Advanced Remanufacturing and Technology Centre
ARTC

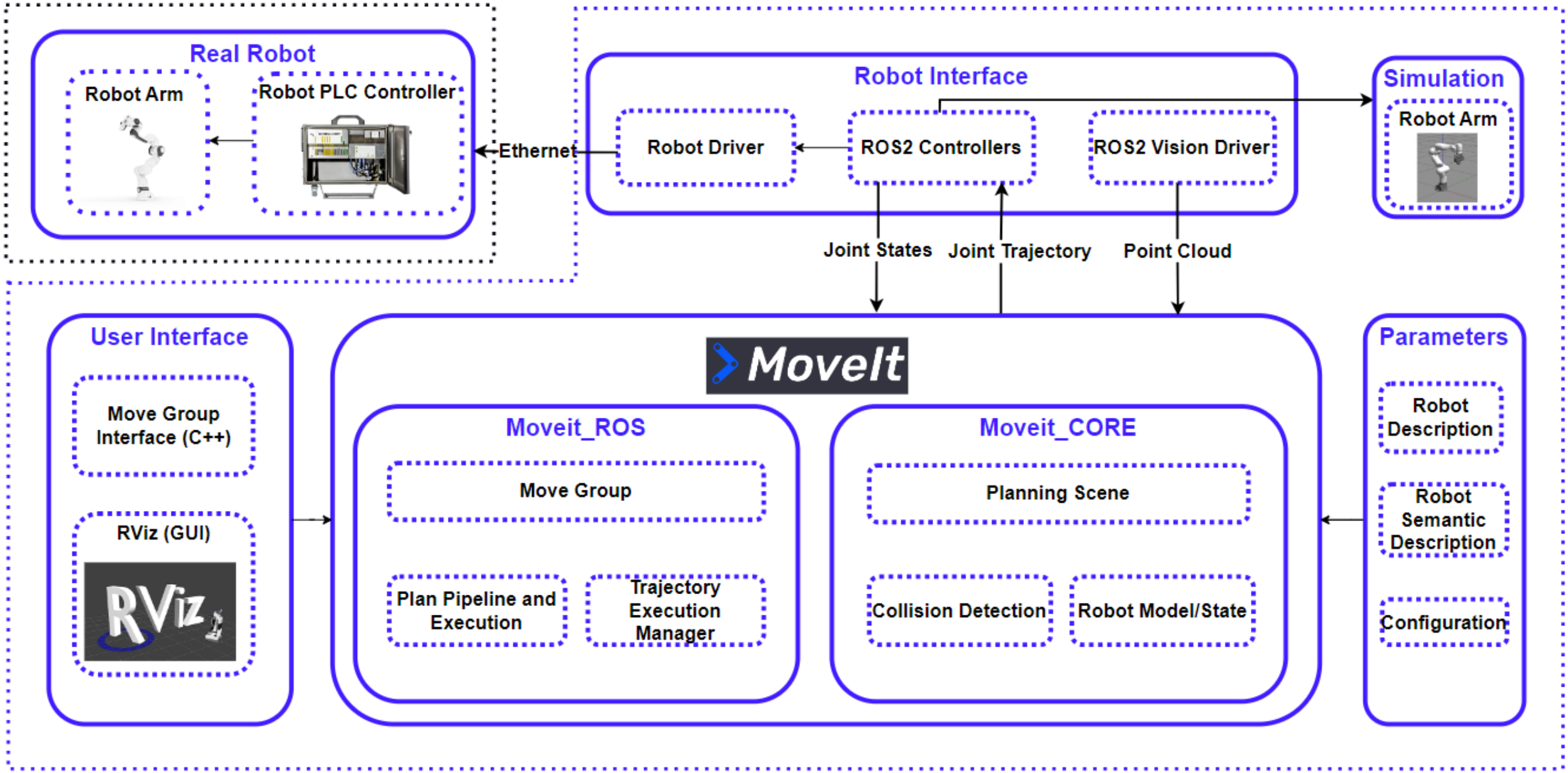
ROS-INDUSTRIAL ASIA PACIFIC MANIPULATION TRAINING

DAY 1

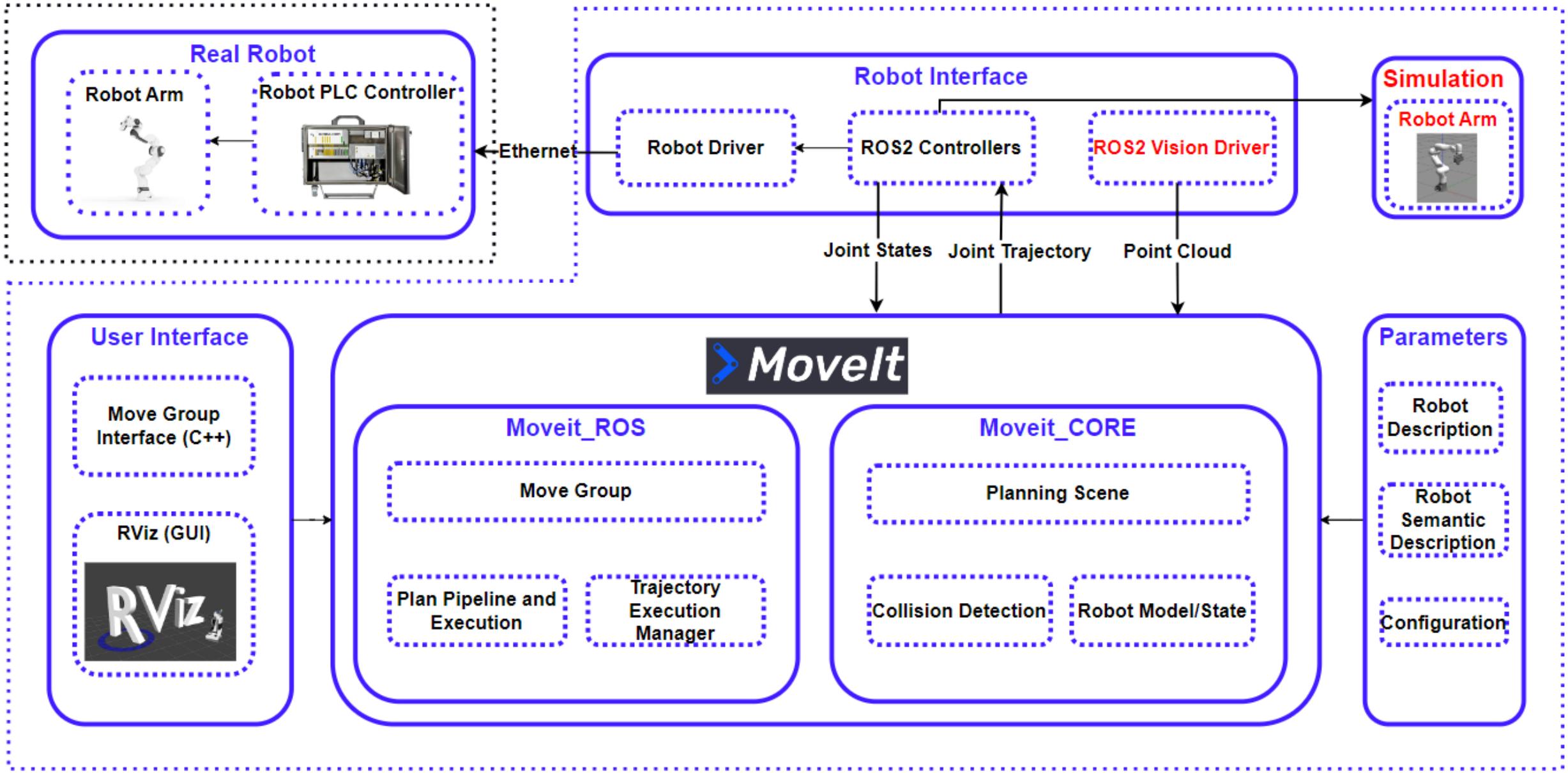
Shalman Khan

4 December 2023

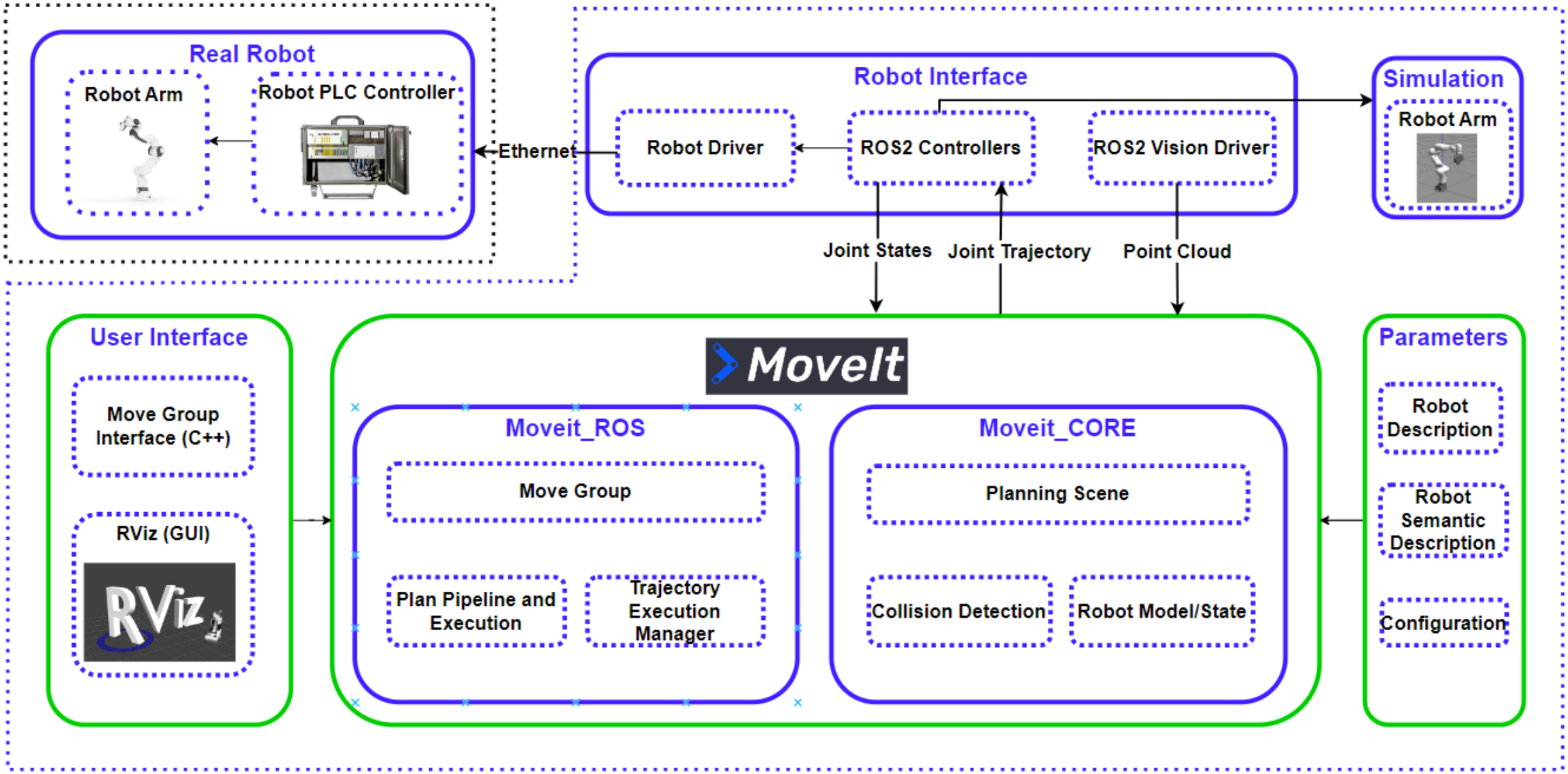
ROS2 Manipulation Architecture | [High – Level]



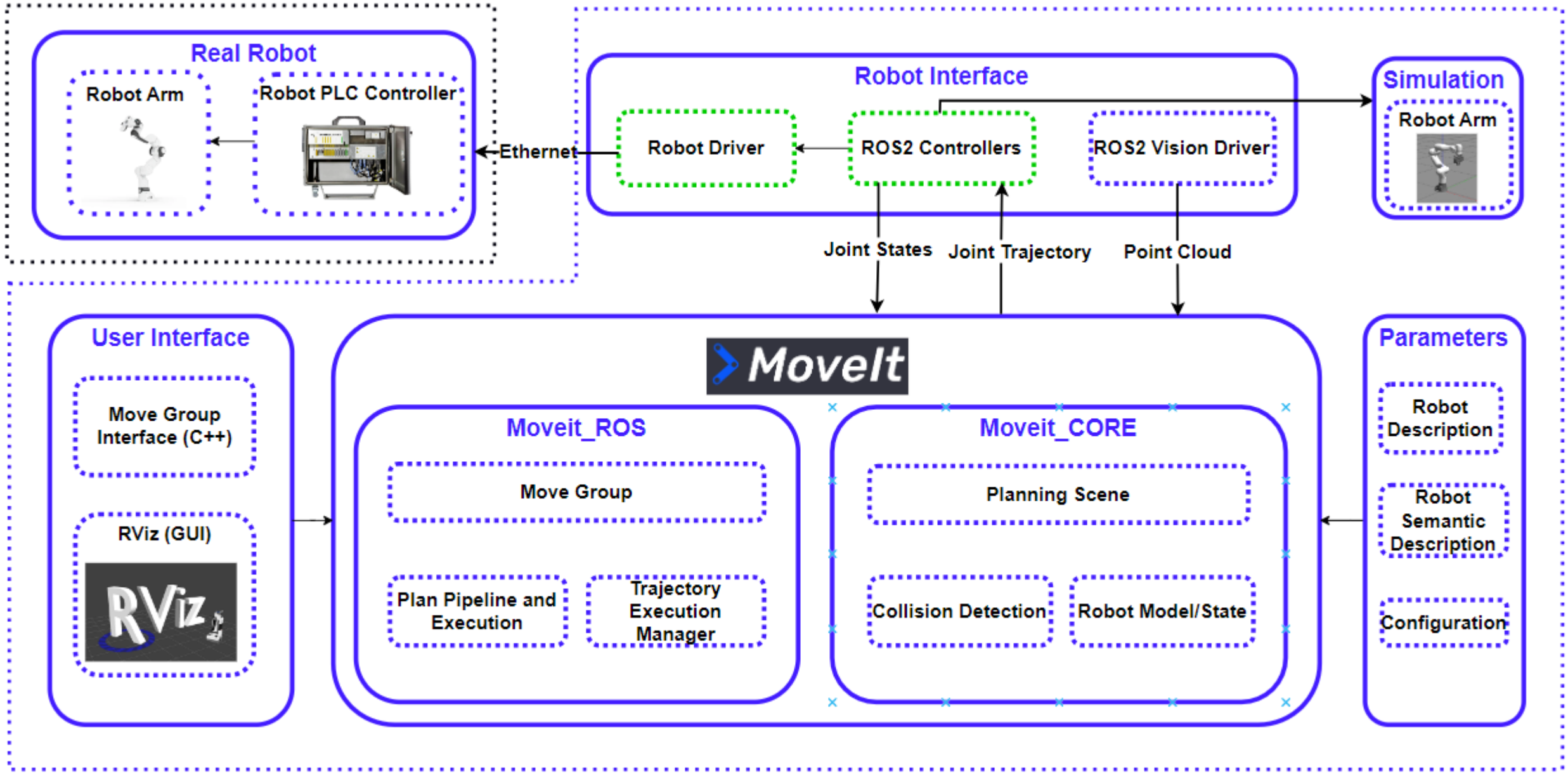
ROS2 Manipulation Course | Overall Scope



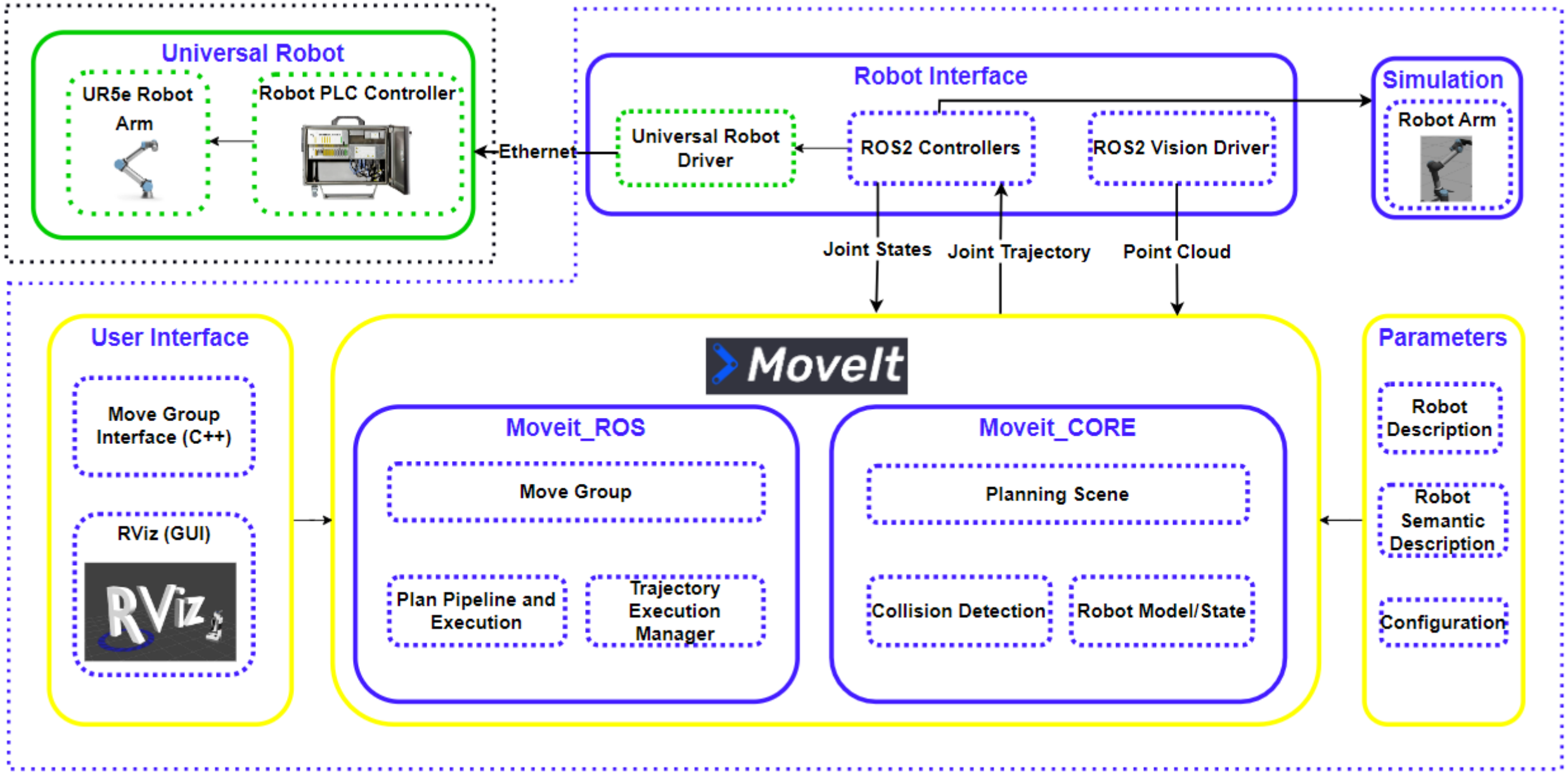
ROS2 Manipulation Course | Day 1 Scope



ROS2 Manipulation Course | Day 2 Scope



ROS2 Manipulation Course | Day 3 Scope



ROS-Industrial Manipulation Training - Agenda Day 1

- What is Manipulation?
  - Concepts
  - Applications
- Robot Description
  - URDF
    - Package Content
    - URDF Components
    - Exercise 1: Visualize Robot Description by adding table and camera
- Moveit Setup Assistant
    - Setup Components
    - ROS2 Controllers [High-Level]
    - Moveit Controllers [High-Level]
    - Exercise 2: Create Moveit Config and Launch
- Move Group
    - Move Group Capabilities
    - Exercise 3: Motion Planning with Move Group
- Perception with Camera and ARUCO Marker
    - Xacro [High-Level]
    - Exercise 4: ARUCO Detection and Spawn Collision Object
- Perception with Gazebo Simulation [Take Home (optional)]
    - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo

