = self.openai.beta.threads.messages.create(

thread\_id=thread.id,

role="user",

content="Pick up the red cup and put it on the blue plate"

)

*# Let assistant reason and plan*

run = self.openai.beta.threads.runs.create\_and\_poll(

thread\_id=thread.id,

assistant\_id=self.assistant.id

)

*# Handle tool calls in a loop*

while not done:

for tool\_call in run.required\_action.submit\_tool\_outputs.tool\_calls:

*# Assistant decided to call these tools*

self.handle\_tool\_call(tool\_call, ...)

*# Send results back and continue reasoning*

run = self.openai.beta.threads.runs.submit\_tool\_outputs\_and\_poll(...)

**6. Internal Agent Loop**

The assistant internally follows this reasoning pattern:

1. Analyze user request: "Pick up red cup, place on blue plate"

2. Plan: Need to detect objects first

3. Call: detect\_objects("red cup, blue plate")

4. Receive: {success: true, objects: ["red cup at index 0", "blue plate at index 1"]}

5. Plan: Now pick up the red cup

6. Call: pick\_object(0)

7. Receive: {success: true}

8. Plan: Now place above blue plate

9. Call: move\_above\_object\_and\_release(1)

10. Receive: {success: true}

11. Conclude: Task completed successfully

**Why Not LangChain?**

**OpenAI Assistants Advantages:**

* **Zero setup**: No need to implement agent loops, memory, or state management
* **Reliable**: Battle-tested on OpenAI's infrastructure
* **Integrated**: Native tool calling with GPT models
* **Persistent**: Conversations automatically saved

**Potential Downsides:**

* **Vendor lock-in**: Tied to OpenAI's ecosystem
* **Less control**: Can't customize the reasoning loop
* **Cost**: Pay per API call vs. running your own models

**So Yes, It's an Agent!**

The OpenAI Assistant is essentially a **hosted, managed agent service** that provides the same capabilities as LangChain agents but with less code and infrastructure management. It's doing sophisticated multi-step reasoning, planning, and tool orchestration - all the hallmarks of an AI agent - but packaged as a ready-to-use API service.

### Which tools to use?

*Q: How does the AI Assistant know which tools it should use at any point?*

The OpenAI Assistant uses several mechanisms to decide which tools to use:

**1. Tool Descriptions and Context**

The assistant is configured with detailed tool descriptions that guide its decision-making:

# Example tool definition (from the assistant configuration)

{

"type": "function",

"function": {

"name": "detect\_objects",

"description": "Detect and identify objects in the current camera view. Use this before any manipulation tasks to see what objects are available.",

"parameters": {

"type": "object",

"properties": {

"object\_classes": {

"type": "string",

"description": "Comma-separated list of object types to look for (e.g., 'cup, plate, bottle')"

}

}

}

}

},

{

"type": "function",

"function": {

"name": "pick\_object",

"description": "Pick up a detected object using its index from the detection results. Only use after detect\_objects has been called.",

"parameters": {

"type": "object",

"properties": {

"object\_index": {

"type": "integer",

"description": "Index of the object to pick up (0-based, from detection results)"

}

}

}

}

}

**2. System Instructions (Agent Prompt)**

The assistant has system-level instructions that define its role and strategy:

You are a robotic manipulation assistant. Your goal is to help users manipulate objects on a tabletop using a robotic arm.

IMPORTANT RULES:

- Always call detect\_objects FIRST before any manipulation

- Only pick objects that have been detected

- Check if the gripper is already holding something before picking

- Plan your actions step by step

- If a tool call fails, try alternative approaches

Available tools:

- detect\_objects: See what objects are present

- pick\_object: Grasp a detected object

- move\_above\_object\_and\_release: Drop held object above another object

- release\_gripper: Open gripper immediately

- flick\_wrist\_while\_release: Dynamic release with wrist motion

Example workflow for "put A on B":

1. detect\_objects("A, B")

2. pick\_object(index\_of\_A)

3. move\_above\_object\_and\_release(index\_of\_B)

**3. Contextual Reasoning Process**

The GPT model uses its reasoning capabilities to analyze:

**Current State Understanding:**

* What has been accomplished so far?
* What information is missing?
* What's the next logical step?

**Example Internal Reasoning:**

User says: "Pick up the red cup and place it on the blue plate"

GPT thinks:

- I need to manipulate objects, but I don't know what's in the scene

- The description says "Always call detect\_objects FIRST"

- I should look for "red cup" and "blue plate"

- → Call detect\_objects("red cup, blue plate")

After detection succeeds:

- Now I know red cup is at index 0, blue plate at index 1

- I need to pick the red cup first

- The pick\_object description says "Only use after detect\_objects"

- → Call pick\_object(0)

After pick succeeds:

- Now I'm holding the red cup

- I need to place it on the blue plate

- move\_above\_object\_and\_release seems right for this

- → Call move\_above\_object\_and\_release(1)

**4. Dependency Chain Recognition**

The AI understands tool dependencies from descriptions and results:

# In the robot code - this creates the dependency feedback

def handle\_tool\_call(self, tool\_call, ...):

if tool\_call.function.name == "pick\_object":

if self.\_last\_detections is None:

self.execution\_failure(tool\_call, tool\_outputs,

"No detections available.") # ← Forces detect\_objects first

return

The AI learns: "If pick\_object fails with 'No detections available', I must call detect\_objects first"

**5. State Tracking Through Conversation**

The assistant maintains conversation context to track what's been done:

Assistant: I'll help you pick up the red cup and place it on the blue plate.

First, let me see what objects are available.

→ Calls detect\_objects("red cup, blue plate")

Great! I can see a red cup at position 0 and a blue plate at position 1.

→ Calls pick\_object(0)

Perfect! I've picked up the red cup. Now I'll place it on the blue plate.

→ Calls move\_above\_object\_and\_release(1)

Task completed successfully!

**6. Error-Driven Adaptation**

When tools fail, the AI adapts its strategy:

*# If this happens:*

{

"success": false,

"failure\_reason": "Gripper is already holding an object."

}

*# AI might reason:*

*# "I can't pick up the cup because something is already in the gripper.*

*# I should release it first, then try again."*

*# → Calls release\_gripper()*

*# → Then retries pick\_object(0)*

**7. Multi-Modal Context Integration**

The AI can also consider:

* **Previous tool outputs** (what objects were detected)
* **Success/failure patterns** (what worked before)
* **State information** (is gripper full?)

**The Decision Algorithm**

Essentially, the AI uses this decision process:

1. **Parse user intent** ("pick up red cup")
2. **Check prerequisites** (do I know what objects exist?)
3. **Select appropriate tool** (detect\_objects if no info, pick\_object if ready)
4. **Execute and get feedback**
5. **Update mental model** and repeat

**Why This Works**

This approach leverages the **reasoning capabilities of large language models**:

* **Pattern recognition** from training on code and instructions
* **Causal understanding** of tool dependencies
* **Error analysis** and recovery strategies
* **Multi-step planning** based on descriptions and context

The key insight is that GPT models are already good at following instructions and reasoning about sequential processes - the OpenAI Assistants API just provides the infrastructure to turn that reasoning into **executable actions**.

## execution\_failure method

### Code

def execution\_failure(self, tool\_call: RequiredActionFunctionToolCall,

tool\_outputs: List[dict], failure\_reason: str):

output\_dict = {

"success": False,

"failure\_reason": failure\_reason,

}

tool\_outputs.append({

"tool\_call\_id": tool\_call.id,

"output": json.dumps(output\_dict),

})

### Explanation

The execution\_failure method in the provided TabletopHandyBotNode class is used to handle cases where a tool call (from an OpenAI Assistant API interaction) fails. Here's a breakdown of what the method does:

* **Purpose**: This method constructs a failure response for a tool call and appends it to the tool\_outputs list, which is likely used to communicate the result back to the OpenAI Assistant API.
* **Context**: It is called within the handle\_tool\_call method when a tool call (e.g., pick\_object, move\_above\_object\_and\_release) cannot be executed successfully.

**Parameters**

1. **tool\_call: RequiredActionFunctionToolCall**
   * This is an object representing a tool call request from the OpenAI Assistant API. It contains details such as the tool's ID and the function to be called.
   * The tool\_call.id is used to identify the specific tool call in the response.
2. **tool\_outputs: List[dict]**
   * A list of dictionaries that stores the outputs of tool calls. This method appends the failure response to this list.
3. **failure\_reason: str**
   * A string describing why the tool call failed (e.g., "No detections available" or "Gripper is already holding an object").

**Functionality**

1. **Create Failure Response**:
   * Constructs a dictionary (output\_dict) with two keys:
     + "success": False - Indicates the tool call was unsuccessful.
     + "failure\_reason": failure\_reason - Provides the reason for the failure, as passed in the failure\_reason parameter.
2. **Serialize and Append Output**:
   * The output\_dict is converted to a JSON string using json.dumps.
   * A new dictionary is created with:
     + "tool\_call\_id": The ID of the tool call, extracted from tool\_call.id.
     + "output": The JSON-serialized output\_dict.
   * This dictionary is appended to the tool\_outputs list.

**Example Usage**

This method is used in the handle\_tool\_call method to handle error cases. For example:

if self.\_last\_detections is None:

self.execution\_failure(tool\_call, tool\_outputs, "No detections available.")

return

In this case, if no object detections are available, the method is called to log the failure and append the failure response to tool\_outputs.

**Example Output**

If the method is called with a tool\_call object with ID "call\_123" and failure\_reason = "No detections available", the tool\_outputs list will have a new entry like:

{

"tool\_call\_id": "call\_123",

"output": '{"success": false, "failure\_reason": "No detections available"}'

}

**Notes**

* **Error Handling**: The method is straightforward and does not include additional error handling, assuming the inputs are valid.
* **Integration with OpenAI**: The structure of the output matches the expected format for submitting tool call results to the OpenAI Assistant API (via submit\_tool\_outputs\_and\_poll).
* **Thread Safety**: Since the node uses a MultiThreadedExecutor, the method should be safe for concurrent execution, but care must be taken to ensure tool\_outputs is not modified concurrently elsewhere.

*Q: Can you tell me more about "RequiredActionFunctionToolCall" ?*

The RequiredActionFunctionToolCall is a type from the OpenAI Python SDK, specifically used in the context of the OpenAI Assistant API. It represents a request from an Assistant's run to execute a specific function as part of an interaction.

## handle\_tool\_call method method

### Code

    def handle\_tool\_call(self, tool\_call: RequiredActionFunctionToolCall,

                         rgb\_image: np.ndarray, depth\_image: np.ndarray,

                         tool\_outputs: List[dict]):

        if tool\_call.function.name == "detect\_objects":

            args = json.loads(tool\_call.function.arguments)

            classes\_str = args["object\_classes"]

            classes = classes\_str.split(",")

            self.\_last\_detections = self.detect\_objects(rgb\_image, classes)

            detected\_classes = [

                classes[class\_id]

                for class\_id in self.\_last\_detections.class\_id

            ]

            self.logger.info(f"Detected {detected\_classes}.")

            self.logger.info(

                f"detection confidence: {self.\_last\_detections.confidence}")

            output\_dict = {

                "object\_classes": detected\_classes,

            }

            tool\_outputs.append({

                "tool\_call\_id": tool\_call.id,

                "output": json.dumps(output\_dict),

            })

        elif tool\_call.function.name == "pick\_object":

            args = json.loads(tool\_call.function.arguments)

            if self.\_last\_detections is None:

                self.execution\_failure(tool\_call, tool\_outputs,

                                       "No detections available.")

                return

            if self.\_object\_in\_gripper:

                self.execution\_failure(

                    tool\_call, tool\_outputs,

                    "Gripper is already holding an object.")

                return

            if args["object\_index"] >= len(self.\_last\_detections.mask):

                self.execution\_failure(

                    tool\_call, tool\_outputs,

                    ("Invalid object index. Only "

                     f"{len(self.\_last\_detections.mask)} objects detected."))

                return

            self.pick\_object(args["object\_index"], self.\_last\_detections,

                             depth\_image)

            self.logger.info(

                f"done picking object. Joint states: {self.arm\_joint\_state.position}"

            )

            self.\_object\_in\_gripper = True

            self.execution\_success(tool\_call, tool\_outputs)

        elif tool\_call.function.name == "move\_above\_object\_and\_release":

            args = json.loads(tool\_call.function.arguments)

            if self.\_last\_detections is None:

                self.execution\_failure(tool\_call, tool\_outputs,

                                       "No detections available.")

                return

            if args["object\_index"] >= len(self.\_last\_detections.mask):

                self.execution\_failure(

                    tool\_call, tool\_outputs,

                    ("Invalid object index. Only "

                     f"{len(self.\_last\_detections.mask)} objects detected."))

                return

            self.release\_above(args["object\_index"], self.\_last\_detections,

                               depth\_image)

            self.\_object\_in\_gripper = False

            self.execution\_success(tool\_call, tool\_outputs)

        elif tool\_call.function.name == "release\_gripper":

            self.release\_gripper()

            self.execution\_success(tool\_call, tool\_outputs)

            self.\_object\_in\_gripper = False

        elif tool\_call.function.name == "flick\_wrist\_while\_release":

            self.flick\_wrist\_while\_release()

            self.execution\_success(tool\_call, tool\_outputs)

            self.\_object\_in\_gripper = False

The handle\_tool\_call method is essentially a sophisticated command dispatcher that translates OpenAI Assistant function calls into robotic actions

### Tool Handler 1: detect\_objects

if tool\_call.function.name == "detect\_objects":

args = json.loads(tool\_call.function.arguments)

classes\_str = args["object\_classes"]

classes = classes\_str.split(",")

self.\_last\_detections = self.detect\_objects(rgb\_image, classes)

detected\_classes = [

classes[class\_id]

for class\_id in self.\_last\_detections.class\_id

]

self.logger.info(f"Detected {detected\_classes}.")

self.logger.info(f"detection confidence: {self.\_last\_detections.confidence}")

output\_dict = {

"object\_classes": detected\_classes,

}

tool\_outputs.append({

"tool\_call\_id": tool\_call.id,

"output": json.dumps(output\_dict),

})

**Step-by-step breakdown:**

1. **Parse Arguments**: Extracts JSON arguments and gets comma-separated object class string
2. **Split Classes**: Converts "apple,banana,cup" → ["apple", "banana", "cup"]
3. **Run Detection**: Calls self.detect\_objects() with RGB image and class list
4. **Storage**: Saves detections in self.\_last\_detections for subsequent operations
5. **Map Results**: Converts class IDs back to class names using the original classes list
6. **Logging**: Records detected objects and their confidence scores
7. **Response Formation**: Creates JSON response with detected object classes

### Tool Handler 2: pick\_object

elif tool\_call.function.name == "pick\_object":

args = json.loads(tool\_call.function.arguments)

if self.\_last\_detections is None:

self.execution\_failure(tool\_call, tool\_outputs,

"No detections available.")

return

if self.\_object\_in\_gripper:

self.execution\_failure(tool\_call, tool\_outputs,

"Gripper is already holding an object.")