ROS-Industrial Manipulation Training - Agenda Day 1

- What is Manipulation?
  - Concepts
  - Applications
- Robot Description
  - URDF
    - Package Content
    - URDF Components
    - Exercise 1: Visualize Robot Description by adding table and camera
- Moveit Setup Assistant
    - Setup Components
    - ROS2 Controllers [High-Level]
    - Moveit Controllers [High-Level]
    - Exercise 2: Create Moveit Config and Launch
- Move Group
    - Move Group Capabilities
    - Exercise 3: Motion Planning with Move Group
- Perception with Camera and ARUCO Marker
    - Xacro [High-Level]
    - Exercise 4: ARUCO Detection and Spawn Collision Object
- Perception with Gazebo Simulation [Take Home (optional)]
    - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo

Advanced Remanufacturing and Technology Centre

ARTC

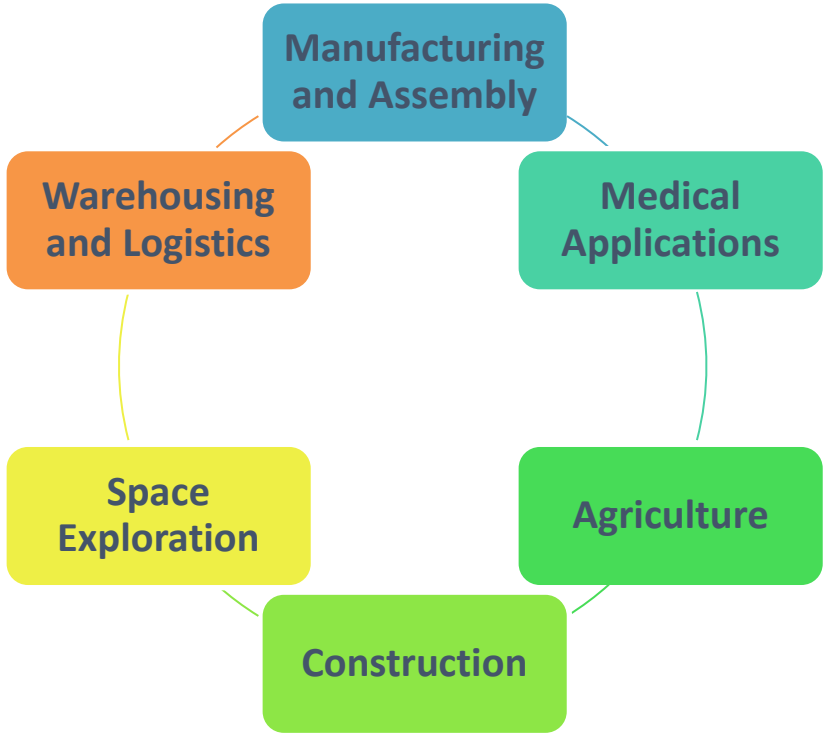
WHAT IS MANIPULATION?

Shalman Khan

4th December 2023

What is Manipulation?

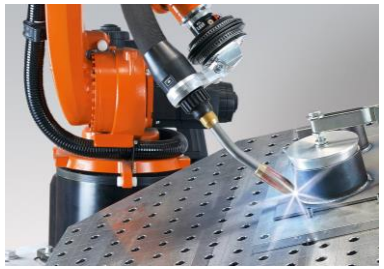
Fields of Manipulation?



Day 4 Final Assessment with UR robot

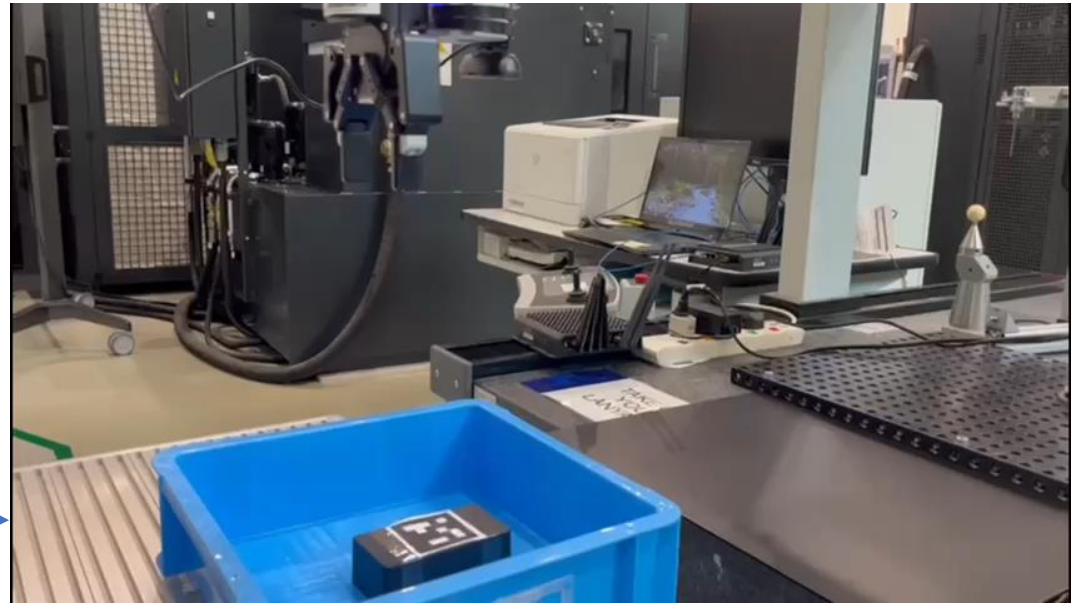
Applications of Manipulation?

Disassembly



Welding

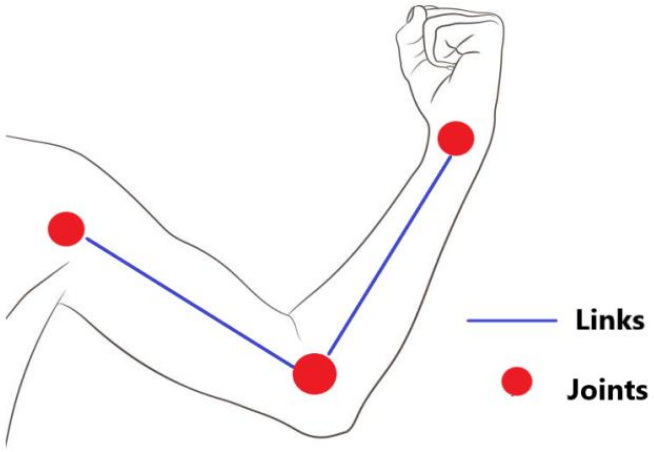
Painting

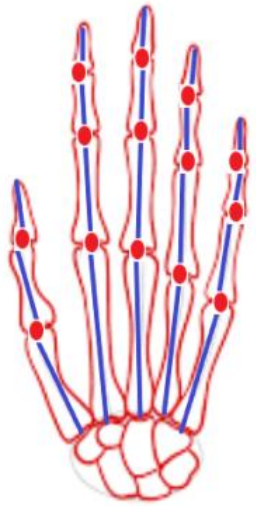


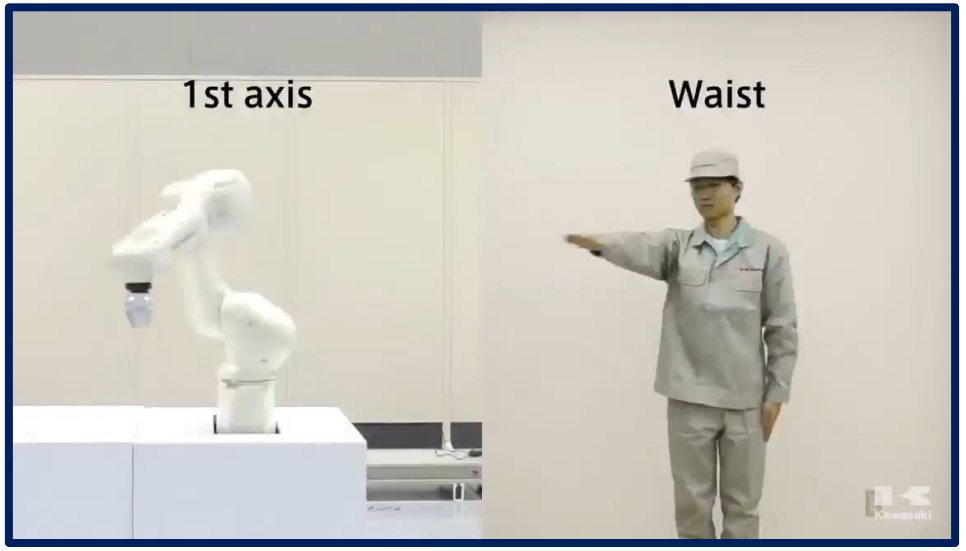
Pick and Place

Image Source: ABB, Kuka,

What is Manipulation?







Manipulation Concept:

Robot manipulation and the human arm share several similarities in their basic principles and movements.

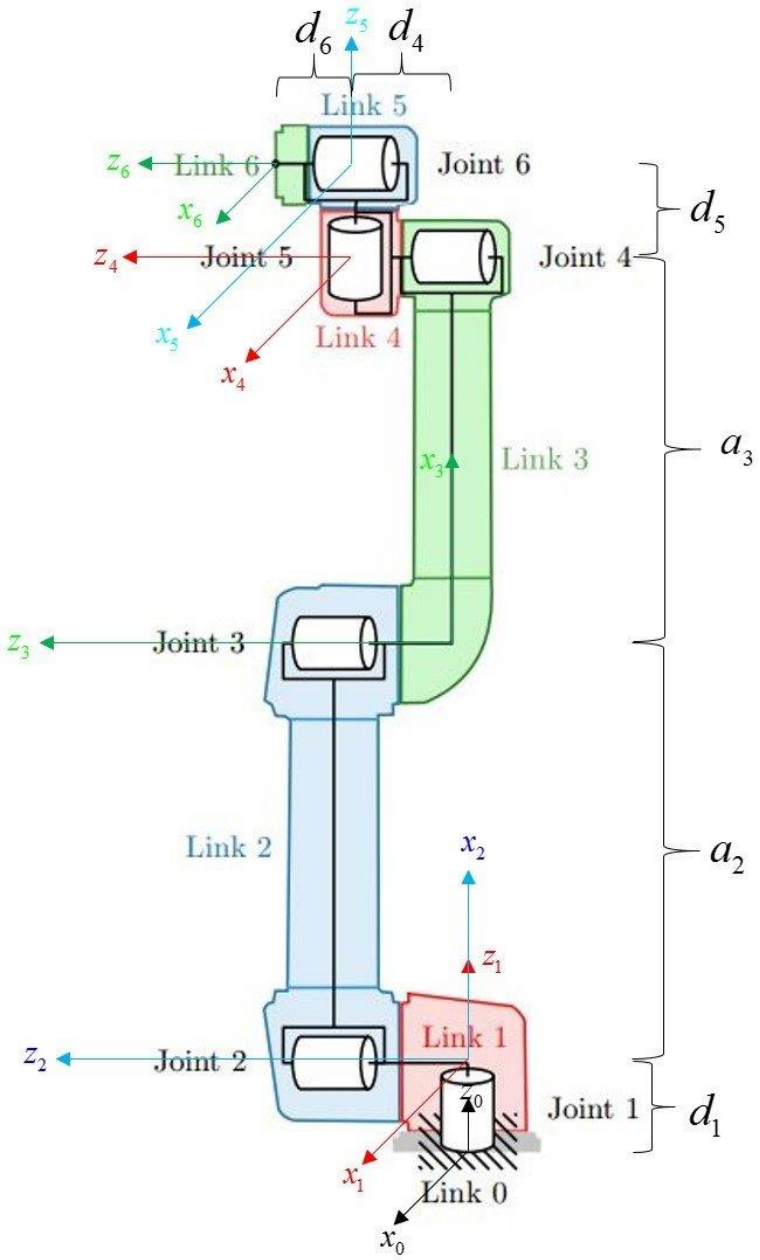
Kinematic Chains

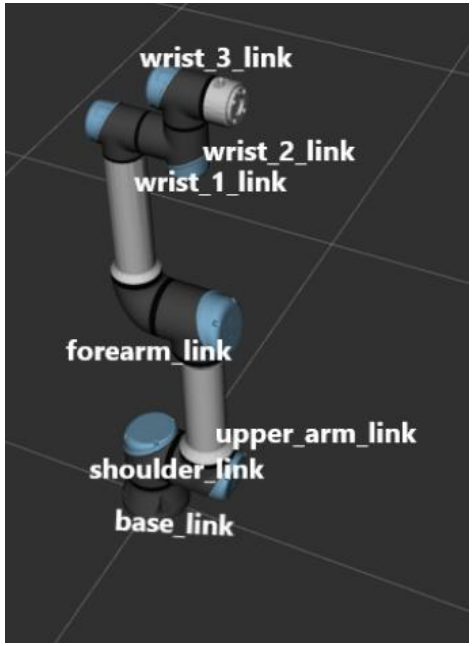
Degrees of Freedom (DOF)

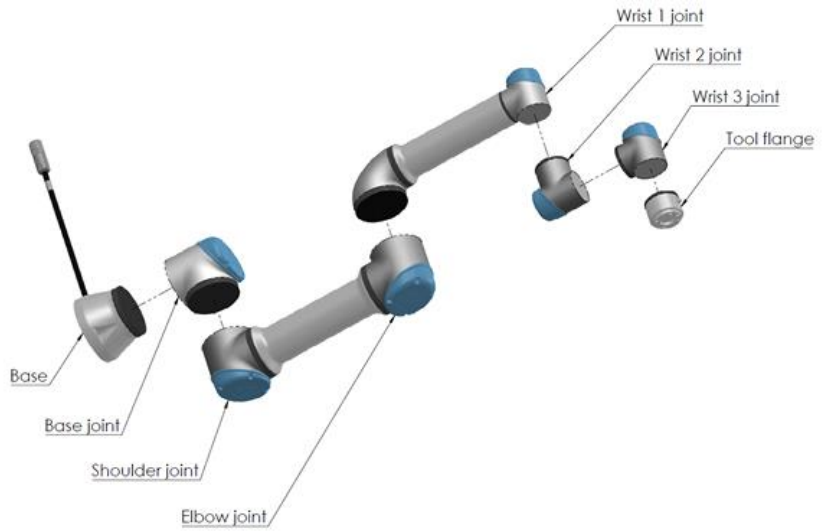
End Effector

Video Source: Kawasaki Robotics

Kinematics Chain – Links and Joints



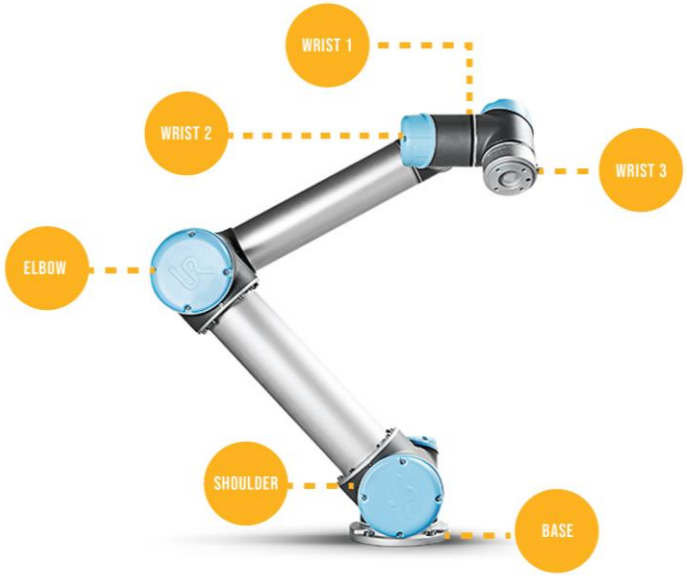




Links and Joints

- Links: Links in robotics are rigid structural elements that connect joints. Links can be simple, straight pieces or more complex, multi-shaped segments.

- Joints: Joints are the movable components that connect links in a robot. They enable relative motion between the connected links, allowing the robot to bend, rotate, or move in various ways.

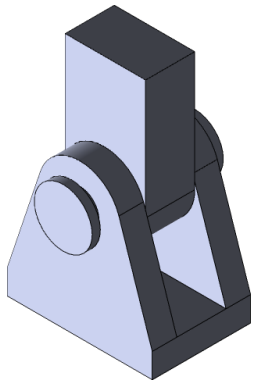


Kinematics Chain – Joints and Degrees of Freedom

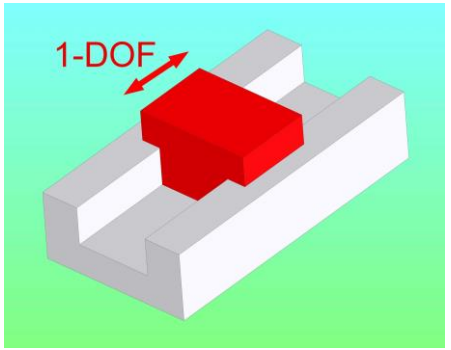
- **Degrees of Freedom** indicate how many ways a joint or a combination of joints can move.