return

if args["object\_index"] >= len(self.\_last\_detections.mask):

self.execution\_failure(tool\_call, tool\_outputs,

("Invalid object index. Only "

f"{len(self.\_last\_detections.mask)} objects detected."))

return

self.pick\_object(args["object\_index"], self.\_last\_detections, depth\_image)

self.logger.info(

f"done picking object. Joint states: {self.arm\_joint\_state.position}"

)

self.\_object\_in\_gripper = True

self.execution\_success(tool\_call, tool\_outputs)

**Validation Chain:**

1. **Detection Check**: Ensures previous object detection was performed
2. **Gripper State Check**: Prevents picking when gripper already holds something
3. **Index Validation**: Ensures requested object index exists in detection results
4. **Execution**: If all checks pass, performs the pick operation
5. **State Update**: Sets gripper flag to indicate object is held
6. **Success Response**: Reports successful completion

### Tool Handler 3: move\_above\_object\_and\_release

elif tool\_call.function.name == "move\_above\_object\_and\_release":

args = json.loads(tool\_call.function.arguments)

if self.\_last\_detections is None:

self.execution\_failure(tool\_call, tool\_outputs,

"No detections available.")

return

if args["object\_index"] >= len(self.\_last\_detections.mask):

self.execution\_failure(

tool\_call, tool\_outputs,

("Invalid object index. Only "

f"{len(self.\_last\_detections.mask)} objects detected."))

return

self.release\_above(args["object\_index"], self.\_last\_detections,

depth\_image)

self.\_object\_in\_gripper = False

self.execution\_success(tool\_call, tool\_outputs)

**Logic Flow:**

1. **Same Validation**: Checks for detections and valid object index
2. **Release Operation**: Moves above target object and releases gripper
3. **State Reset**: Clears the gripper holding flag
4. **Success Response**: Confirms completion

### Tool Handler 4: release\_gripper

elif tool\_call.function.name == "release\_gripper":

self.release\_gripper()

self.execution\_success(tool\_call, tool\_outputs)

self.\_object\_in\_gripper = False

**Simple Operation:**

1. **Direct Release**: Just opens gripper without moving
2. **State Reset**: Clears gripper flag
3. **Success Response**: Reports completion

### Tool Handler 5: flick\_wrist\_while\_release

elif tool\_call.function.name == "flick\_wrist\_while\_release":

self.flick\_wrist\_while\_release()

self.execution\_success(tool\_call, tool\_outputs)

self.\_object\_in\_gripper = False

**Dynamic Release:**

1. **Flick Motion**: Performs wrist rotation while releasing
2. **State Reset**: Clears gripper flag
3. **Success Response**: Reports completion

## detect\_objects method

### Code

    def detect\_objects(self, image: np.ndarray, object\_classes: List[str]):

        self.logger.info(f"Detecting objects of classes: {object\_classes}")

        detections: sv.Detections = self.grounding\_dino\_model.predict\_with\_classes(

            image=image,

            classes=object\_classes,

            box\_threshold=BOX\_THRESHOLD,

            text\_threshold=TEXT\_THRESHOLD,

        )

        # NMS post process

        nms\_idx = (torchvision.ops.nms(

            torch.from\_numpy(detections.xyxy),

            torch.from\_numpy(detections.confidence),

            NMS\_THRESHOLD,

        ).numpy().tolist())

        detections.xyxy = detections.xyxy[nms\_idx]

        detections.confidence = detections.confidence[nms\_idx]

        detections.class\_id = detections.class\_id[nms\_idx]

        detections.mask = segment(

            sam\_predictor=self.sam\_predictor,

            image=cv2.cvtColor(image, cv2.COLOR\_BGR2RGB),

            xyxy=detections.xyxy,

        )

        if self.annotate:

            box\_annotator = sv.BoxAnnotator()

            mask\_annotator = sv.MaskAnnotator()

            labels = [

                f"{object\_classes[class\_id]} {confidence:0.2f}"

                for \_, \_, confidence, class\_id, \_, \_ in detections

            ]

            annotated\_frame = box\_annotator.annotate(scene=image.copy(),

                                                     detections=detections,

                                                     labels=labels)

            cv2.imwrite(

                f"annotated\_image\_detections\_{self.n\_frames\_processed}.jpg",

                annotated\_frame)

            annotated\_frame = mask\_annotator.annotate(scene=image.copy(),

                                                      detections=detections)

            cv2.imwrite(f"annotated\_image\_masks\_{self.n\_frames\_processed}.jpg",

                        annotated\_frame)

        if self.publish\_point\_cloud:

            depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

            # mask out the depth image except for the detected objects

            masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

            for mask in detections.mask:

                masked\_depth\_image[mask] = depth\_image[mask]

            masked\_depth\_image /= 1000.0

            # convert the masked depth image to a point cloud

            pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

                o3d.geometry.Image(masked\_depth\_image),

                o3d.camera.PinholeCameraIntrinsic(

                    o3d.camera.PinholeCameraIntrinsicParameters.

                    PrimeSenseDefault),

            )

            # convert it to a ROS PointCloud2 message

            points = np.asarray(pcd.points)

            pc\_msg = point\_cloud\_to\_msg(points, "/camera\_color\_frame")

            self.point\_cloud\_pub.publish(pc\_msg)

        self.n\_frames\_processed += 1

        return detections

### Overview

The detect\_objects method is essentially a state-of-the-art computer vision pipeline that can understand natural language object descriptions and find those objects in camera images.

What makes this particularly powerful is the combination of GroundingDINO (which can detect objects from text descriptions without being specifically trained on them) and SAM (which provides pixel-perfect masks). This means you can ask the robot to find "a red ceramic mug" or "the book with a blue cover" and it will likely succeed, even if it has never seen those exact objects before.

### Section 1: Initial Logging and Object Detection

self.logger.info(f"Detecting objects of classes: {object\_classes}")

detections: sv.Detections = self.grounding\_dino\_model.predict\_with\_classes(

image=image,

classes=object\_classes,

box\_threshold=BOX\_THRESHOLD,

text\_threshold=TEXT\_THRESHOLD,

)

**Process:**

1. **Logging**: Records which object classes are being searched for
2. **GroundingDINO Detection**: Uses the pre-loaded GroundingDINO model for object detection
3. **Text-to-Vision**: GroundingDINO can detect objects based on natural language descriptions
4. **Threshold Filtering**: Only detections above confidence thresholds are kept
   * BOX\_THRESHOLD = 0.4: Minimum confidence for bounding box detection
   * TEXT\_THRESHOLD = 0.25: Minimum confidence for text-image matching

**Key Insight**: GroundingDINO is special because it can detect objects from text descriptions without pre-training on specific classes.

### Section 2: Non-Maximum Suppression (NMS) Post-Processing

# NMS post process

nms\_idx = (torchvision.ops.nms(

torch.from\_numpy(detections.xyxy),

torch.from\_numpy(detections.confidence),

NMS\_THRESHOLD,

).numpy().tolist())

detections.xyxy = detections.xyxy[nms\_idx]

detections.confidence = detections.confidence[nms\_idx]

detections.class\_id = detections.class\_id[nms\_idx]

**What NMS Does:**

1. **Problem**: Multiple bounding boxes might detect the same object
2. **Solution**: NMS removes overlapping detections, keeping only the best one
3. **Process**:
   * Convert NumPy arrays to PyTorch tensors for torchvision.ops.nms
   * Apply NMS with threshold NMS\_THRESHOLD = 0.8
   * Get indices of boxes to keep
   * Filter all detection arrays to keep only selected boxes

**Why It's Important:**

* Prevents duplicate detections of the same object
* Improves detection quality by removing redundant boxes
* Essential for accurate object counting

NMS stands for **Non-Maximum Suppression**, and it's a crucial post-processing step in object detection. Let me explain what it does and why it's needed.

**What is Non-Maximum Suppression (NMS)?**

NMS is an algorithm that eliminates duplicate detections of the same object by removing overlapping bounding boxes and keeping only the best one.

**The Problem NMS Solves**

**Without NMS, you get this:**

Original Detection Results:

Box 1: "cup" confidence=0.9, coordinates=(100,100,200,200)

Box 2: "cup" confidence=0.8, coordinates=(105,105,205,205)

Box 3: "cup" confidence=0.7, coordinates=(95,95,195,195)

Box 4: "apple" confidence=0.85, coordinates=(300,150,400,250)

All three "cup" boxes are detecting the **same physical cup** - they're just slightly different positions with different confidence scores.

**After NMS, you get this:**

Cleaned Detection Results:

Box 1: "cup" confidence=0.9, coordinates=(100,100,200,200) [KEPT - highest confidence]

Box 4: "apple" confidence=0.85, coordinates=(300,150,400,250) [KEPT - different object]

The overlapping cup boxes are removed, keeping only the best one.

**How NMS Works (Step by Step)**

1. **Sort by Confidence**: Order all detections from highest to lowest confidence
2. **Pick the Best**: Take the detection with highest confidence
3. **Calculate Overlap**: For each remaining detection, calculate how much it overlaps with the chosen one
4. **Remove Duplicates**: If overlap > threshold (0.8 in this code), remove the lower-confidence detection
5. **Repeat**: Continue with remaining detections until all are processed

**The NMS Threshold (0.8)**

NMS\_THRESHOLD = 0.8

This controls how much overlap is allowed before considering boxes as duplicates:

* **Higher threshold (0.9)**: Boxes must overlap a lot to be considered duplicates
  + **Result**: Keeps more boxes, might keep some duplicates
  + **Use case**: When you want to be conservative about removing detections
* **Lower threshold (0.5)**: Boxes only need moderate overlap to be considered duplicates
  + **Result**: Removes more boxes, might remove valid detections
  + **Use case**: When you want aggressive duplicate removal
* **Current value (0.8)**: Balanced approach - removes clear duplicates but keeps distinct objects

### Section 3: Segmentation with SAM (Segment Anything Model)

detections.mask = segment(

sam\_predictor=self.sam\_predictor,

image=cv2.cvtColor(image, cv2.COLOR\_BGR2RGB),

xyxy=detections.xyxy,

)

**Process:**

1. **Color Conversion**: OpenCV uses BGR, SAM expects RGB
2. **Segmentation Call**: Uses the standalone segment() function
3. **Input**: Bounding boxes from GroundingDINO detection
4. **Output**: Pixel-precise masks for each detected object

**Integration Pattern**: This combines two state-of-the-art models:

* **GroundingDINO**: Finds where objects are (bounding boxes)
* **SAM**: Determines exactly which pixels belong to each object (masks)

### Section 4: Optional Visualization and Annotation

if self.annotate:

box\_annotator = sv.BoxAnnotator()

mask\_annotator = sv.MaskAnnotator()

labels = [

f"{object\_classes[class\_id]} {confidence:0.2f}"

for \_, \_, confidence, class\_id, \_, \_ in detections

]

annotated\_frame = box\_annotator.annotate(scene=image.copy(),

detections=detections, labels=labels)

cv2.imwrite(

f"annotated\_image\_detections\_{self.n\_frames\_processed}.jpg",

annotated\_frame)

annotated\_frame = mask\_annotator.annotate(scene=image.copy(),

detections=detections)

cv2.imwrite(f"annotated\_image\_masks\_{self.n\_frames\_processed}.jpg",

annotated\_frame)

**Visualization Features:**

1. **Label Creation**: Combines class names with confidence scores
2. **Box Annotation**: Draws bounding boxes with labels on image
3. **Mask Annotation**: Overlays segmentation masks on image
4. **File Output**: Saves annotated images with frame counter
5. **Debugging Aid**: Helps verify detection quality visually

**When Used**: Only when self.annotate = True (constructor parameter)

### Section 5: Optional Point Cloud Generation

if self.publish\_point\_cloud:

depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

*# mask out the depth image except for the detected objects*

masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

for mask in detections.mask:

masked\_depth\_image[mask] = depth\_image[mask]

masked\_depth\_image /= 1000.0

**3D Processing:**

1. **Depth Integration**: Uses stored depth image from camera
2. **Mask Application**: Only keeps depth pixels where objects were detected
3. **Background Removal**: Sets non-object pixels to zero
4. **Unit Conversion**: Converts from millimeters to meters (÷ 1000)

### 5.1: Point Cloud Creation and Publishing

# convert the masked depth image to a point cloud

pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

o3d.geometry.Image(masked\_depth\_image),

o3d.camera.PinholeCameraIntrinsic(

o3d.camera.PinholeCameraIntrinsicParameters.

PrimeSenseDefault),

)

# convert it to a ROS PointCloud2 message

points = np.asarray(pcd.points)

pc\_msg = point\_cloud\_to\_msg(points, "/camera\_color\_frame")

self.point\_cloud\_pub.publish(pc\_msg)

**3D Reconstruction:**

1. **Point Cloud Generation**: Uses Open3D to convert depth image to 3D points
2. **Camera Model**: Uses PrimeSense default intrinsic parameters
3. **Coordinate Conversion**: Transforms from image coordinates to 3D world coordinates
4. **ROS Integration**: Converts to ROS PointCloud2 message format
5. **Publishing**: Sends point cloud to ROS topic for visualization/processing

*Q: We have "if self.publish\_point\_cloud:" regarding optional point cloud generation. Why would we want this to be optional. Don't we always need this?*

Great observation! You'd think point clouds would always be needed for 3D manipulation, but making it optional is actually a smart design choice. Let me explain why:

**Why Point Cloud Publishing is Optional**

**1. Different Use Cases**

**When you DO need point clouds:**

* Debugging and visualization in RViz
* Research and development
* Teaching/demonstrations
* Integration with other ROS nodes that process point clouds
* Analysis of 3D scene understanding

**When you DON'T need point clouds:**

* Production deployment where only manipulation matters
* Headless operation without visualization
* Bandwidth-limited systems
* When no other nodes consume the point cloud data

### Section 6: Frame Counter and Return

self.n\_frames\_processed += 1

return detections

**Cleanup:**

1. **Frame Tracking**: Increments counter for file naming and debugging
2. **Return Results**: Provides detections object for further processing

### Integration with Robot System

**Upstream Integration:**

* Called by handle\_tool\_call() when AI requests object detection
* Uses latest camera images from ROS subscriptions

**Downstream Integration:**

* Results stored in self.\_last\_detections for pick/place operations
* Masks used by pick\_object() and release\_above() for 3D positioning
* Point clouds can be visualized in RViz for debugging

## pick\_object

### Code

    def pick\_object(self, object\_index: int, detections: sv.Detections,

                    depth\_image: np.ndarray):

        """Perform a top-down grasp on the object."""

        # mask out the depth image except for the detected objects

        masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

        mask = detections.mask[object\_index]

        masked\_depth\_image[mask] = depth\_image[mask]

        masked\_depth\_image /= 1000.0

        # convert the masked depth image to a point cloud

        pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

            o3d.geometry.Image(masked\_depth\_image),

            o3d.camera.PinholeCameraIntrinsic(

                o3d.camera.PinholeCameraIntrinsicParameters.PrimeSenseDefault),

        )

        pcd.transform(self.cam\_to\_base\_affine)

        points = np.asarray(pcd.points)

        grasp\_z = points[:, 2].max()

        near\_grasp\_z\_points = points[points[:, 2] > grasp\_z - 0.008]

        xy\_points = near\_grasp\_z\_points[:, :2]

        xy\_points = xy\_points.astype(np.float32)

        center, dimensions, theta = cv2.minAreaRect(xy\_points)

        gripper\_rotation = theta

        if dimensions[0] > dimensions[1]:

            gripper\_rotation -= 90

        if gripper\_rotation < -90:

            gripper\_rotation += 180

        elif gripper\_rotation > 90:

            gripper\_rotation -= 180

        gripper\_opening = min(dimensions)

        grasp\_pose = Pose()

        grasp\_pose.position.x = center[0] + self.offset\_x

        grasp\_pose.position.y = center[1] + self.offset\_y

        grasp\_pose.position.z = grasp\_z + self.offset\_z

        top\_down\_rot = Rotation.from\_quat([0, 1, 0, 0])

        extra\_rot = Rotation.from\_euler("z", gripper\_rotation, degrees=True)

        grasp\_quat = (extra\_rot \* top\_down\_rot).as\_quat()

        grasp\_pose.orientation.x = grasp\_quat[0]

        grasp\_pose.orientation.y = grasp\_quat[1]

        grasp\_pose.orientation.z = grasp\_quat[2]

        grasp\_pose.orientation.w = grasp\_quat[3]

        self.grasp\_at(grasp\_pose, gripper\_opening)

### Section 1: Depth Image Masking and Preprocessing

# mask out the depth image except for the detected objects

masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

mask = detections.mask[object\_index]

masked\_depth\_image[mask] = depth\_image[mask]

masked\_depth\_image /= 1000.0

**Process:**

1. **Create Empty Depth Image**: Zeros out everything initially
2. **Extract Object Mask**: Gets the pixel-precise mask for the target object
3. **Apply Mask**: Only keeps depth values where the object pixels are located
4. **Unit Conversion**: Converts from millimeters (camera default) to meters (robot units)

**Result:** A depth image containing only the target object, with background set to zero.

**Why This Matters:**

* Isolates the target object from cluttered scenes
* Prevents interference from background objects
* Ensures grasp planning focuses only on the intended target

### Section 2: 2D to 3D Point Cloud Conversion

# convert the masked depth image to a point cloud

pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

o3d.geometry.Image(masked\_depth\_image),

o3d.camera.PinholeCameraIntrinsic(

o3d.camera.PinholeCameraIntrinsicParameters.PrimeSenseDefault),

)

pcd.transform(self.cam\_to\_base\_affine)

points = np.asarray(pcd.points)

**Step-by-step:**

1. **Open3D Integration**: Wraps NumPy array as Open3D Image object
2. **Camera Model**: Uses PrimeSense default intrinsic parameters (focal length, principal point)
3. **3D Reconstruction**: Converts 2D depth pixels to 3D world coordinates - create\_from\_depth\_image & pcd.transform .
4. **Coordinate Transformation**: Transforms from camera frame to robot base frame
5. **Data Extraction**: Converts back to NumPy array for processing

**Coordinate Systems:**

* **Camera Frame**: Origin at camera, Z pointing forward
* **Robot Base Frame**: Origin at robot base, coordinates relative to robot arm
* **Transform**: self.cam\_to\_base\_affine handles rotation and translation between frames

#### o3d.geometry.PointCloud.create\_from\_depth\_image

**What It Does - It turns a flat depth image into a 3D point cloud.**

Think of it like this:

**Input: A Depth Image**

This is like a black and white photo where:

- Dark pixels = things that are FAR away

- Bright pixels = things that are CLOSE to the camera

Example depth image:

[0.5m, 0.6m, 0.7m] <- Row of pixels with distance values

[0.5m, 0.6m, 0.8m]

[0.5m, 0.7m, 0.9m]

**Output: A 3D Point Cloud**

A list of 3D coordinates showing where each pixel actually is in space:

Point 1: (x=0.1, y=0.2, z=0.5) <- This pixel is at this 3D location

Point 2: (x=0.2, y=0.2, z=0.6) <- This pixel is at this 3D location

Point 3: (x=0.3, y=0.2, z=0.7) <- This pixel is at this 3D location

... (thousands more points)

**Real-World Analogy**

Imagine you're looking at a photograph of a room:

* **2D Photo**: You see a table, but you don't know how far away it is
* **Depth Image**: Same photo, but now each pixel tells you the distance
* **Point Cloud**: Now you have a 3D model where you know exactly where each piece of the table is in 3D space

**The Magic**

The function basically asks: *"If this pixel in the image represents something 0.5 meters away from the camera, where exactly is that thing located in the real 3D world?"*

It uses **camera math** (the intrinsic parameters) to figure out the 3D coordinates.

**Visual Example**

Camera View (2D): Point Cloud (3D):

┌─────────────┐

│ 📱 🍎 │ → 📱 (x=0.2, y=0.1, z=0.4)

│ 📚 │ 🍎 (x=0.4, y=0.1, z=0.5)

│ │ 📚 (x=0.3, y=0.3, z=0.6)

└─────────────┘

**Why This Matters for the Robot**

* **Before**: Robot sees a flat image - "I see a cup somewhere in the picture"
* **After**: Robot knows 3D positions - "The cup is 0.4 meters away, 0.2 meters to the right, and 0.1 meters up"

Now the robot can actually reach out and grab the cup because it knows exactly where it is in 3D space!

**The Bottom Line**

**2D depth image + camera settings = 3D point cloud**

It's like converting a flat map into a 3D landscape model where you can see the actual heights and positions of everything.

*Q: But does the converstion happens immediately or we need to call "transform" method later?*

**The Conversion Process**

**Step 1: create\_from\_depth\_image (Immediate 3D conversion)**

pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

o3d.geometry.Image(masked\_depth\_image),

o3d.camera.PinholeCameraIntrinsic(...)

)

**This IMMEDIATELY creates 3D points, but they're in CAMERA coordinates:**

* Camera is at origin (0, 0, 0)
* Z-axis points away from camera (into the scene)
* X-axis points right from camera's perspective
* Y-axis points down from camera's perspective

**Result after this step:**

Point Cloud in CAMERA frame:

Point 1: (x=0.1, y=0.2, z=0.5) ← Relative to camera position

Point 2: (x=0.2, y=0.3, z=0.6) ← Relative to camera position

**Step 2: transform (Change coordinate system)**

pcd.transform(self.cam\_to\_base\_affine)

**This transforms the EXISTING 3D points to ROBOT coordinates:**

* Robot base is at origin (0, 0, 0)
* Coordinates are relative to the robot's base
* Now the robot arm can actually reach these positions

**Result after this step:**

Point Cloud in ROBOT BASE frame:

Point 1: (x=0.45, y=-0.12, z=0.73) ← Relative to robot base

Point 2: (x=0.46, y=-0.13, z=0.74) ← Relative to robot base

### Section 3: Grasp Height Determination

grasp\_z = points[:, 2].max()

**Logic:**

* Takes the **maximum Z-coordinate** (highest point) of the object
* Represents the top surface where the gripper should approach
* Simple but effective for top-down grasps

**Why Maximum vs Average:**

* **Avoids** grasping through the middle of tall objects
* **Ensures** gripper approaches from above the object
* **Works well** for typical tabletop objects (cups, boxes, fruits)

### Section 4: Grasp Point Selection and Object Analysis

near\_grasp\_z\_points = points[points[:, 2] > grasp\_z - 0.008]

xy\_points = near\_grasp\_z\_points[:, :2]

xy\_points = xy\_points.astype(np.float32)

center, dimensions, theta = cv2.minAreaRect(xy\_points)

**Point Filtering:**

1. **Height Filtering**: Only considers points within 8mm of the top surface
2. **2D Projection**: Projects filtered 3D points to XY plane (top-down view)
3. **Data Type**: Ensures float32 for OpenCV compatibility

**Minimum Area Rectangle:**

* **cv2.minAreaRect**: Finds smallest rectangle that contains all top-surface points
* **center**: (x, y) coordinates of rectangle center → **grasp location**
* **dimensions**: (width, height) of rectangle → **object shape info**
* **theta**: Rotation angle of rectangle → **object orientation**

**Why This Approach:**

* **Robust**: Works for irregular object shapes
* **Centered**: Finds geometric center for stable grasping
* **Orientation-aware**: Accounts for object rotation

### Section 5: Gripper Orientation Calculation

gripper\_rotation = theta

if dimensions[0] > dimensions[1]:

gripper\_rotation -= 90

if gripper\_rotation < -90:

gripper\_rotation += 180

elif gripper\_rotation > 90:

gripper\_rotation -= 180

**Orientation Logic:**

1. **Start with object angle**: Initial rotation from minAreaRect
2. **Align with longest dimension**: If width > height, rotate gripper 90°
3. **Angle normalization**: Keep rotation within [-90°, 90°] range

**Why This Strategy:**

* **Efficient grasping**: Aligns gripper with object's longest dimension
* **Stable grip**: Provides maximum contact area
* **Mechanical limits**: Respects gripper's rotation constraints

**Example:**

Object: Rectangular book (width=20cm, height=5cm, theta=30°)

→ dimensions[0] > dimensions[1] → gripper\_rotation = 30° - 90° = -60°

→ Gripper aligns with book's length for better grip

### Section 6: Gripper Opening Calculation

gripper\_opening = min(dimensions)

**Simple but Effective:**

* Uses the **smaller dimension** of the object's bounding rectangle
* Ensures gripper opens wide enough to encompass the object
* Prevents over-opening which could destabilize the grasp

**Limitation:** Doesn't account for object thickness, but works well for most tabletop objects.

### Section 7: 3D Pose Construction

grasp\_pose = Pose()

grasp\_pose.position.x = center[0] + self.offset\_x

grasp\_pose.position.y = center[1] + self.offset\_y

grasp\_pose.position.z = grasp\_z + self.offset\_z

**Position Calculation:**

1. **XY Position**: Uses computed center of object
2. **Z Position**: Uses highest point of object
3. **Calibration Offsets**: Adds hardware-specific corrections
   * offset\_x = 0.015: Compensates for gripper-to-camera displacement
   * offset\_y = -0.015: Accounts for mounting alignment
   * offset\_z = 0.08: Accounts for gripper geometry/height

### Section 8: Orientation Quaternion Construction

top\_down\_rot = Rotation.from\_quat([0, 1, 0, 0])

extra\_rot = Rotation.from\_euler("z", gripper\_rotation, degrees=True)

grasp\_quat = (extra\_rot \* top\_down\_rot).as\_quat()

grasp\_pose.orientation.x = grasp\_quat[0]

grasp\_pose.orientation.y = grasp\_quat[1]

grasp\_pose.orientation.z = grasp\_quat[2]

grasp\_pose.orientation.w = grasp\_quat[3]

**Rotation Composition:**

1. **Base Orientation**: [0, 1, 0, 0] represents gripper pointing straight down
2. **Additional Rotation**: Adds Z-axis rotation for object alignment
3. **Quaternion Math**: Multiplies rotations (order matters!)
4. **Pose Assignment**: Converts to ROS Pose message format

**Why Two-Step Rotation:**

* **Modular design**: Separates "pointing down" from "aligning with object"
* **Easy modification**: Can change base orientation without affecting alignment logic
* **Mathematical clarity**: Each rotation has a clear purpose

## grasp\_at

### Code

    def grasp\_at(self, msg: Pose, gripper\_opening: float):

        self.logger.info(f"Grasp at: {msg} with opening: {gripper\_opening}")

        self.gripper\_interface.open()

        self.gripper\_interface.wait\_until\_executed()

        # move 5cm above the item first

        msg.position.z += 0.05

        self.move\_to(msg)

        time.sleep(0.05)

        # grasp the item

        msg.position.z -= 0.05

        self.move\_to(msg)

        time.sleep(0.05)

        gripper\_pos = -gripper\_opening / 2. \* self.gripper\_squeeze\_factor

        gripper\_pos = min(gripper\_pos, 0.0)

        self.gripper\_interface.move\_to\_position(gripper\_pos)

        self.gripper\_interface.wait\_until\_executed()

        # lift the item

        msg.position.z += 0.12

        self.move\_to(msg)

        time.sleep(0.05)

### Section 1: Gripper Preparation

self.gripper\_interface.open()

self.gripper\_interface.wait\_until\_executed()

**Process:**

1. **Open Command**: Moves gripper to fully open position (open\_gripper\_joint\_positions=[-0.012])
2. **Blocking Wait**: Ensures gripper fully opens before proceeding
3. **Safety Measure**: Prevents collision with object during approach

**Why This First:**

* **Clearance**: Ensures gripper won't hit object during approach
* **Preparation**: Gets gripper ready for grasping motion
* **Predictable State**: Starts from known gripper configuration

### Section 2: Pre-Grasp Positioning

# move 5cm above the item first

msg.position.z += 0.05

self.move\_to(msg)

time.sleep(0.05)

**Motion Sequence:**

1. **Height Adjustment**: Adds 5cm (0.05m) to Z-coordinate
2. **Safe Approach**: Moves to position above the target object
3. **Settling Time**: 50ms pause for motion to stabilize

**Safety Strategy:**

* **Collision Avoidance**: Approaches from above to avoid side obstacles
* **Gradual Descent**: Two-stage approach instead of direct motion (it will move down in next step, doing the 2nd approach)
* **Motion Stabilization**: Time delay allows arm dynamics to settle

**Coordinate Modification:**

Original target: (x=0.3, y=0.2, z=0.15)

Approach pose: (x=0.3, y=0.2, z=0.20) *# 5cm higher*

**Why Two-Stage Instead of Direct?**

The real reasons are:

**1. Collision Avoidance**

* **Direct motion**: Robot might plan a path that goes through obstacles or other objects on the table
* **Two-stage approach**: Guarantees the final approach comes from directly above, avoiding side collisions

**2. Predictable Final Approach**

* **Direct motion**: MoveIt2 might choose any path to reach the target (could approach from side, angle, etc.)
* **Two-stage approach**: Forces the gripper to approach the object vertically from above, which is more reliable for top-down grasps

### Section 3: Final Approach

Move to final grasp position.

msg.position.z -= 0.05

self.move\_to(msg)

time.sleep(0.05)

**Process:**

1. **Height Restoration**: Subtracts 5cm to return to original target Z
2. **Final Positioning**: Moves to exact grasp location
3. **Stabilization**: Another 50ms pause before gripping

**Result:** Gripper is now at the precise location computed by pick\_object() method.

### Section 4: Gripper Closing

gripper\_pos = -gripper\_opening / 2. \* self.gripper\_squeeze\_factor

gripper\_pos = min(gripper\_pos, 0.0)

self.gripper\_interface.move\_to\_position(gripper\_pos)

self.gripper\_interface.wait\_until\_executed()