DOF = 3*(links−1)−2*Joints−higher_pairs

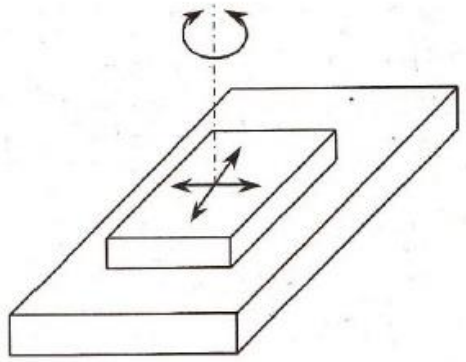
- **Types of Joints:** There are many types of joints. Here's a refined description of some significant joints used in robotics manipulation:
  - Revolute Joint
  - Prismatic Joint
  - Continuous Joint
  - Planar Joint
  - Fixed Joint



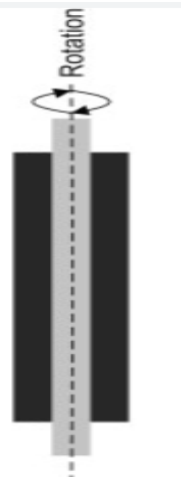
Revolute Joint (1 DOF)



Prismatic Joint (1 DOF)



Planar Joint (2 DOF)



Continuous Joint (1 DOF)

ROS-Industrial Manipulation Training - Agenda Day 1

What is Manipulation?

- Concepts
- Applications

Robot Description

- URDF
  - Package Content
  - URDF Components
  - Exercise 1: Visualize Robot Description by adding table and camera

Moveit Setup Assistant

- Setup Components
- ROS2 Controllers [High-Level]
- Moveit Controllers [High-Level]
- Exercise 2: Create Moveit Config and Launch

Move Group

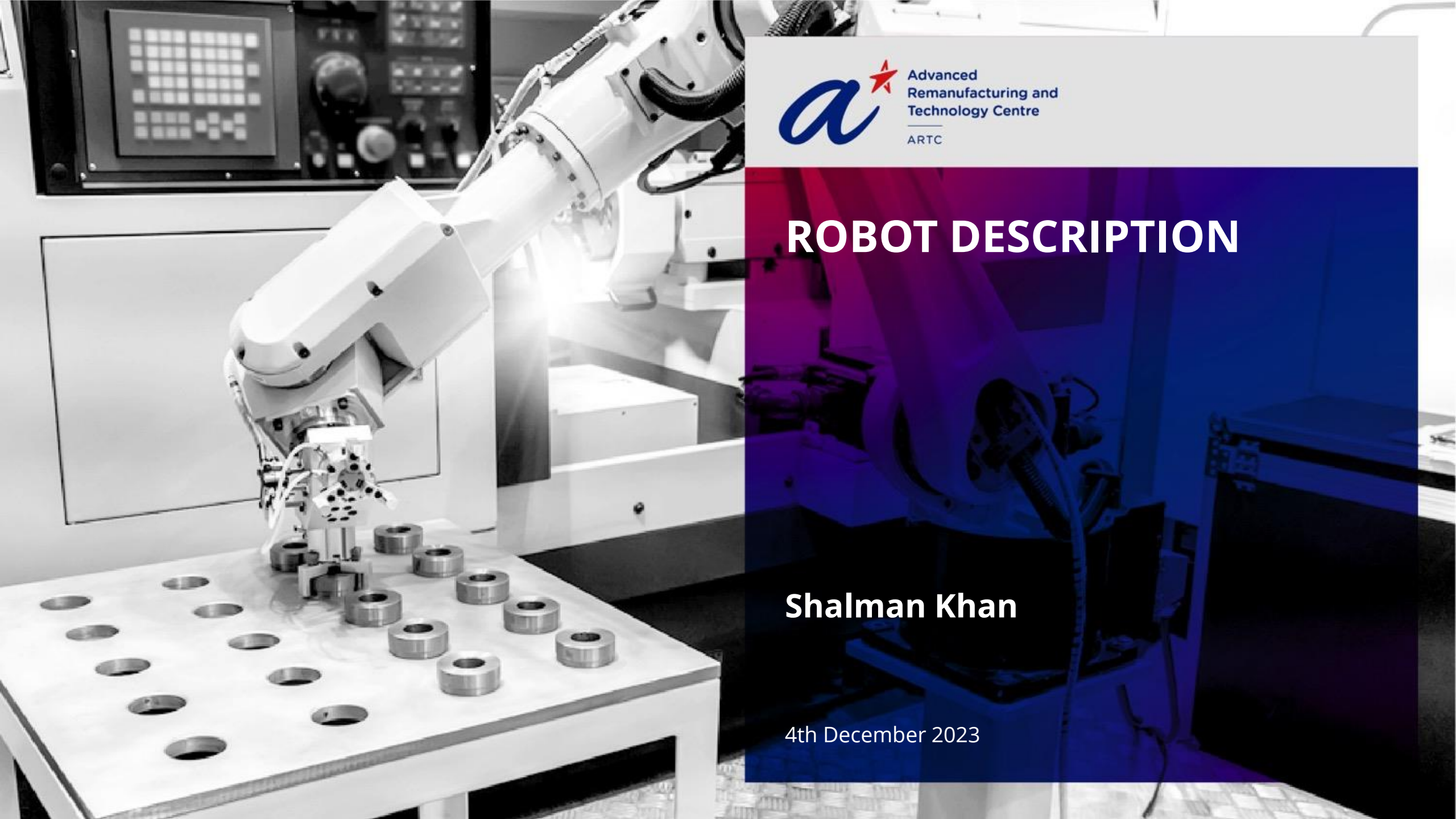
- Move Group Capabilities
- Exercise 3: Motion Planning with Move Group

Perception with Camera and ARUCO Marker

- Xacro [High-Level]
- Exercise 4: ARUCO Detection and Spawn Collision Object

Perception with Gazebo Simulation [Take Home (optional)]

- Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo



Advanced Remanufacturing and Technology Centre

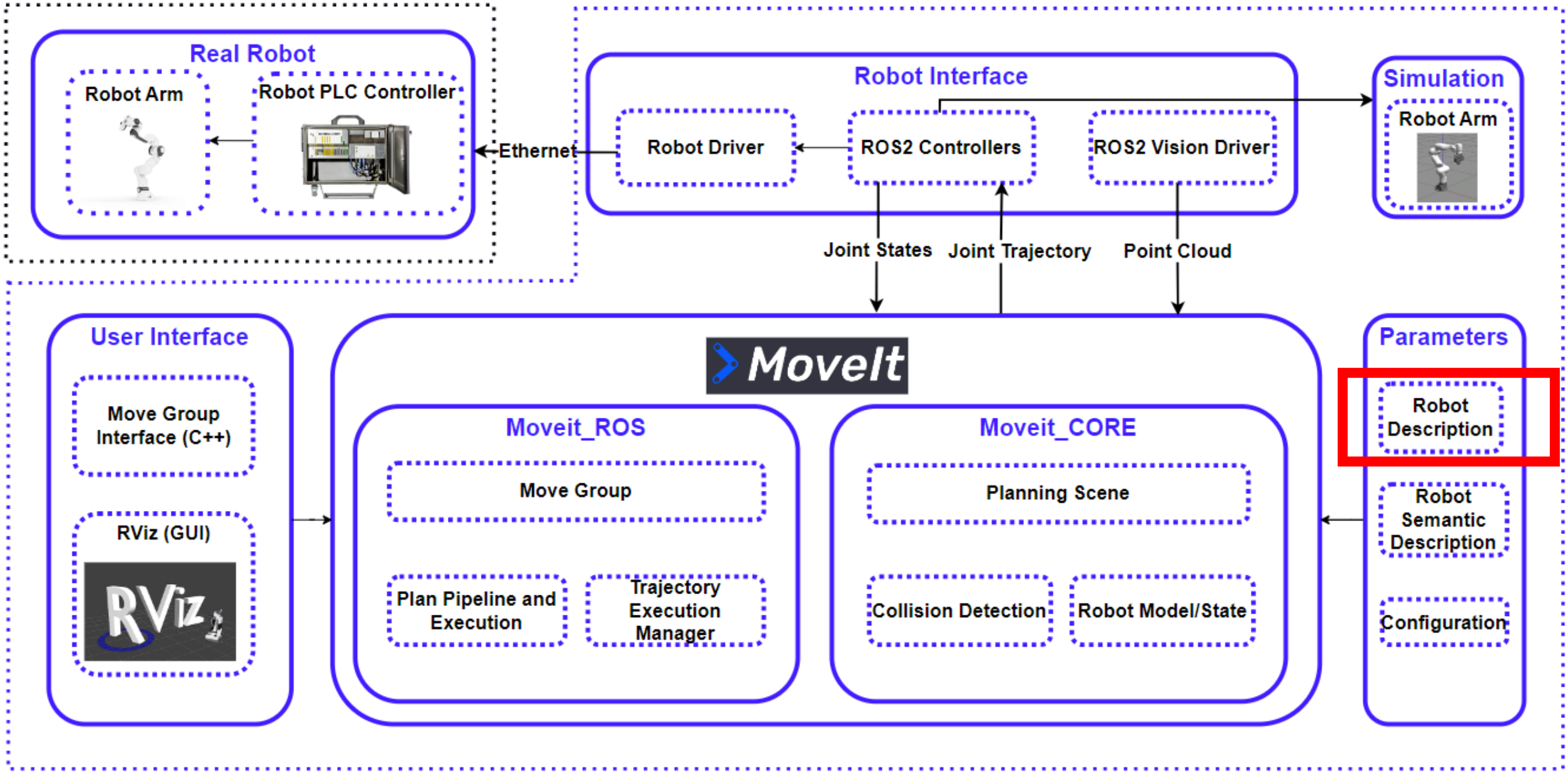
ARTC

ROBOT DESCRIPTION

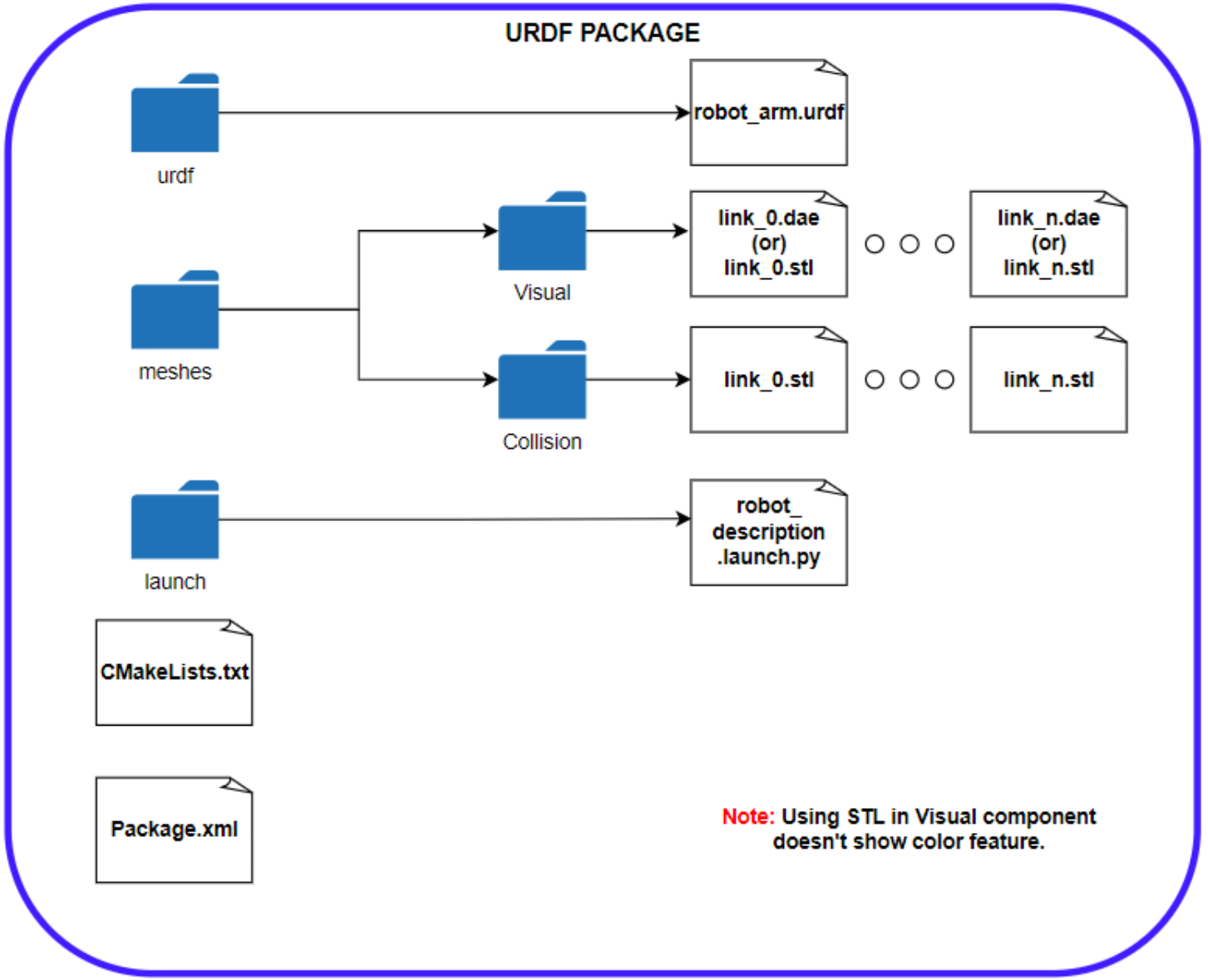
Shalman Khan

4th December 2023

Robot Description

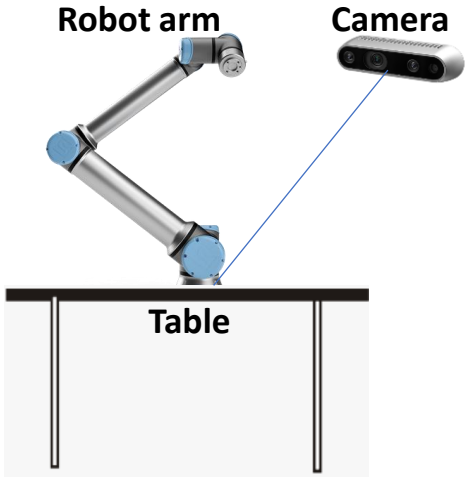


Robot Description

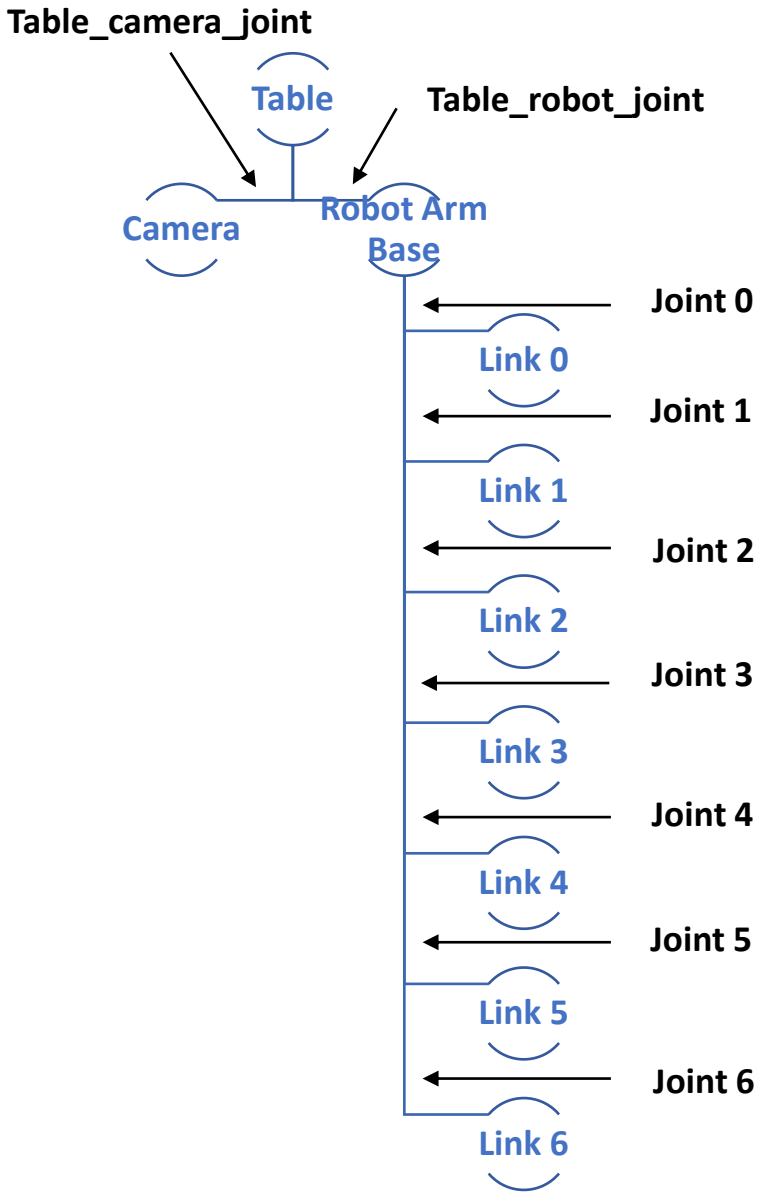


Universal Robot Description Format (URDF)

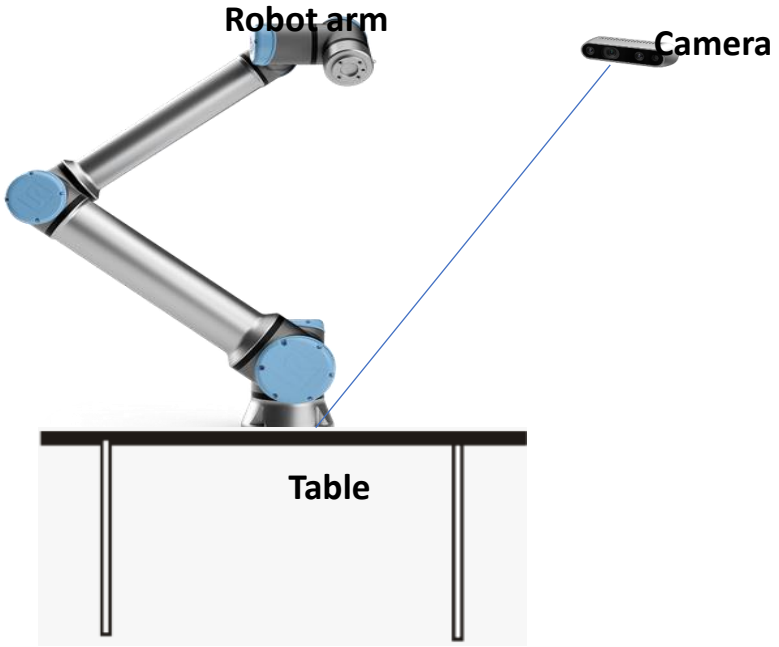
- URDF is an XML-based file format used in the field of robotics to describe the robot based on
  - Links
    - Visual Component
    - Collision Component
    - Inertial Component
  - Joints
  - Sensors