**Gripper Position Calculation:**

1. **Size Adaptation**: Uses gripper\_opening from object analysis
2. **Squeeze Factor**: Applies self.gripper\_squeeze\_factor = 0.5 (50% of computed opening)
3. **Safety Clamping**: min(gripper\_pos, 0.0) ensures position doesn't exceed limits
4. **Sign Convention**: Negative values typically mean "closed" in this gripper system

**Mathematical Example:**

# For an object requiring 6cm gripper opening:

gripper\_opening = 0.06 # 6cm from pick\_object()

gripper\_pos = -0.06 / 2 \* 0.5 # = -0.015 meters

gripper\_pos = min(-0.015, 0.0) # = -0.015 (within limits)

**Why Squeeze Factor:**

* **Force Control**: Prevents over-squeezing delicate objects
* **Grip Security**: Still provides enough force to hold objects
* **Hardware Protection**: Reduces stress on gripper motors

### Section 5: Lifting Motion – lift the item

# lift the item

msg.position.z += 0.12

self.move\_to(msg)

time.sleep(0.05)

**Lift Sequence:**

1. **Height Increase**: Adds 12cm (0.12m) to current Z position
2. **Vertical Motion**: Lifts object well above the surface
3. **Final Settling**: 50ms pause to stabilize with object in gripper

**Lift Height Strategy:**

* **12cm clearance**: Sufficient to clear most tabletop obstacles
* **Straight up motion**: Minimizes risk of collision during lift
* **Object security**: Maintains grasp while moving

### Key Motion Planning Insights

**Three-Stage Approach Strategy:**

Stage 1: Approach (Z + 5cm) → Safe positioning above object

Stage 2: Descend (Z - 5cm) → Precise grasp location

Stage 3: Lift (Z + 12cm) → Clear object from surface

**Blocking vs Non-Blocking Operations:**

* **move\_to()**: Blocking operation - waits until motion completes
* **time.sleep()**: Additional safety buffer for mechanical settling
* **wait\_until\_executed()**: Ensures gripper operations complete before proceeding

**Coordinate System Consistency:**

* All positions in robot base frame (transformed from camera coordinates)
* Z-axis modifications for vertical approach/lift strategy
* Original X,Y coordinates maintained throughout sequence

### Integration with Robot Hardware

**MoveIt2 Integration:**

def move\_to(self, msg: Pose):

pose\_goal = PoseStamped()

pose\_goal.header.frame\_id = "base\_link"

pose\_goal.pose = msg

self.moveit2.move\_to\_pose(pose=pose\_goal)

self.moveit2.wait\_until\_executed()

**Gripper Hardware Interface:**

* Uses GripperInterface for standardized gripper control
* Position-based control (not force-based)
* Blocking operations ensure sequence integrity

## release\_above

### Code

    def release\_above(self, object\_index: int, detections: sv.Detections,

                      depth\_image: np.ndarray):

        """Move the robot arm above the object and release the gripper."""

        masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

        mask = detections.mask[object\_index]

        masked\_depth\_image[mask] = depth\_image[mask]

        masked\_depth\_image /= 1000.0

        # convert the masked depth image to a point cloud

        pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

            o3d.geometry.Image(masked\_depth\_image),

            o3d.camera.PinholeCameraIntrinsic(

                o3d.camera.PinholeCameraIntrinsicParameters.PrimeSenseDefault),

        )

        pcd.transform(self.cam\_to\_base\_affine)

        points = np.asarray(pcd.points).astype(np.float32)

        # release 5cm above the object

        drop\_z = np.percentile(points[:, 2], 95) + 0.05

        median\_z = np.median(points[:, 2])

        xy\_points = points[points[:, 2] > median\_z, :2]

        xy\_points = xy\_points.astype(np.float32)

        center, \_, \_ = cv2.minAreaRect(xy\_points)

        drop\_pose = Pose()

        drop\_pose.position.x = center[0] + self.offset\_x

        drop\_pose.position.y = center[1] + self.offset\_y

        drop\_pose.position.z = drop\_z + self.offset\_z

        # Straight down pose

        drop\_pose.orientation.x = 0.0

        drop\_pose.orientation.y = 1.0

        drop\_pose.orientation.z = 0.0

        drop\_pose.orientation.w = 0.0

        self.release\_at(drop\_pose)

### Method Signature

def release\_above(self, object\_index: int, detections: sv.Detections, depth\_image: np.ndarray):

**Parameters:**

* object\_index: Index of the target object in the detections array (where to drop the held object)
* detections: Detection results containing object masks and locations
* depth\_image: Depth camera data as NumPy array (in millimeters)

**Purpose:** Move the robot arm above a specified detected object and release the gripper to drop the currently held object.

### Section 1: Depth Image Masking for Target Object

masked\_depth\_image = np.zeros\_like(depth\_image, dtype=np.float32)

mask = detections.mask[object\_index]

masked\_depth\_image[mask] = depth\_image[mask]

masked\_depth\_image /= 1000.0

**Process:**

1. **Create Empty Depth Image**: Initialize with zeros to mask out everything
2. **Extract Target Mask**: Get pixel-precise mask for the target drop location object
3. **Apply Mask**: Only preserve depth values where target object pixels are located
4. **Unit Conversion**: Convert from millimeters (camera default) to meters (robot units)

**Purpose:** Isolates the target object (e.g., a plate or container) where the held object should be dropped.

### Section 2: 3D Point Cloud Generation

# convert the masked depth image to a point cloud

pcd = o3d.geometry.PointCloud.create\_from\_depth\_image(

o3d.geometry.Image(masked\_depth\_image),

o3d.camera.PinholeCameraIntrinsic(

o3d.camera.PinholeCameraIntrinsicParameters.PrimeSenseDefault),

)

pcd.transform(self.cam\_to\_base\_affine)

**Process:**

1. **2D to 3D Conversion**: Convert masked depth image to 3D point cloud
2. **Camera Intrinsics**: Use PrimeSense default parameters for perspective projection
3. **Coordinate Transformation**: Transform from camera coordinates to robot base coordinates
4. **Result**: 3D point cloud representing only the target object's surface

### Section 3: Point Cloud Preprocessing with Drop Height Calculation

points = np.asarray(pcd.points).astype(np.float32)

*# release 5cm above the object*

drop\_z = np.percentile(points[:, 2], 95) + 0.05

median\_z = np.median(points[:, 2])

**Height Analysis:**

1. **Data Extraction**: Convert Open3D point cloud to NumPy array
2. **Drop Height Calculation**: Uses 95th percentile of Z-coordinates + 5cm clearance
3. **Median Height**: Calculate median Z for surface filtering

**Why 95th Percentile Instead of Maximum:**

* **Noise Robustness**: Avoids outlier points from sensor noise
* **Surface Estimation**: Represents the "top surface" while filtering noise spikes
* **Reliable Reference**: More stable than max() which could be a single noisy point

**Height Strategy:**

# Example point cloud Z-values: [0.15, 0.16, 0.15, 0.17, 0.16, 0.25, 0.16]

# max() = 0.25 (could be noise)

# 95th percentile = 0.17 (more reliable top surface)

# drop\_z = 0.17 + 0.05 = 0.22 (5cm above surface)

### Section 4: Target Location Calculation (“center” position)

xy\_points = points[points[:, 2] > median\_z, :2]

xy\_points = xy\_points.astype(np.float32)

center, \_, \_ = cv2.minAreaRect(xy\_points)

**Surface Point Filtering:**

1. **Height Filtering**: Only consider points above median height (upper surface points)
2. **2D Projection**: Extract only X,Y coordinates (ignore Z for center calculation)
3. **Data Type Conversion**: Ensure float32 for OpenCV compatibility

**Center Point Calculation:**

* **cv2.minAreaRect**: Finds minimum area rectangle containing all surface points
* **center**: Returns (x, y) coordinates of rectangle center
* **Ignores rotation**: Only uses center point, discards dimensions and theta

**Why Filter by Median Z (1st line):**

* **Surface Focus**: Only considers points on the upper surface of the object
* **Avoids Edge Effects**: Excludes points on the sides or bottom of the object
* **Stable Center**: Computes center based on actual surface area, not volume

### Section 5: Drop Pose Construction

drop\_pose = Pose()

drop\_pose.position.x = center[0] + self.offset\_x

drop\_pose.position.y = center[1] + self.offset\_y

drop\_pose.position.z = drop\_z + self.offset\_z

*# Straight down pose (orientation setting)*

drop\_pose.orientation.x = 0.0

drop\_pose.orientation.y = 1.0

drop\_pose.orientation.z = 0.0

drop\_pose.orientation.w = 0.0

**Position Calculation:**

1. **XY Position**: Uses computed center of target object's surface
2. **Hardware Offsets**: Applies calibration corrections for gripper-camera alignment
3. **Z Position**: Uses 95th percentile height + 5cm clearance + hardware offset

**Orientation (Quaternion):**

* **Fixed Orientation**: [0, 1, 0, 0] represents gripper pointing straight down
* **No Object Alignment**: Unlike grasping, dropping doesn't need to align with object orientation
* **Simple Strategy**: Always drops from directly above

### Section 6: Drop Execution

self.release\_at(drop\_pose)

**Delegation:**

* Passes computed drop pose to release\_at() method
* release\_at() handles the actual robot motion and gripper opening
* Separates **planning** (this method) from **execution** (release\_at())

### Potential Issues

* **No container awareness**: Doesn't optimize drop location for containers
* **Fixed clearance**: 5cm might be too much or too little for different objects
* **No trajectory planning**: Assumes straight-line approach is collision-free
* **No precision control**: Basic "drop and hope" strategy

## flick\_wrist\_while\_release

### Code

    def flick\_wrist\_while\_release(self):

        joint\_positions = self.arm\_joint\_state.position

        joint\_positions[4] -= np.deg2rad(25)

        self.moveit2.move\_to\_configuration(joint\_positions,

                                           self.arm\_joint\_names,

                                           tolerance=0.005)

        time.sleep(3)

        self.gripper\_interface.open()

        self.gripper\_interface.wait\_until\_executed()

        self.moveit2.wait\_until\_executed()

### Overview

The flick\_wrist\_while\_release method is particularly fascinating because it demonstrates how robots can mimic intuitive human behaviors to solve practical problems. When we have something sticky on our fingers, we instinctively give our hand a quick shake or flick to get it off - this method implements exactly that same strategy for a robotic gripper.

The engineering elegance lies in how simple the implementation is despite solving a complex physical problem. By rotating just one joint (the wrist) by a modest 25 degrees, the robot creates enough angular momentum to overcome adhesion forces that might keep an object stuck to the gripper fingers. This is much more effective than simply opening the gripper and hoping gravity will do the work.

The timing element is also crucial here. That three-second pause isn't arbitrary - it reflects an understanding of the physics involved. The robot needs time for the motion to fully develop momentum, for any oscillations to settle, and for the system to reach a stable state before releasing. This shows how robotics often requires balancing multiple physical constraints simultaneously.

**The Physics Behind the Motion**

This method implements what roboticists call a "dynamic release" - using momentum and quick motion to help separate objects that might be stuck to the gripper. Think of it like how you might flick your wrist when trying to get something sticky off your fingers.

### Step 1: Capture Current Robot Configuration

joint\_positions = self.arm\_joint\_state.position

**What's happening**: The robot takes a "snapshot" of where all its joints are currently positioned. This is like recording the exact pose of your arm before you make a motion.

**Why this matters**: The robot needs to know its starting position so it can calculate exactly how to move. Joint positions are typically stored as angles in radians for each of the six arm joints.

**Teaching moment**: Think of joint\_positions as a list of six numbers, where each number represents how much each joint has rotated from its zero position. For example: [0.1, -0.5, 0.8, 0.0, 0.2, -0.1].

### Step 2: Calculate the Wrist Rotation

joint\_positions[4] -= np.deg2rad(25)

**The key insight**: This line modifies only the 5th joint (index 4, since we start counting at 0). In most 6-DOF robot arms, this is the wrist rotation joint - the one that twists the gripper left and right.

**The math**: np.deg2rad(25) converts 25 degrees to radians (approximately 0.436 radians). The minus sign means we're rotating the wrist 25 degrees in the negative direction.

**Physical meaning**: Imagine holding your arm out straight and then quickly rotating your wrist counterclockwise - that's essentially what the robot is doing. This creates a sharp, sudden motion at the end effector.

### Step 3: Execute the Wrist Motion

self.moveit2.move\_to\_configuration(joint\_positions,

self.arm\_joint\_names,

tolerance=0.005)

**What's happening**: The robot moves to the new configuration where only the wrist has rotated 25 degrees. The tolerance=0.005 means the robot will consider the motion complete when it gets within 0.005 radians (about 0.3 degrees) of the target.

**Why it works**: This creates angular momentum at the gripper. Any object held in the gripper experiences a sudden rotational force that can help break adhesion or overcome friction.

### Step 4: The Critical Timing

time.sleep(3)

**The strategy**: This 3-second pause serves multiple purposes:

* It gives the wrist motion time to fully complete and settle
* It allows any object momentum to develop fully
* It ensures the robot is stable before opening the gripper

**Why timing matters**: If the gripper opened too quickly, the object might not have built up enough momentum. Too slow, and the momentum dissipates. Three seconds appears to be an empirically determined sweet spot.

### Step 5: Coordinated Release

self.gripper\_interface.open()

self.gripper\_interface.wait\_until\_executed()

self.moveit2.wait\_until\_executed()

**Simultaneous actions**: The robot opens the gripper while ensuring both the arm motion and gripper action complete properly.

**Safety consideration**: The wait\_until\_executed() calls ensure that each command finishes before the method returns, preventing any race conditions or incomplete actions.

## go\_home

### Code

   def go\_home(self):

        joint\_positions = [0., 0., 0., 0., 0., 0.]

        self.moveit2.move\_to\_configuration(joint\_positions,

                                           self.arm\_joint\_names,

                                           tolerance=0.005)

        self.moveit2.wait\_until\_executed()

### Explanation

The tolerance=0.005 parameter means the robot will consider itself "home" when each joint is within 0.005 radians (about 0.3 degrees) of zero. This is remarkably precise - imagine trying to point your finger to within a third of a degree!

**Ensuring Completion**

self.moveit2.wait\_until\_executed()

This final line embodies a crucial principle in robotics: never assume a command has completed just because you've issued it. The wait\_until\_executed() call ensures that the method doesn't return control to the calling program until the robot has actually reached the home position.

## cam\_to\_base\_affine

### Code

    @cached\_property

    def cam\_to\_base\_affine(self):

        cam\_to\_base\_link\_tf = self.tf\_buffer.lookup\_transform(

            target\_frame="base\_link",

            source\_frame="camera\_color\_frame",

            time=Time(),

            timeout=Duration(seconds=5))

        cam\_to\_base\_rot = Rotation.from\_quat([

            cam\_to\_base\_link\_tf.transform.rotation.x,

            cam\_to\_base\_link\_tf.transform.rotation.y,

            cam\_to\_base\_link\_tf.transform.rotation.z,

            cam\_to\_base\_link\_tf.transform.rotation.w,

        ])

        cam\_to\_base\_pos = np.array([

            cam\_to\_base\_link\_tf.transform.translation.x,

            cam\_to\_base\_link\_tf.transform.translation.y,

            cam\_to\_base\_link\_tf.transform.translation.z,

        ])

        affine = np.eye(4)

        affine[:3, :3] = cam\_to\_base\_rot.as\_matrix()

        affine[:3, 3] = cam\_to\_base\_pos

        return affine

### Overview

The cam\_to\_base\_affine method solves one of the fundamental challenges in robotics: how do you translate what a camera sees into coordinates that a robot arm can actually use?

This method creates what mathematicians call an affine transformation matrix, but thinking of it more simply, it's like creating a universal translator between two different ways of describing the same physical space. The camera sees the world in terms of its own perspective - distances and directions relative to where it's pointing. But the robot arm needs to understand everything in terms of its own coordinate system - distances and directions relative to its base.

The mathematical beauty lies in how this single 4×4 matrix can capture both the rotational relationship (how the camera is tilted or turned relative to the robot base) and the translational relationship (how far and in what direction the camera is positioned from the robot base). Once this transformation is computed, it can be applied to any point the camera detects, instantly converting those camera coordinates into robot coordinates.

What makes this method particularly sophisticated is the use of quaternions to handle the rotation component. Quaternions might seem mysterious, but they solve a fundamental problem in 3D mathematics: how do you represent complex rotations without running into mathematical singularities or ambiguities? They provide a robust, computation-friendly way to encode any possible 3D rotation in just four numbers.

The @cached\_property decorator adds another layer of engineering wisdom. Since the physical relationship between the camera and robot base doesn't change during operation, why recalculate this transformation matrix every time you need it? The decorator ensures this expensive calculation happens only once, then reuses the result for maximum efficiency.

### Step 1: The Transform Lookup: Asking ROS for Help

cam\_to\_base\_link\_tf = self.tf\_buffer.lookup\_transform(

target\_frame="base\_link",

source\_frame="camera\_color\_frame",

time=Time(),

timeout=Duration(seconds=5))

This is where ROS (Robot Operating System) shows its brilliance. ROS maintains what's called a "transform tree" - a constantly updated database of how every coordinate frame in the robot system relates to every other coordinate frame. Think of it like a GPS system that always knows the relationship between any two locations.

The lookup\_transform call is essentially asking ROS: "Given that a point is at some location in the camera's coordinate system, what would be the coordinates of that same point if I described it using the robot base's coordinate system?"

The five-second timeout ensures that if something goes wrong with the transform system, the robot doesn't wait forever.

### Step 2: Extracting the Rotation Component

cam\_to\_base\_rot = Rotation.from\_quat([

cam\_to\_base\_link\_tf.transform.rotation.x,

cam\_to\_base\_link\_tf.transform.rotation.y,

cam\_to\_base\_link\_tf.transform.rotation.z,

cam\_to\_base\_link\_tf.transform.rotation.w,

])

Rotation in 3D space is mathematically complex. If you've ever tried to describe how to orient a book in space by saying "turn it 30 degrees this way, then 45 degrees that way," you've discovered that the order of rotations matters enormously, and it gets confusing quickly.

Robotics uses quaternions to represent rotations because they avoid many of the mathematical problems that plague other rotation representations. Think of a quaternion as a mathematical package that contains all the information needed to describe any possible 3D rotation in a compact, computation-friendly form.

The four values (x, y, z, w) extracted here represent the quaternion that describes how the camera's coordinate axes are oriented relative to the robot base's coordinate axes. The **Rotation.from\_quat** function from scipy creates a rotation object that can perform the mathematical operations needed to transform vectors from one orientation to another.

**Extracting the Translation Component**

cam\_to\_base\_pos = np.array([

cam\_to\_base\_link\_tf.transform.translation.x,

cam\_to\_base\_link\_tf.transform.translation.y,

cam\_to\_base\_link\_tf.transform.translation.z,

])

This is the simpler part - it's just the 3D offset between the origin of the camera's coordinate system and the origin of the robot's coordinate system. If the camera is mounted 30 cm forward, 10 cm left, and 45 cm up from the robot base, these three numbers capture that offset.

### Step 3: Constructing the Affine Transformation Matrix

affine = np.eye(4) # Start with identity matrix

affine[:3, :3] = cam\_to\_base\_rot.as\_matrix() # Insert rotation

affine[:3, 3] = cam\_to\_base\_pos # Insert translation

return affine

Here's where the mathematical elegance really shines. An affine transformation matrix is a 4x4 matrix that can represent any combination of rotation, translation, and scaling in 3D space. It's like a mathematical Swiss Army knife for coordinate transformations.

The matrix starts as an identity matrix, which represents "no transformation at all." Then we systematically build in our specific transformation:

The upper-left 3x3 portion gets filled with the rotation matrix (converted from the quaternion). This part says "here's how to rotate vectors from the camera orientation to the robot base orientation."

The rightmost column's top three positions get the translation vector. This part says "after rotating, shift the result by these amounts in X, Y, and Z."

The bottom row stays as [0, 0, 0, 1], which maintains the mathematical properties needed for affine transformations.

## move\_to

### Code

    def move\_to(self, msg: Pose):

        pose\_goal = PoseStamped()

        pose\_goal.header.frame\_id = "base\_link"

        pose\_goal.pose = msg

        self.moveit2.move\_to\_pose(pose=pose\_goal)

        self.moveit2.wait\_until\_executed()

### Step 1: Creating the Pose Message - Adding Context to Intent

pose\_goal = PoseStamped()

pose\_goal.header.frame\_id = "base\_link"

pose\_goal.pose = msg

This first section demonstrates a crucial principle in robotics: context matters enormously. The input msg contains a Pose - a specification of where we want the robot's end effector (gripper) to be positioned and how we want it oriented. But a pose by itself is like saying "meet me at the corner of 5th and Main" without specifying which city you're talking about.

The PoseStamped message adds that crucial context. Think of it as upgrading from "meet me at the corner" to "meet me at the corner of 5th and Main Street in downtown Springfield, at 3 PM today." The header.frame\_id = "base\_link" part is particularly important because it tells the motion planner "interpret all these coordinates relative to the robot's base."

Why does this matter so much? In robotics, there might be dozens of coordinate frames active at once - the camera frame, the gripper frame, the table frame, various joint frames, and more. Without explicitly stating that we want the pose interpreted relative to "base\_link," the motion planner wouldn't know how to understand our request. It's like the difference between saying "drive 5 miles north" (north from where?) and "drive 5 miles north from the robot's current base position."

The PoseStamped message also includes timing information, though we don't see it explicitly set here. This timestamp helps the system understand whether this is a request for immediate motion or something to be executed later, and it helps with coordinate frame transformations that might change over time.

**PoseStamped()** comes from ROS:

In the ROS message system, a Pose contains seven numbers:

* Three numbers for position (x, y, z coordinates)
* Four numbers for orientation (a quaternion representing the rotation)

Now, here's where it gets interesting. A PoseStamped is a Pose with additional context information "stamped" onto it. The "stamp" includes:

* A timestamp (when was this pose measurement taken or when should it be achieved?)
* A coordinate frame reference (relative to what reference point should these coordinates be interpreted?)

**Understanding the Structure**

When you create a PoseStamped object in the code:

pythonpose\_goal = PoseStamped()

You're actually creating an object with this structure:

pythonpose\_goal.header.stamp # When this pose is relevant (timestamp)

pose\_goal.header.frame\_id # What coordinate system to use

pose\_goal.pose.position.x # X coordinate

pose\_goal.pose.position.y # Y coordinate

pose\_goal.pose.position.z # Z coordinate

pose\_goal.pose.orientation.x # Quaternion x component

pose\_goal.pose.orientation.y # Quaternion y component

pose\_goal.pose.orientation.z # Quaternion z component

pose\_goal.pose.orientation.w # Quaternion w component

The beauty of this system is that every robot in the world that uses ROS understands exactly what these fields mean and how to work with them.

### Step 2: Delegating to the Motion Planning Expert

self.moveit2.move\_to\_pose(pose=pose\_goal)

This single line of code is where the real magic happens, though it's mostly hidden from view. The moveit2 object represents MoveIt2. When we call move\_to\_pose, we're asking it to solve the problem for us.

Let me break down what MoveIt2 is actually doing behind the scenes:

**Forward and Inverse Kinematics**: First, MoveIt2 needs to figure out what joint angles will position the gripper at the desired location and orientation. This is called inverse kinematics, and it's like solving a complex trigonometry puzzle with six variables. For any given pose, there might be multiple solutions, no solutions, or solutions that require the robot to contort into impossible positions.

**Collision Detection**: Next, MoveIt2 builds a detailed 3D model of the robot and its environment, then checks whether any proposed motion would cause collisions. This includes self-collision (the robot hitting itself) and environmental collisions (hitting the table, walls, or other objects). Think of it as running a continuous physics simulation to ensure safety.

**Path Planning**: Once MoveIt2 knows the target joint configuration is achievable and safe, it needs to plan a path from the current position to the goal. This isn't simply moving in a straight line - it needs to find a path through the complex, high-dimensional space of possible joint configurations while avoiding obstacles. The algorithms used here are often based on sophisticated techniques like rapidly-exploring random trees (RRT) or probabilistic roadmaps.

**Trajectory Generation**: Finally, MoveIt2 converts the planned path into a smooth trajectory - a detailed specification of how each joint should move over time. This includes calculating velocities and accelerations that respect the physical limitations of the motors while ensuring smooth, coordinated motion.

What makes this particularly impressive is that MoveIt2 does all of this in real-time, often completing these calculations in a fraction of a second for typical robot motions.

### Step 3: Ensuring Completion - The Importance of Synchronization

self.moveit2.wait\_until\_executed()

The wait\_until\_executed() call serves several critical purposes:

**Synchronization**: It ensures that the calling method doesn't continue until the robot has actually reached its destination. This is crucial for sequential operations - you don't want to start trying to grasp an object until the robot has actually moved to the grasping position.

**Error Handling**: If something goes wrong during the motion (perhaps an unexpected obstacle appears, or a joint motor malfunctions), this method provides a place where such errors can be detected and handled appropriately.

**State Consistency**: By waiting for completion, we ensure that the robot's physical state matches what our software believes about its state. This consistency is crucial for reliable autonomous operation.

**Resource Management**: The method also ensures that the motion planning and control resources are properly cleaned up before the next operation begins.

Think of this like confirming that your GPS navigation has actually guided you to your destination before you start looking for the house number. Without this confirmation, you might start acting on the assumption that you've arrived when you're actually still in transit.

**The Hidden Complexity**

What makes the move\_to method so elegant is how it encapsulates enormous complexity behind a simple interface. From the perspective of other parts of the code (like pick\_object or grasp\_at), moving the robot is as simple as calling self.move\_to(pose). But underneath this simple call lies decades of robotics research and engineering.

The method represents a perfect example of abstraction in engineering - it provides a simple, reliable interface while hiding the complexity of motion planning, kinematics calculations, collision detection, and trajectory optimization. This allows the higher-level manipulation behaviors to focus on the logic of what to do rather than getting bogged down in the details of how to move safely.

### MoveIt2 envrionment info

*Q: You said that "MoveIt2 builds a detailed 3D model of the robot and its environment". Exactly where in the code does MoveIt2 gets this info about the environment to avoid colisions?*

Ah, you've spotted something absolutely crucial! You're asking the right detective question here. Looking at this code, MoveIt2 is indeed performing collision detection and avoidance, but if you scan through every line of the HandyBot code, you won't find anywhere that explicitly tells MoveIt2 about tables, walls, or other obstacles in the environment. This seems like magic, but there's a very logical explanation.

Let me walk you through exactly how this works, because understanding this will give you deep insight into how professional robotics systems are architected.

**The Missing Piece: Configuration Files and Launch Systems**

The key insight is that not everything a robot needs to know is contained in the Python code we're looking at. In professional robotics development, there's a clear separation between the **application logic** (what the robot should do) and the **system configuration** (how the robot should do it). The HandyBot node we're examining represents the application logic, but the environmental knowledge lives elsewhere in the ROS ecosystem.

Think of it like this: when you write a web application, your Python code doesn't contain information about which server it will run on, what database credentials to use, or what port to listen on. That information lives in configuration files, environment variables, or deployment scripts. Robotics follows the same principle, but the configuration is much more complex because it involves the physical world.

**Where the Environmental Model Actually Lives**

MoveIt2 gets its environmental information through several interconnected systems that are typically set up when the robot system is launched, but before your HandyBot node even starts running. Let me trace through this process:

**The Robot Description System**: There's a whole separate system called the "robot description server" that publishes detailed information about the robot's physical structure. This includes the exact dimensions of every link, the range of motion of every joint, and the collision geometry of every part of the robot. This information is typically stored in files called URDF (Unified Robot Description Format) files, which are like blueprints that describe every physical aspect of the robot.

**The Planning Scene**: MoveIt2 maintains what it calls a "planning scene" - essentially a real-time 3D model of everything in the robot's world. This planning scene gets updated through several channels. Some obstacles might be defined statically in configuration files (like "there's always a table at this position"). Others might be detected dynamically by sensors and published as "collision objects" by other ROS nodes.

**Sensor Integration Nodes**: In a complete system, there are usually additional nodes running that take sensor data (like point clouds from depth cameras) and convert them into collision objects that get published to the planning scene. These nodes might be processing the same camera data that HandyBot uses for object detection, but instead of looking for specific objects to manipulate, they're looking for obstacles to avoid.

**The Launch File Architecture**

In ROS systems, everything typically starts with what's called a "launch file." This is like the conductor of an orchestra, starting up all the different nodes that need to work together. A complete HandyBot launch file might start:

* The robot driver nodes (to communicate with the actual motors)
* The camera driver nodes (to get sensor data)
* The robot description server (to publish the robot's physical model)
* Sensor processing nodes (to convert raw sensor data into collision objects)
* MoveIt2 planning server (with all its configuration)
* Finally, the HandyBot node we've been studying

**Where to Find the Missing Configuration**

If you wanted to see where the environmental information actually comes from in a real HandyBot deployment, you would need to look at:

**The launch files** that start the complete system (usually with .launch.py extensions in ROS2) **The robot description files** (usually .urdf or .xacro files) that describe the robot's physical structure **The MoveIt2 configuration package** that defines things like joint limits, planning algorithms, and static obstacles **The sensor processing configuration** that defines how camera data gets converted into collision information

*Q: Can you tell me where are the configuration files for MoveIt2 in this project?*

Based on the workspace structure, I can see there's a directory called annin\_ar4\_moveit\_config which contains the MoveIt2 configuration files (see subfolder annin\_ar4\_moveit\_config / config )

The MoveIt2 configuration files are located in [config](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) directory. Here are the main configuration files:

1. controllers.yaml - Contains the controller configurations for the robot
2. joint\_limits.yaml - Defines the joint limits for each robot joint
3. kinematics.yaml - Contains the kinematics solver configurations
4. moveit\_servo.yaml - Configuration for MoveIt Servo (real-time control)
5. ompl\_planning.yaml - Settings for the OMPL motion planning pipeline
6. pilz\_planning.yaml - Configuration for the Pilz industrial motion planner

The annin\_ar4\_moveit\_config folder  a custom configuration package specifically made for the AR4 robot.