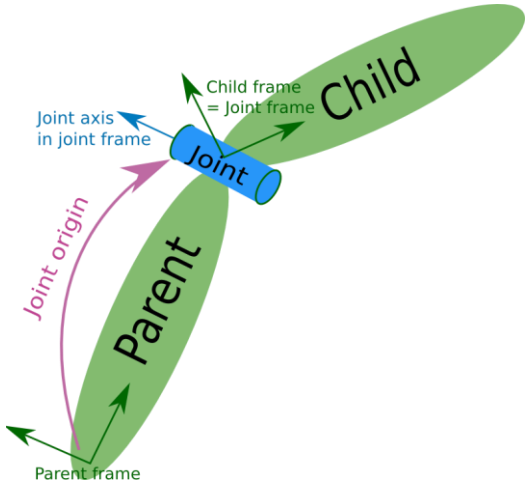


Robot Description



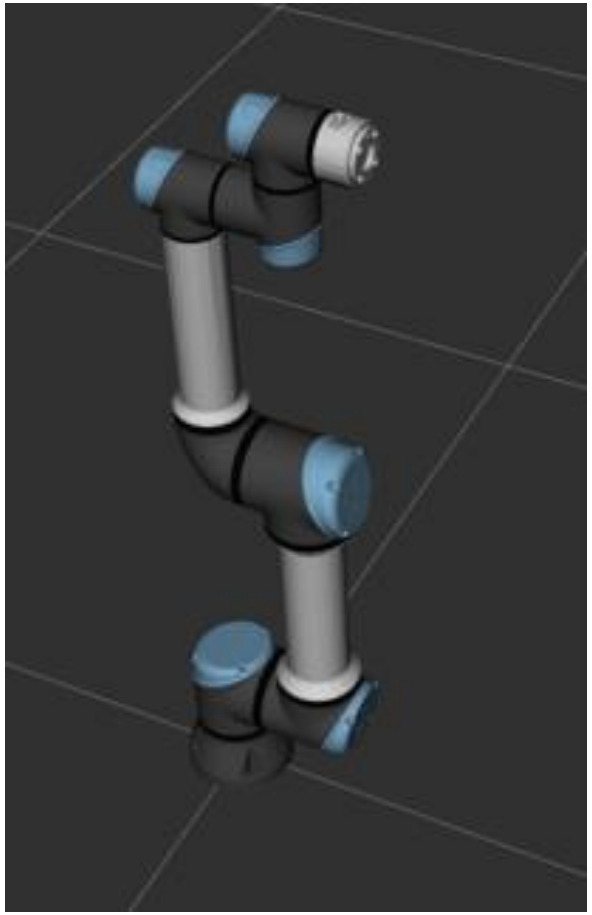

```xml
<?xml version="1.0"?>
<robot name="ur_robot">

  <!-- Link Definition for table -->
  <link name="table">
  <!-- Visual Component to be added -->
  <!-- Collision Component to be added -->
  </link>

  <!-- Link Definition for table -->
  <link name="camera">
  <!-- Visual Component to be added -->
  <!-- Collision Component to be added -->
  </link>

  <!-- Link Definition for table -->
  <link name="robot_arm_base">
  <!-- Visual Component to be added -->
  <!-- Collision Component to be added -->
  </link>

  <!-- Joint Connecting Table and Camera -->
  <joint name="table_camera_joint" type="fixed">
    <origin xyz="1 1 2.5" rpy="0 0 0"/>
    <parent link="table"/>
    <child link="camera"/>
  </joint>

  <!-- Joint Connecting Table and Camera -->
  <joint name="table_robot_joint" type="fixed">
    <origin xyz="0 0 1.5" rpy="0 0 0"/>
    <parent link="table"/>
    <child link="robot_arm_base"/>
  </joint>

</robot>
```



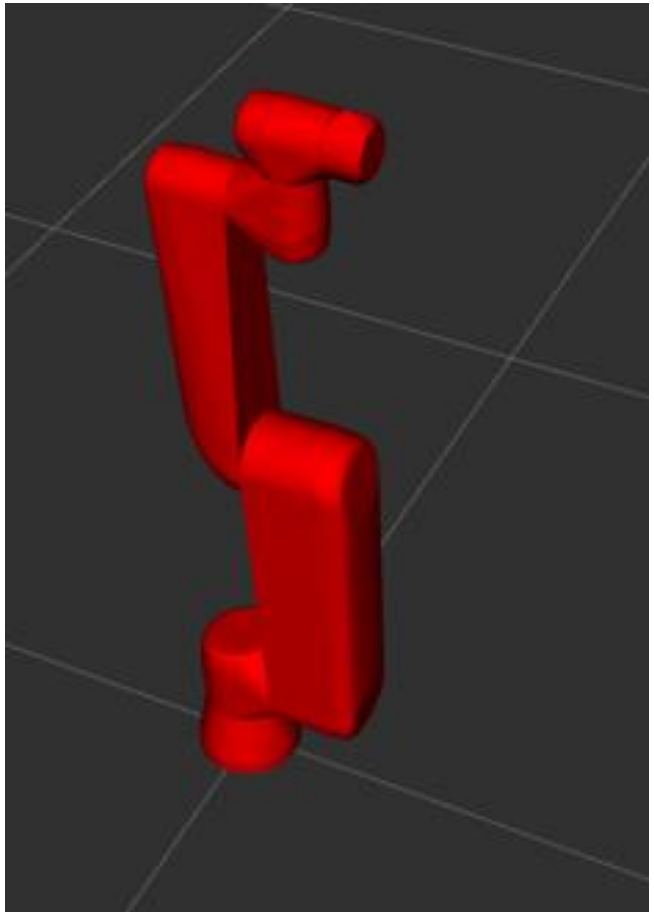


Tree Image source: http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file

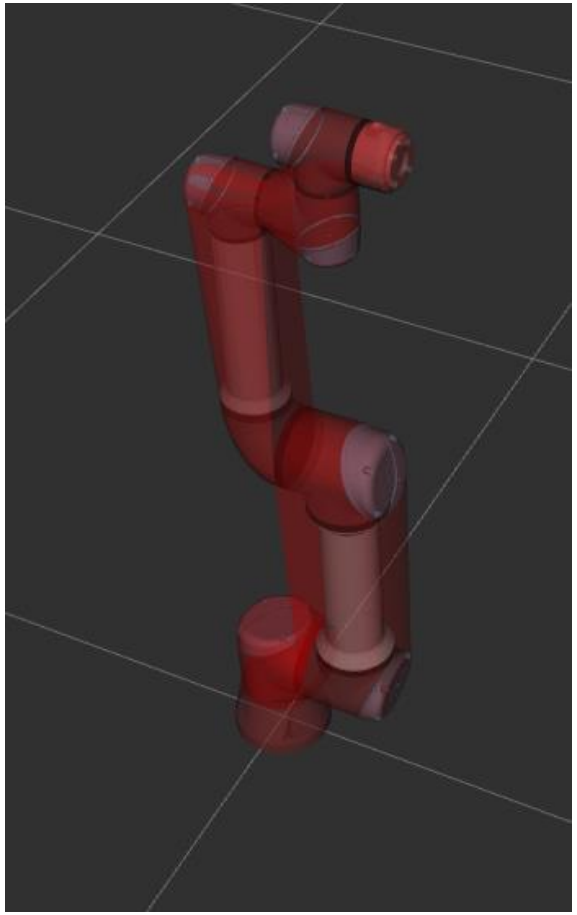
Robot Description – Visual Component and Collision Component



Visual Component



Collision Component



Visual + Collision Component

Robot Description – Visual Component and Collision Component

```
<!-- Link Definition for Link_0 -->
<link name="Link_0">
<!-- Visual Component to be added -->
<!-- Collision Component to be added -->
</link>
```

Box

```
<!-- Link Definition for link_0 -->
<link name="Link_0">
  <!-- Visual representation of the link -->
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </visual>
  <!-- Collision representation of the link -->
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>
</link>
```

Cylinder

```
<!-- Link Definition for link_0 -->
<link name="Link_0">
  <!-- Visual representation of the link -->
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="0.05" length="0.1"/>
    </geometry>
  </visual>
  <!-- Collision representation of the link -->
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="0.05" length="0.1"/>
    </geometry>
  </collision>
</link>
```