*Q: These config files will tell moveIt about the configuration of the robot. What about the environment. How will moveit2 in this proejct know about any obstacules?*

 In MoveIt2, obstacles can be added to the planning scene in several ways:

1. Through a collision mesh/model in the URDF/SRDF
2. Through sensors (like cameras or depth sensors) publishing to appropriate ROS2 topics
3. Through scene objects added programmatically

Based on my research, in this project, MoveIt2 can handle obstacles in several ways:

1. Through the Python MoveIt2 Interface ([pymoveit2](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)):
   * You can add collision objects programmatically using the following methods:
     + [add\_collision\_box()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - for box-shaped obstacles
     + [add\_collision\_sphere()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - for spherical obstacles
     + [add\_collision\_cylinder()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - for cylindrical obstacles
     + [add\_collision\_cone()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - for cone-shaped obstacles
     + [add\_collision\_mesh()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - for arbitrary 3D mesh obstacles (requires the [trimesh](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) Python package)
2. There are example scripts showing how to add obstacles:
   * [ex\_collision\_primitive.py](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - demonstrates adding basic shapes
   * [ex\_collision\_mesh.py](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) - demonstrates adding mesh-based obstacles

For example, to add a simple spherical obstacle, you can run:

ros2 run pymoveit2 ex\_collision\_primitive.py --ros-args -p shape:="sphere" -p position:="[0.5, 0.0, 0.5]" -p dimensions:="[0.04]"

Or to add a mesh-based obstacle:

ros2 run pymoveit2 ex\_collision\_mesh.py --ros-args -p action:="add" -p position:="[0.5, 0.0, 0.5]" -p quat\_xyzw:="[0.0, 0.0, -0.707, 0.707]"

All these obstacles are added to the MoveIt2 planning scene, and the motion planner will automatically avoid them when planning paths for the robot. You can also:

* Remove obstacles using [remove\_collision\_object()](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)
* Attach obstacles to the robot using attach\_collision\_object()
* Detach obstacles using detach\_collision\_object()

## detect\_objects\_cb

### Code

    def detect\_objects\_cb(self, msg: String):

        if self.\_last\_rgb\_msg is None:

            self.logger.warning("No RGB image available.")

            return

        class\_names = msg.data.split(",")

        rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

        self.\_last\_detections = self.detect\_objects(rgb\_image, class\_names)

        detected\_classes = [

            class\_names[class\_id]

            for class\_id in self.\_last\_detections.class\_id

        ]

        self.logger.info(f"Detected objects: {detected\_classes}")

### Overview

The detect\_objects\_cb method is a "callback function" - a piece of code that springs into action whenever a specific event occurs; a robot can be awakened from waiting and immediately put to work identifying objects in its visual field. The "cb" suffix stands for "callback".

It sits quietly waiting until something specific happens - namely, until someone publishes a message to the /detect\_objects ROS topic.

This callback was registered during the robot's initialization with this line from the \_\_init\_\_ method:

self.detect\_objects\_sub = self.create\_subscription(

String, "/detect\_objects", self.detect\_objects\_cb, 10)

This project has 2 interfaces: command line or voice: During development and testing, engineers need ways to test individual components without going through the full complexity of the natural language processing system.

The ROS callback interface provides exactly this capability. A robotics engineer can use ROS command-line tools to send direct commands like this from a terminal window. They might type something like ros2 topic pub /detect\_objects std\_msgs/String "data: 'apple,orange,cup'" to trigger object detection immediately, without any involvement from the language processing system.

**The Integration Point: How They Work Together**

Look carefully at the OpenAI assistant integration in the main start method. When the assistant decides it needs to detect objects, it doesn't call detect\_objects\_cb. Instead, it calls the core detect\_objects method directly.

### Step 1: Parsing Human-Readable Commands

class\_names = msg.data.split(",")

The split(",") operation transforms this human-readable string into a Python list like ["apple", "orange", "cup"]. This transformation is preparing the data for the computer vision system, which expects a structured list rather than a comma-separated string.

Someone operating the robot can send commands like /detect\_objects apple,banana,knife through ROS command-line tools.

### Step 2: Image Format Conversion Bridge

rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

We have data in one format (a ROS Image message) that needs to be converted to another format (an OpenCV image array) so that our computer vision algorithms can process it.

### Step 3: Delegating to the Computer Vision Expert

self.\_last\_detections = self.detect\_objects(rgb\_image, class\_names)

## joint\_states\_callback

### Code

    def joint\_states\_callback(self, msg: JointState):

        joint\_state = JointState()

        joint\_state.header = msg.header

        for name in self.arm\_joint\_names:

            for i, joint\_state\_joint\_name in enumerate(msg.name):

                if name == joint\_state\_joint\_name:

                    joint\_state.name.append(name)

                    joint\_state.position.append(msg.position[i])

                    joint\_state.velocity.append(msg.velocity[i])

                    joint\_state.effort.append(msg.effort[i])

self.arm\_joint\_state = joint\_state

### Overview

The joint\_states\_callback method is a callback function triggered whenever a new message is received on the /joint\_states ROS topic (subscribed to in ROSSensorDataSubscriptionsInitialization). Its primary purpose is to process an incoming JointState message (from the sensor\_msgs.msg package), filter it to include only the relevant arm joints (defined in self.arm\_joint\_names), and store the filtered state in the class attribute self.arm\_joint\_state. This filtered state is used elsewhere in the node (e.g., for logging or motion planning with MoveIt2).

The method ensures that only the robot arm's joints (e.g., "joint\_1" to "joint\_6") are extracted, ignoring other joints like the gripper (which is handled separately via GripperInterface). This separation keeps the arm's state clean and focused for manipulation tasks.

### 1: Create a New JointState Object:

joint\_state = JointState()

* Initializes a new, empty JointState message object to store the filtered data.
* This will be populated with only the arm-specific joints.

the expression **JointState()** is a constructor call used to create a new instance of the JointState message class from the sensor\_msgs.msg module in ROS 2.

The JointState message structure, as defined in ROS, is:

# sensor\_msgs/JointState.msg

Header header

string[] name

float64[] position

float64[] velocity

float64[] effort

### 2: Outer Loop Over Arm Joint Names

for name in self.arm\_joint\_names:

This loops over each expected arm joint name (typically 6 iterations): ["joint\_1", "joint\_2", "joint\_3", "joint\_4", "joint\_5", "joint\_6"]. Defined earlier in RobotArmConfigurationMoveIt2

In the joint\_states\_callback method of the tabletop\_handybot\_node.py code, this first for loop iterates over self.arm\_joint\_names to filter the incoming JointState message (msg) and extract data only for the specified arm joints (["joint\_1", "joint\_2", ..., "joint\_6"]). The question asks why this loop is necessary, whether simply appending data from the message would suffice, and if the loop serves as a validation mechanism. Let’s analyze the purpose of the first for loop, explore whether it could be avoided, and clarify its role in validation and filtering.

* **Key Components**:
  + self.arm\_joint\_names: A list of arm joint names (["joint\_1", "joint\_2", "joint\_3", "joint\_4", "joint\_5", "joint\_6"]).
  + msg: The incoming JointState message containing name, position, velocity, and effort arrays for all joints (e.g., arm joints plus gripper joints like gripper\_jaw1\_joint).
  + The outer for loop iterates over self.arm\_joint\_names, and the inner loop searches msg.name for a matching joint name.

**Why the First for Loop is Needed**

The first for loop (for name in self.arm\_joint\_names) serves two critical purposes: **filtering** and **ordering**. It is not just for validation but is essential to ensure that the resulting self.arm\_joint\_state contains only the arm joints’ data in a specific order. Here’s a detailed explanation:

1. **Filtering Specific Joints**:
   * The incoming msg may include data for all joints in the robot, including non-arm joints like the gripper (gripper\_jaw1\_joint) or others (e.g., a mobile base or additional manipulators).
   * The node only cares about the six arm joints defined in self.arm\_joint\_names. The first loop ensures that only these joints are included in the filtered joint\_state.
   * Without this loop, appending all data from msg would include unwanted joints, which could break downstream logic (e.g., motion planning with MoveIt2, which expects exactly the arm joints).
2. **Preserving Order**:
   * The order of joints in msg.name is not guaranteed to match self.arm\_joint\_names. For example, msg.name might be ["gripper\_jaw1\_joint", "joint\_1", "joint\_2", ...], while self.arm\_joint\_names is ["joint\_1", "joint\_2", ..., "joint\_6"].
   * The first loop ensures that joint\_state.name (and its corresponding position, velocity, effort) follows the order of self.arm\_joint\_names. This is critical for consistency, as MoveIt2 and other methods (e.g., flick\_wrist\_while\_release) rely on a fixed joint order (e.g., joint\_positions[4] for the wrist joint).
   * Without the loop, appending msg data directly would preserve msg’s order, potentially misaligning with self.arm\_joint\_names.

### 3: Inner Loop to Find Matching Joint Index

for i, joint\_state\_joint\_name in enumerate(msg.name):

if name == joint\_state\_joint\_name:

*  enumerate(msg.name) iterates over the incoming message's joint names, providing both the index i and the name joint\_state\_joint\_name.
* This searches for a match between the current name (from the outer loop) and the joints in msg.
* Compares the arm joint name with the current name from msg.
* If they match, proceed to append the data;

### Q: When Does the JointState Message Come from ROS?

In ROS, the /joint\_states topic is typically published by a **robot controller**, **driver**, or a node like robot\_state\_publisher to provide real-time updates on the state of a robot’s joints (e.g., position, velocity, and effort). The timing of these messages depends on the specific robot hardware, its controller, and the ROS setup. Here are the key possibilities:

1. **Periodic Updates (Every x Seconds)**:
   * **Most Common Case**: In most ROS setups, the /joint\_states topic is published at a fixed rate, typically ranging from **10 Hz to 100 Hz** (every 0.01 to 0.1 seconds). This rate is determined by the robot’s controller or driver (e.g., a URDF-based robot driver for a Universal Robots arm or a custom controller for the robot ).

**In the Code**: It subscribes to /joint\_states with a queue size of 10:

self.joint\_states\_sub = self.create\_subscription(

JointState, "/joint\_states", self.joint\_states\_callback, 10)

This queue size suggests the node expects frequent messages and can buffer up to 10 if processing lags, consistent with a periodic, high-frequency publication.

## save\_images

### Code

    def save\_images(self, msg):

        if not self.\_last\_rgb\_msg or not self.\_last\_depth\_msg:

            return

        rgb\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_rgb\_msg)

        depth\_image = self.cv\_bridge.imgmsg\_to\_cv2(self.\_last\_depth\_msg)

        save\_dir = msg.data

        cv2.imwrite(

            os.path.join(save\_dir, f"rgb\_image\_{self.n\_frames\_processed}.png"),

            rgb\_image)

        np.save(

            os.path.join(save\_dir, f"depth\_image\_{self.n\_frames\_processed}"),

            depth\_image)

### Purpose

The save\_images method serves the following purposes:

* **Save Images to Disk**: It saves the most recent RGB and depth images, which are stored as ROS Image messages in self.\_last\_rgb\_msg and self.\_last\_depth\_msg, respectively, to a directory specified in the input message.
* **Support Debugging and Analysis**: By saving images with a frame counter in the filename, it enables tracking of image data over time, which is useful for debugging, post-processing, or analyzing the robot’s perception capabilities.
* **Interface with ROS**: The method is triggered by a message on the /save\_images ROS topic, providing an external interface to command the robot to save images.

### Integration with the Codebase

* **ROS Subscription**:
  + The save\_images method is triggered by a subscription to the /save\_images topic, set up in the ROSSensorDataSubscriptionsInitialization method:

self.save\_images\_sub = self.create\_subscription(

String, "/save\_images", self.save\_images, 10)

* + This subscription listens for String messages with a queue size of 10, meaning up to 10 messages can be buffered if the node is busy.
* **Image Sources**:
  + The RGB and depth images are updated by the image\_callback and depth\_callback methods, which are subscribed to /camera/color/image\_raw and /camera/aligned\_depth\_to\_color/image\_raw, respectively.
  + These callbacks store the latest images in self.\_last\_rgb\_msg and self.\_last\_depth\_msg, which save\_images relies on.

## main

### Code

def main():

    rclpy.init()

    node = TabletopHandyBotNode()

    executor = MultiThreadedExecutor(4)

    executor.add\_node(node)

    try:

        executor.spin()

    except KeyboardInterrupt:

        pass

    rclpy.shutdown()

### ROS2 Initialization

rclpy.init()

* Calls rclpy.init() from the rclpy library (ROS2's Python client library) to initialize the ROS2 runtime environment.
* This sets up the underlying ROS2 context, including communication layers (e.g., DDS middleware for topics, services, and actions). Without this, no ROS2 nodes or messaging can function.
* It's the first step in any ROS2 Python program. If initialization fails (e.g., due to environment issues), it would raise an exception, but the code doesn't handle it explicitly here.
* Relates to the rest of the code: The TabletopHandyBotNode class inherits from rclpy.node.Node, so this init is required before creating any nodes.

### Node Creation

node = TabletopHandyBotNode()

* Instantiates an object of the TabletopHandyBotNode class, which is the core of the script.
* This constructor call triggers the node's \_\_init\_\_() method

### Executor Setup

executor = MultiThreadedExecutor(4)

* Creates a MultiThreadedExecutor from rclpy.executors with 4 threads.
* An executor in ROS2 manages the execution of node callbacks (e.g., when a message arrives on a subscribed topic like /prompt or /camera/color/image\_raw).
* Using a multi-threaded executor (instead of the default single-threaded one) allows concurrent processing of callbacks, which is crucial for this node because it handles time-sensitive tasks like image processing, AI calls, and robot motion in parallel. For example:
  + One thread might process a depth image callback while another handles a prompt.
  + The node uses ReentrantCallbackGroup and MutuallyExclusiveCallbackGroup in its subscriptions (e.g., for /prompt), which benefit from multi-threading to avoid blocking.
* The number 4 is arbitrary but chosen for balance (too few threads could cause bottlenecks; too many could increase overhead).

# openai\_assistant.py

## Code

from openai import OpenAI  # pylint: disable=import-self

from openai.types.beta import Assistant

# pylint: disable=line-too-long

def get\_or\_create\_assistant(client: OpenAI,

                            assistant\_id: str = "") -> Assistant:

    if assistant\_id:

        return client.beta.assistants.retrieve(assistant\_id)

    return client.beta.assistants.create(

        name="Tabletop Assistant",

        instructions=(

            "You are a robot arm mounted on a table. Write and run code to "

            "do tasks on the table. You can only pick up one object at a time."

        ),

        model="gpt-4o",

        temperature=0.01,

        tools=[{

            "type": "function",

            "function": {

                "name": "detect\_objects",

                "description":

                "Detect objects in the field of view of the camera",

                "parameters": {

                    "type": "object",

                    "properties": {

                        "object\_classes": {

                            "type":

                            "string",

                            "description":

                            ("Object classes to detect, comma separated"

                             "For example: horses,rivers,plain"),

                        }

                    },

                    "required": ["object\_classes"],

                },

            },

        }, {

            "type": "function",

            "function": {

                "name": "pick\_object",

                "description":

                "Pick up an object from the output of get\_objects.",

                "parameters": {

                    "type": "object",

                    "properties": {

                        "object\_index": {

                            "type":

                            "integer",

                            "description":

                            "index of target object in the detected objects list to execute the action for."

                        }

                    },

                    "required": ["object\_index"]

                }

            }

        }, {

            "type": "function",

            "function": {

                "name": "move\_above\_object\_and\_release",

                "description":

                "move the end effector above the object and release the gripper. The object is from the output of get\_objects",

                "parameters": {

                    "type": "object",

                    "properties": {

                        "object\_index": {

                            "type":

                            "integer",

                            "description":

                            "index of target object in the detected objects list to execute the action for."

                        }

                    },

                    "required": ["object\_index"]

                }

            }

        }, {

            "type": "function",

            "function": {

                "name": "release\_gripper",

                "description": "Open up the gripper",

                "parameters": {

                    "type": "object",

                    "properties": {},

                    "required": []

                }

            }

        }, {

            "type": "function",

            "function": {

                "name": "flick\_wrist\_while\_release",

                "description":

                "Flick the wrist while releasing the gripper, basically tossing the object.",

                "parameters": {

                    "type": "object",

                    "properties": {},

                    "required": []

                }

            }

        }],

    )

## Explanation

This code defines a Python function that creates or retrieves an OpenAI Assistant specifically designed to control a robotic arm on a tabletop. Here's what it does:

**Main Function**

The get\_or\_create\_assistant() function either retrieves an existing assistant by ID or creates a new one if no ID is provided.

**Assistant Configuration**

The assistant is configured as a "Tabletop Assistant" with these characteristics:

* **Identity**: A robot arm mounted on a table that can manipulate objects
* **Limitation**: Can only pick up one object at a time
* **Model**: Uses GPT-4o with very low temperature (0.01) for consistent, deterministic responses
* **Purpose**: Write and execute code to perform tasks on the table

**Available Tools/Functions**

The assistant has access to five robotic functions:

1. **detect\_objects** - Uses computer vision to identify objects in the camera's field of view, with the ability to specify which object classes to look for
2. **pick\_object** - Grabs a specific object using its index from the detection results
3. **move\_above\_object\_and\_release** - Positions the robotic arm above a target object and opens the gripper to drop whatever it's holding
4. **release\_gripper** - Simply opens the gripper to release any held object
5. **flick\_wrist\_while\_release** - Performs a more dynamic release by flicking the wrist while opening the gripper, effectively tossing the object

### The Foundation: What Are OpenAI Assistants?

Think of OpenAI Assistants as persistent AI agents that you can customize for specific tasks.

**Breaking Down the Creation Process**

When you call client.beta.assistants.create(), you're essentially filling out a job application for an AI worker.

# audio\_prompt\_node.py

## Code

import rclpy

import rclpy.node

from std\_msgs.msg import Empty, String

from whisper\_mic import WhisperMic

class AudioPromptNode(rclpy.node.Node):

    """ROS2 Node that listens for a /listen message and publishes the

    result of the audio prompt to /prompt."""

    def \_\_init\_\_(self):

        super().\_\_init\_\_("audio\_prompt\_node")

        self.whisper\_mic = WhisperMic()

        self.prompt\_pub = self.create\_publisher(String, "/prompt", 10)

        self.listen\_sub = self.create\_subscription(Empty, "/listen",

                                                   self.listen, 10)

        self.\_logger.info("Audio Prompt Node Initialized")

    def listen(self, \_: Empty):

        result = self.whisper\_mic.listen(timeout=10.0, phrase\_time\_limit=10.0)

        self.\_logger.info(f"Prompt: {result}")

        self.prompt\_pub.publish(String(data=result))

def main():

    rclpy.init()

    node = AudioPromptNode()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        pass

    rclpy.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Explanation

This code creates a ROS2 (Robot Operating System 2) node that serves as a bridge between speech input and text output in a robotic system. Let me walk you through how it works, starting with the big picture and then diving into the details.

**The Overall Purpose**

Imagine you want to give voice commands to a robot. This node acts like the robot's "ears" - it waits for a signal to start listening, captures your speech through a microphone, converts that speech to text using AI, and then sends that text to other parts of the robot system that can act on your commands.

**Understanding the Key Components**

The code builds on two main technologies that work together. ROS2 is a framework that helps different parts of a robot communicate with each other through a publish-subscribe system where different components can send and receive messages on specific "topics." The WhisperMic component uses OpenAI's Whisper AI model to convert speech to text in real-time.

**Breaking Down the Class Structure**

The AudioPromptNode class inherits from rclpy.node.Node, which gives it all the capabilities needed to participate in the ROS2 communication network. When you inherit from a ROS2 Node, you're essentially saying "this class can send and receive messages with other parts of the robot system."

Let's examine what happens during initialization. The node creates a WhisperMic object, which handles the actual speech recognition. It then sets up two communication channels: a publisher that will send text messages on the "/prompt" topic, and a subscriber that listens for trigger messages on the "/listen" topic. The number "10" in both cases refers to the queue size - how many messages can be buffered if the system gets busy.

**The Communication Flow**

Here's how the system works in practice. When another part of the robot system wants to capture voice input, it publishes an Empty message to the "/listen" topic. An Empty message is exactly what it sounds like - it carries no data, just serves as a signal or trigger. Think of it like ringing a doorbell - you don't need to send any information, just the fact that you rang it.

When this node receives that trigger, its listen method gets called automatically. This method then activates the microphone and starts the speech recognition process. The WhisperMic listens for up to 10 seconds total (timeout parameter), but will also stop listening if there's a pause of 10 seconds in speech (phrase\_time\_limit parameter). These parameters prevent the system from listening indefinitely if no one is speaking.

**The Speech Recognition Process**

The whisper\_mic.listen() method does several things behind the scenes. It captures audio from the microphone, processes it through the Whisper AI model to convert speech to text, and returns the resulting text string. The beauty of using WhisperMic is that it handles all the difficult audio processing, noise filtering, and AI inference internally.

Once the speech is converted to text, the node logs this information for debugging purposes and then publishes the text as a String message on the "/prompt" topic. Other nodes in the robot system can subscribe to this topic to receive and act on these voice commands.

**The Main Function and Program Lifecycle**

The main() function follows a standard ROS2 pattern. It initializes the ROS2 system, creates an instance of the AudioPromptNode, and then enters a spinning loop where it continuously processes incoming messages. The try-except block ensures that if someone presses Ctrl+C, the program shuts down gracefully rather than crashing.

**Practical Applications**

Consider how this might work in a real robot scenario. You could have a cleaning robot where you say "clean the kitchen" after pressing a button. The button press triggers an Empty message to "/listen", this node captures your speech and converts it to text, then publishes "clean the kitchen" to "/prompt". A natural language processing node could subscribe to "/prompt", understand the command, and direct the robot to navigate to and clean the kitchen.

According to the README.md, you can trigger this listening functionality by publishing an empty message to the "/listen" topic using the command:

ros2 topic pub --once /listen std\_msgs/msg/Empty "{}"

*But if I want it to be continuous listening, I would need to make this call multiple times?*

Yes, with the current implementation, you would need to make multiple calls to the "/listen" topic since each listen operation has a timeout of 10 seconds (as seen in the code: [timeout=10.0, phrase\_time\_limit=10.0](vscode-file://vscode-app/c:/Users/pinto/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)).

**Whisper** is a general-purpose speech recognition model from OpenAI that's trained on a large dataset of diverse audio

**WhisperMic** is a third-party project created by a developer named mallorbc that allows you to use a microphone with OpenAI Whisper in real time. [GitHub](https://github.com/mallorbc/whisper_mic)[GitHub](https://github.com/openai/whisper/discussions/75) It's not made by OpenAI themselves, but rather by the community to solve the practical problem of getting microphone input to work seamlessly with Whisper.

# point\_cloud\_conversation.py

## Code

import numpy as np

from sensor\_msgs.msg import PointCloud2, PointField

from std\_msgs.msg import Header

def point\_cloud\_to\_msg(points, parent\_frame):

    """Creates a point cloud message.

    Args:

        points: Nx3 array of xyz positions.

        parent\_frame: frame in which the point cloud is defined

    Returns:

        sensor\_msgs/PointCloud2 message

    Code source:

        https://gist.github.com/pgorczak/5c717baa44479fa064eb8d33ea4587e0

    References:

        http://docs.ros.org/melodic/api/sensor\_msgs/html/msg/PointCloud2.html

        http://docs.ros.org/melodic/api/sensor\_msgs/html/msg/PointField.html

        http://docs.ros.org/melodic/api/std\_msgs/html/msg/Header.html

    """

    # In a PointCloud2 message, the point cloud is stored as an byte

    # array. In order to unpack it, we also include some parameters

    # which desribes the size of each individual point.

    ros\_dtype = PointField.FLOAT32

    dtype = np.float32

    itemsize = np.dtype(dtype).itemsize  # A 32-bit float takes 4 bytes.

    data = points.astype(dtype).tobytes()

    # The fields specify what the bytes represents. The first 4 bytes

    # represents the x-coordinate, the next 4 the y-coordinate, etc.

    fields = [

        PointField(name=n, offset=i \* itemsize, datatype=ros\_dtype, count=1)

        for i, n in enumerate("xyz")

    ]

    # The PointCloud2 message also has a header which specifies which

    # coordinate frame it is represented in.

    header = Header(frame\_id=parent\_frame)

    return PointCloud2(

        header=header,

        height=1,

        width=points.shape[0],

        is\_dense=False,

        is\_bigendian=False,

        fields=fields,

        point\_step=(itemsize \* 3),  # Every point consists of three float32s.

        row\_step=(itemsize \* 3 \* points.shape[0]),

        data=data,

    )

## Explanation

This Python file creates a function that converts 3D point cloud data into a ROS (Robot Operating System) message format. Let me break this down step by step to help you understand what's happening and why it's structured this way.

**What is a Point Cloud?**

First, let's establish what we're working with. A point cloud is a collection of 3D points in space, where each point has X, Y, and Z coordinates. Think of it like a digital representation of objects or environments captured by sensors like LiDAR scanners or depth cameras. Imagine taking thousands of tiny dots and placing them in 3D space to outline the shape of a car, building, or any other object.

**The Purpose of This Function**

The point\_cloud\_to\_msg function takes simple point cloud data (just an array of XYZ coordinates) and packages it into a standardized ROS message format called **PointCloud2**. This is necessary because ROS systems need data in specific message formats to communicate between different software components.

**Understanding the Input Parameters**

The function accepts two inputs:

* points: An Nx3 NumPy array where N is the number of points, and each row contains the X, Y, Z coordinates of one point
* parent\_frame: A string that identifies which coordinate system these points are measured in (for example, "camera\_frame" or "world\_frame")

**The Data Conversion Process**

ROS stores point cloud data as a raw byte array rather than keeping it as floating-point numbers. This might seem counterintuitive at first, but there's a good reason: it allows for very flexible data storage and efficient memory usage.

The function first defines the data type it will use - 32-bit floating-point numbers. It calculates that each float32 takes 4 bytes of memory. Then it converts the entire points array into a raw byte sequence using tobytes(). Think of this like taking a book written in English and converting it to binary code - the information is the same, but it's in a different format.

**Describing the Data Structure**

Since the data is now just a stream of bytes, the function needs to create a "map" that tells other systems how to interpret these bytes. This is done through the fields array. Each field describes one coordinate (X, Y, or Z):

* name: What this coordinate represents ("x", "y", or "z")
* offset: How many bytes into the data stream this coordinate starts
* datatype: What type of number this is (FLOAT32)
* count: How many values of this type (just 1 for each coordinate)

For example, if you have a point at coordinates (1.5, 2.3, -0.8), the first 4 bytes represent 1.5, the next 4 bytes represent 2.3, and the final 4 bytes represent -0.8.

**Building the Complete Message**

The function then constructs the full **PointCloud2** message with several important pieces of metadata:

The header contains the coordinate frame information - this is crucial for robots because they need to know "relative to what" these coordinates are measured. If your robot's camera sees a point 2 meters away, that's very different from a point that's 2 meters away from the robot's base.

The height and width parameters describe how the points are organized. Setting height to 1 and width to the number of points means we're treating this as an unorganized point cloud (just a list of points) rather than an organized grid like you might get from a camera.

**Memory Layout Information**

Finally, the function specifies how the data is laid out in memory:

* point\_step: How many bytes each complete point takes (12 bytes for three 4-byte floats)
* row\_step: How many bytes each row takes (since we only have one row, this equals the total data size)

**Why This Complexity?**

You might wonder why this conversion is necessary when we could just pass around arrays of coordinates. The answer lies in ROS's design philosophy: standardization and interoperability. By converting everything to this common format, any ROS node can process point clouds regardless of where they came from or how they were originally stored. A point cloud from a LiDAR sensor, a depth camera, or even a simulation will all look identical once converted to this format.

This standardization is similar to how web browsers all understand HTML - it doesn't matter whether you wrote your webpage in a text editor, a sophisticated IDE, or generated it programmatically. Once it's in HTML format, any browser can display it correctly

Documentation for PointCloud2

<https://docs.ros.org/en/jazzy/p/sensor_msgs/msg/PointCloud2.html>

# This message holds a collection of N-dimensional points, which may

# contain additional information such as normals, intensity, etc. The

# point data is stored as a binary blob, its layout described by the

# contents of the "fields" array.

#

# The point cloud data may be organized 2d (image-like) or 1d (unordered).

# Point clouds organized as 2d images may be produced by camera depth sensors

# such as stereo or time-of-flight.

# Time of sensor data acquisition, and the coordinate frame ID (for 3d points).

std\_msgs/Header header

# 2D structure of the point cloud. If the cloud is unordered, height is

# 1 and width is the length of the point cloud.

uint32 height

uint32 width

# Describes the channels and their layout in the binary data blob.

PointField[] fields

bool is\_bigendian # Is this data bigendian?

uint32 point\_step # Length of a point in bytes

uint32 row\_step # Length of a row in bytes

uint8[] data # Actual point data, size is (row\_step\*height)

bool is\_dense # True if there are no invalid points

## PointField

<https://docs.ros.org/en/jazzy/p/sensor_msgs/msg/PointField.html>

# This message holds the description of one point entry in the

# PointCloud2 message format.

uint8 INT8 = 1

uint8 UINT8 = 2

uint8 INT16 = 3

uint8 UINT16 = 4

uint8 INT32 = 5

uint8 UINT32 = 6

uint8 FLOAT32 = 7

uint8 FLOAT64 = 8

# Common PointField names are x, y, z, intensity, rgb, rgba

string name # Name of field

uint32 offset # Offset from start of point struct

uint8 datatype # Datatype enumeration, see above

uint32 count # How many elements in the field

# run.launch.py (in launch folder) – original version

## Code

import os

import yaml

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

from launch import LaunchDescription

from launch.actions import IncludeLaunchDescription, TimerAction

from launch.launch\_description\_sources import PythonLaunchDescriptionSource

from launch\_ros.actions import Node

def load\_yaml(package\_name, file\_name):

package\_path = get\_package\_share\_directory(package\_name)

absolute\_file\_path = os.path.join(package\_path, file\_name)

with open(absolute\_file\_path, "r", encoding="utf-8") as file:

return yaml.safe\_load(file)

def generate\_launch\_description():

realsense\_args = {

"enable\_rgbd": "true",

"enable\_sync": "true",

"align\_depth.enable": "true",

"enable\_color": "true",

"enable\_depth": "true",

"depth\_module.depth\_profile": "640x480x30",

"depth\_module.infra\_profile": "640x480x30",

}

realsense = IncludeLaunchDescription(

PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("realsense2\_camera"),

"launch", "rs\_launch.py")

]),

launch\_arguments=realsense\_args.items())

calibration\_tf\_publisher = IncludeLaunchDescription(

PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("easy\_handeye2"),

"launch", "publish.launch.py")

]),

launch\_arguments={"name": "ar4\_calibration"}.items())

delay\_calibration\_tf\_publisher = TimerAction(

actions=[calibration\_tf\_publisher],

period=2.0,

)

ar\_moveit\_launch = PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("ar\_moveit\_config"), "launch",

"ar\_moveit.launch.py")

])

ar\_moveit\_args = {

"include\_gripper": "True",

"rviz\_config\_file": "moveit\_with\_camera.rviz"

}.items()

ar\_moveit = IncludeLaunchDescription(ar\_moveit\_launch,

launch\_arguments=ar\_moveit\_args)

tabletop\_handybot\_node = Node(

package="tabletop\_handybot",

executable="tabletop\_handybot\_node",

name="tabletop\_handybot\_node",

output="screen",

)

audio\_prompt\_node = Node(

package="tabletop\_handybot",

executable="audio\_prompt\_node",

name="audio\_prompt\_node",

output="screen",

)

return LaunchDescription([

realsense, delay\_calibration\_tf\_publisher, ar\_moveit,

audio\_prompt\_node, tabletop\_handybot\_node

])

## Overview

This is a ROS 2 (Robot Operating System 2) launch file written in Python that coordinates the startup of multiple components for a robotic system called "tabletop\_handybot". Here's a detailed breakdown:

**Purpose**

The launch file orchestrates the initialization of a complete robotic vision and manipulation system, including camera hardware, motion planning, hand-eye calibration, and custom application nodes.

## Key Components

### 1. RealSense Camera Setup

realsense\_args = {

"enable\_rgbd": "true",

"enable\_sync": "true",

"align\_depth.enable": "true",

"enable\_color": "true",

"enable\_depth": "true",

"depth\_module.depth\_profile": "640x480x30",

"depth\_module.infra\_profile": "640x480x30",

}

Configures an Intel RealSense camera with:

* RGBD (color + depth) output enabled
* Synchronized color and depth streams
* 640x480 resolution at 30 FPS for both depth and infrared streams
* Aligned depth data (depth pixels correspond to color pixels)

### 2. Hand-Eye Calibration

calibration\_tf\_publisher = IncludeLaunchDescription(...)

delay\_calibration\_tf\_publisher = TimerAction(actions=[calibration\_tf\_publisher], period=2.0)

* Uses the easy\_handeye2 package to publish transformation data between the robot and camera
* Includes a 2-second delay before starting the calibration publisher (likely to ensure other components are ready)
* The calibration is named "ar4\_calibration"

### 3. Motion Planning (MoveIt)

ar\_moveit\_args = {

"include\_gripper": "True",

"rviz\_config\_file": "moveit\_with\_camera.rviz"

}

* Launches MoveIt motion planning for what appears to be an AR4 robot arm
* Includes gripper control
* Uses a custom RViz configuration that incorporates camera visualization

### 4. Application Nodes

Two custom nodes from the tabletop\_handybot package:

* **tabletop\_handybot\_node**: Main application logic for tabletop manipulation tasks
* **audio\_prompt\_node**: Handles audio-based user interaction/prompting

## System Architecture

This appears to be a tabletop manipulation robot system that:

1. Uses computer vision (RealSense camera) to perceive objects on a table
2. Employs hand-eye calibration to accurately relate camera coordinates to robot coordinates
3. Uses MoveIt for motion planning and execution
4. Provides audio feedback/interaction capabilities
5. Coordinates all these components through the main tabletop\_handybot\_node

## Launch Sequence

The components start in this order:

1. RealSense camera
2. MoveIt motion planning system
3. Audio prompt node
4. Main tabletop handybot node
5. Hand-eye calibration publisher (delayed by 2 seconds)

The utility function load\_yaml() suggests the system also supports YAML-based configuration files, though none are used in this particular launch file.

## load\_yaml method – never used!

### Code

def load\_yaml(package\_name, file\_name):

    package\_path = get\_package\_share\_directory(package\_name)

    absolute\_file\_path = os.path.join(package\_path, file\_name)

    with open(absolute\_file\_path, "r", encoding="utf-8") as file:

        return yaml.safe\_load(file)

The load\_yaml() function is a utility function designed to load YAML configuration files from ROS 2 packages. Let me break it down in detail:

### Function Signature

def load\_yaml(package\_name, file\_name):

* **package\_name**: The name of the ROS 2 package containing the YAML file
* **file\_name**: The relative path to the YAML file within that package

**Step-by-Step Breakdown:**

### 1. Package Path Resolution

package\_path = get\_package\_share\_directory(package\_name)

* Uses get\_package\_share\_directory() from ament\_index\_python.packages
* This function finds the absolute path to the "share" directory of the specified ROS 2 package
* In ROS 2, the share directory typically contains configuration files, launch files, and other resources
* For example, if package\_name is "my\_robot", this might return something like /opt/ros/humble/share/my\_robot or ~/ros2\_ws/install/my\_robot/share/my\_robot

### 2. Full File Path Construction

absolute\_file\_path = os.path.join(package\_path, file\_name)

* Combines the package's share directory path with the relative file path
* Uses os.path.join() for cross-platform path handling
* Example: If package\_path is /opt/ros/humble/share/my\_robot and file\_name is config/robot\_params.yaml, the result would be /opt/ros/humble/share/my\_robot/config/robot\_params.yaml

### 3. File Reading and YAML Parsing

with open(absolute\_file\_path, "r", encoding="utf-8") as file:

return yaml.safe\_load(file)

* Opens the file in read mode with UTF-8 encoding
* Uses a context manager (with statement) to ensure proper file closure
* yaml.safe\_load() parses the YAML content into Python data structures (dictionaries, lists, etc.)
* safe\_load() is preferred over load() for security reasons as it only loads standard YAML tags

**Usage Example**

If you had a YAML file at my\_robot\_package/share/my\_robot\_package/config/settings.yaml:

robot\_settings:

max\_velocity: 1.5

joint\_limits: [180, 90, 270]

gripper\_enabled: true

You could load it with:

config = load\_yaml("my\_robot\_package", "config/settings.yaml")

*# config would be a Python dictionary containing the YAML data*

**Why This Function Exists**

1. **Package-Relative Paths**: ROS 2 packages can be installed in different locations, so this function provides a reliable way to find files within packages
2. **Reusability**: Common pattern for loading configuration files in ROS 2 launch files
3. **Error Handling**: Centralizes the file loading logic (though this particular implementation doesn't include explicit error handling)

**Note About Usage in This Launch File**

Interestingly, while the load\_yaml() function is defined in this launch file, it's never actually used. This suggests either:

* It was used in an earlier version and left for potential future use
* It's a template/boilerplate that the developer included for convenience
* There are configuration files that could be loaded but are currently handled differently

The function is a common utility pattern in ROS 2 launch files when you need to load robot parameters, controller configurations, or other settings from YAML files.

## generate\_launch\_description

In ROS 2, a LaunchDescription is like a recipe that tells the system exactly how to start up your robotic application.

### Building Block 1: The Camera System

Let's start with the first major component:

realsense\_args = {

"enable\_rgbd": "true",

"enable\_sync": "true",

"align\_depth.enable": "true",

"enable\_color": "true",

"enable\_depth": "true",

"depth\_module.depth\_profile": "640x480x30",

"depth\_module.infra\_profile": "640x480x30",

}

Think of this configuration as setting up a very sophisticated pair of eyes for your robot. The RealSense camera doesn't just take pictures like a phone camera—it creates a three-dimensional understanding of the world. Here's what each setting accomplishes:

The enable\_rgbd setting combines color information (RGB) with depth information (D) to create rich, three-dimensional data about what the camera sees. Imagine being able to not just see that there's a red apple on the table, but also knowing exactly how far away it is and its precise shape in space.

The enable\_sync parameter ensures that the color and depth information are captured at exactly the same moment. This might seem obvious, but cameras actually capture images quite rapidly, and without synchronization, you might have color data from one moment and depth data from a slightly different moment, creating mismatches that could confuse your robot.

The alignment setting (align\_depth.enable) is particularly clever. Normally, the color camera and depth sensor have slightly different viewpoints, like your two eyes seeing things from slightly different angles. This setting mathematically adjusts the depth data so it perfectly matches what the color camera sees, creating a unified view of the world.

The profile settings specify that both depth and infrared data should be captured at 640x480 resolution at 30 frames per second. This represents a balance between having enough detail to recognize objects and maintaining smooth, real-time performance.

### Building Block 2: Launching the Camera

realsense = IncludeLaunchDescription(

PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("realsense2\_camera"),

"launch", "rs\_launch.py")

]),

launch\_arguments=realsense\_args.items())

Here's where we see a powerful pattern in ROS 2 development: composition and reuse. Instead of writing all the camera startup code from scratch, this function includes another launch file that specialists have already created and tested. It's like using a pre-built, high-quality component in an engineering project rather than manufacturing everything yourself.

The IncludeLaunchDescription approach allows us to leverage the expertise of the camera manufacturer's engineers while still customizing the behavior through our arguments. Notice how we pass our carefully crafted configuration (realsense\_args) to this included launch file, tailoring its behavior to our specific needs.

### Building Block 3: Hand-Eye Calibration with Timing

Now we encounter something more sophisticated:

calibration\_tf\_publisher = IncludeLaunchDescription(

PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("easy\_handeye2"),

"launch", "publish.launch.py")

]),

launch\_arguments={"name": "ar4\_calibration"}.items())

delay\_calibration\_tf\_publisher = TimerAction(

actions=[calibration\_tf\_publisher],

period=2.0,

)

This section addresses one of the fundamental challenges in robotics: helping the robot understand the relationship between what it sees and where its arm can reach. Think about how you naturally coordinate your vision with your hand movements—when you see a coffee cup, your brain automatically knows how to direct your hand to grasp it. Robots need explicit calibration to achieve this same coordination.

The hand-eye calibration system publishes what's called a "transform" that mathematically describes the relationship between the camera's coordinate system and the robot arm's coordinate system. This transform is like a translation dictionary that converts "the object is 30 centimeters in front of the camera" into "move the robot arm to position X, Y, Z."

But here's where the timing becomes crucial. The TimerAction wrapper introduces a 2-second delay before starting the calibration publisher. Why this delay? Think of it like waiting for all the musicians to tune their instruments before the conductor begins. The camera needs time to initialize, the robot arm needs to establish its position, and various software components need to start communicating. The delay ensures that when the calibration system starts publishing transforms, all the components it depends on are ready to receive and use that information.

### Building Block 4: Motion Planning Intelligence

ar\_moveit\_launch = PythonLaunchDescriptionSource([

os.path.join(get\_package\_share\_directory("ar\_moveit\_config"), "launch",

"ar\_moveit.launch.py")

])

ar\_moveit\_args = {

"include\_gripper": "True",

"rviz\_config\_file": "moveit\_with\_camera.rviz"

}.items()

ar\_moveit = IncludeLaunchDescription(ar\_moveit\_launch,

launch\_arguments=ar\_moveit\_args)

MoveIt represents the brain of the robotic arm—the system that figures out how to move from point A to point B without hitting obstacles or damaging itself. When you reach for that coffee cup, your brain automatically calculates a smooth path that avoids hitting other objects on the table, keeps your elbow from hyperextending, and positions your hand at the right angle for grasping. MoveIt does this same complex planning for robots.

The configuration here tells us several important things about this particular robot setup. First, include\_gripper: "True" indicates that this isn't just an arm that points at things—it has an end effector that can actually grasp and manipulate objects. The gripper needs to be included in all motion planning calculations because its size and shape affect what paths are possible.

The moveit\_with\_camera.rviz configuration file suggests that the system provides sophisticated visualization capabilities. RViz is like having a detailed 3D representation of your robot and its environment, allowing operators to see planned paths, sensor data, and the robot's understanding of its surroundings all in one integrated view.

**Building Block 5: The Application Logic**

tabletop\_handybot\_node = Node(

package="tabletop\_handybot",

executable="tabletop\_handybot\_node",

name="tabletop\_handybot\_node",

output="screen",

)

audio\_prompt\_node = Node(

package="tabletop\_handybot",

executable="audio\_prompt\_node",

name="audio\_prompt\_node",

output="screen",

)

These two nodes represent the custom application logic—the specific intelligence that makes this system a "tabletop handybot" rather than just a generic robotic arm with a camera.

The tabletop\_handybot\_node likely contains the high-level decision-making logic: identifying objects on the table, determining what actions to take, coordinating the camera and arm to perform manipulation tasks, and managing the overall workflow of the application.

The audio\_prompt\_node adds a human-friendly interface to the system. Rather than requiring users to type commands or click buttons, this component probably allows people to give voice commands or provides audio feedback about what the robot is doing. This makes the system more accessible and natural to interact with.

The output="screen" parameter for both nodes ensures that any status messages, debugging information, or error reports from these components will be visible in the terminal where the launch file is running. This is invaluable for understanding what the system is doing and troubleshooting any issues that arise.

**The Grand Orchestration**

Finally, we see how all these components come together:

return LaunchDescription([

realsense, delay\_calibration\_tf\_publisher, ar\_moveit,

audio\_prompt\_node, tabletop\_handybot\_node

])

This final line creates the actual launch description that ROS 2 will execute. Notice the careful ordering of components. The camera (realsense) starts first because other components depend on its data. The motion planning system (ar\_moveit) and audio interface (audio\_prompt\_node) start next, providing the foundational capabilities the main application needs. The primary application logic (tabletop\_handybot\_node) starts after these dependencies are in place. Finally, the calibration system starts after its 2-second delay, ensuring everything else is ready.