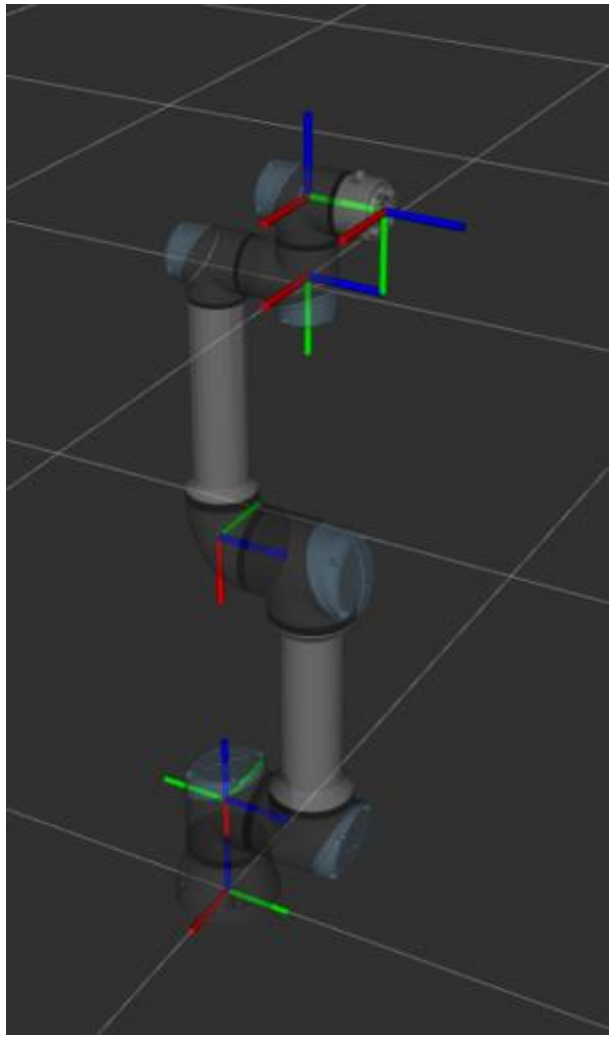
mesh

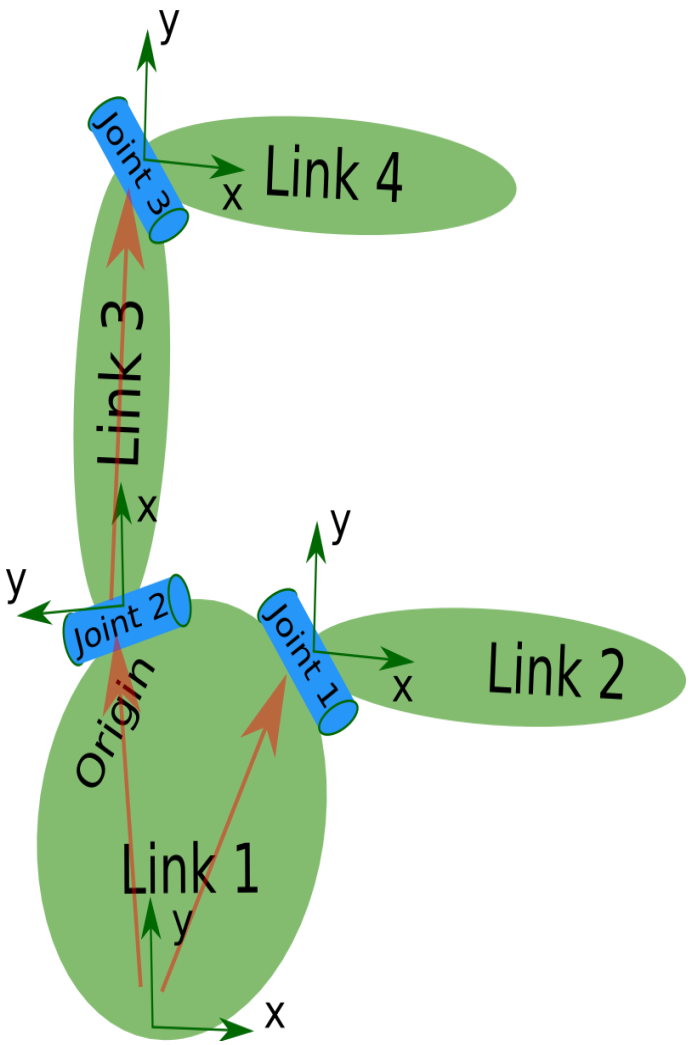
```
<!-- Link Definition for link_0 -->
<link name="Link_0">
  <!-- Visual representation of the link using a mesh -->
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="path/to/your/mesh_file.dae"/>
    </geometry>
  </visual>
  <!-- Collision representation of the link using a mesh -->
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="path/to/your/mesh_file.stl"/>
    </geometry>
  </collision>
</link>
```

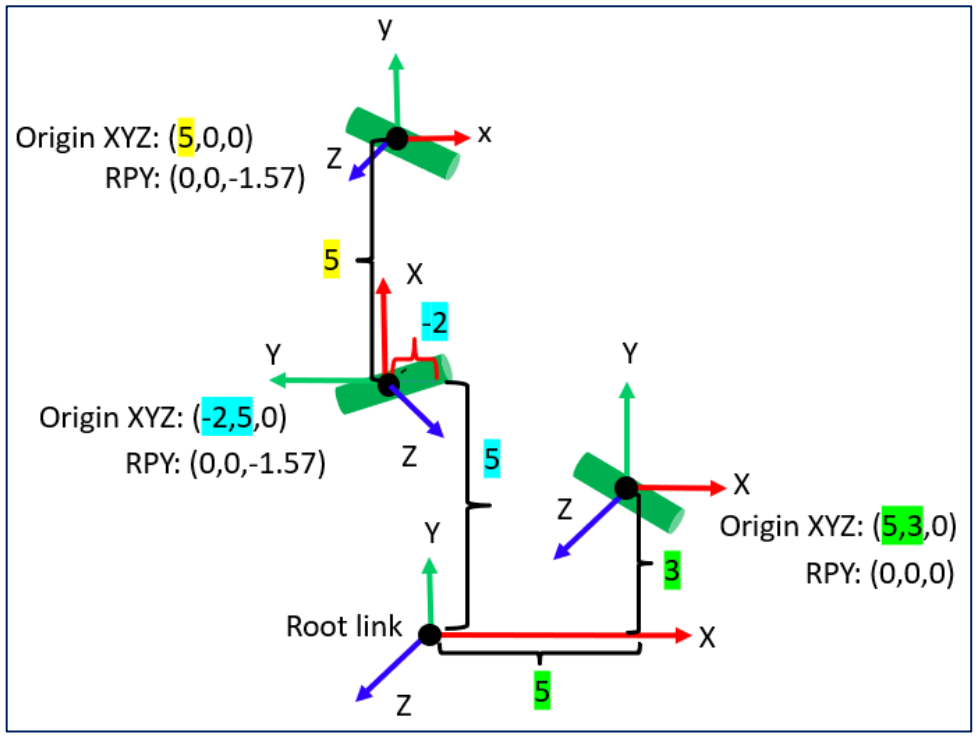
Visual Component and Collision Component

- Both Visual Component and Collision Component comprises of **Geometry** to define the shape.
- A shape can be standard as box (or) cylinder (or) custom meshes in .dae or .stl format.

Robot Description – Joint Origin and Axis





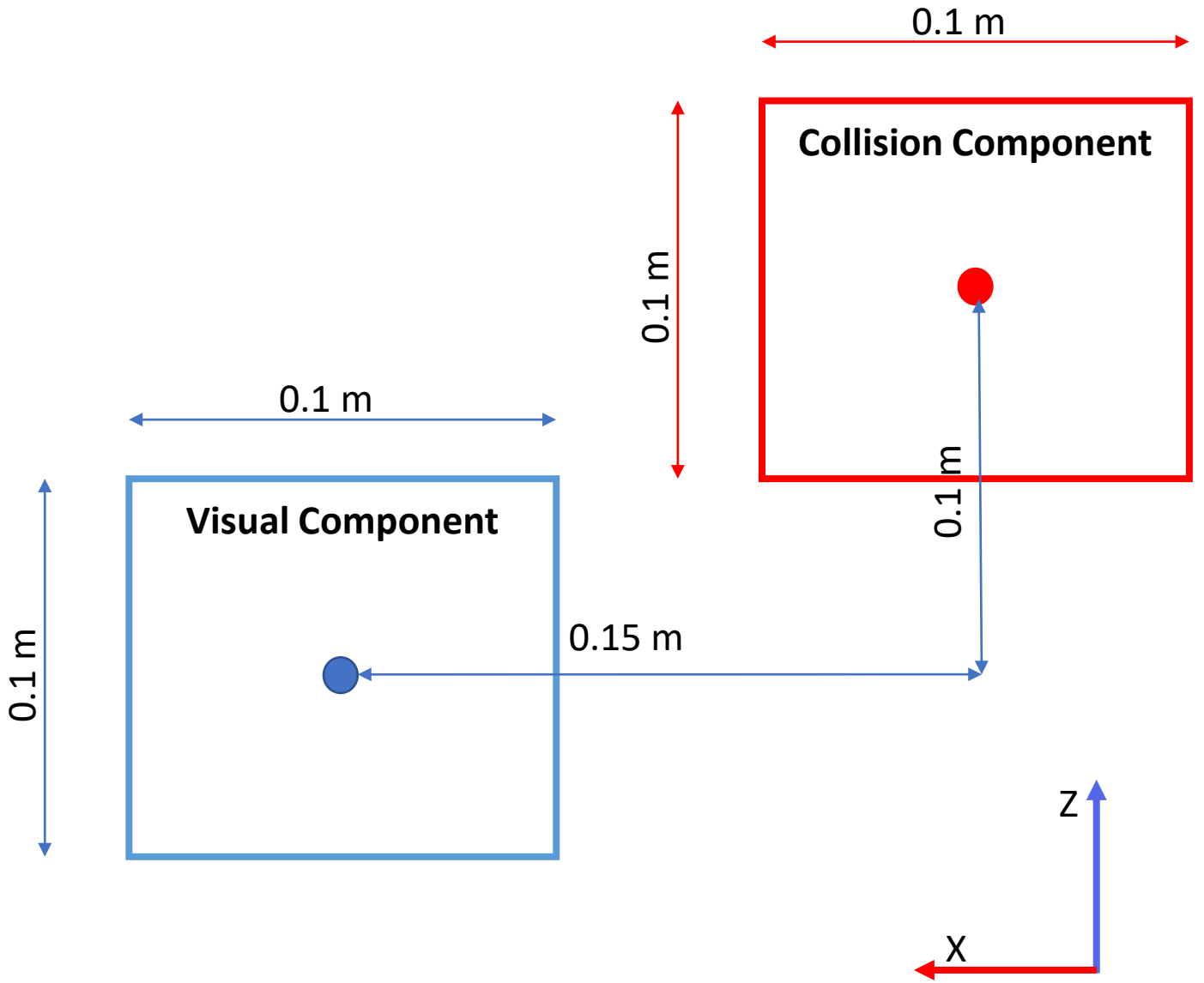



```
<joint name="joint1" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="5 3 0" rpy="0 0 0" />
  <axis xyz="-0.9 0.15 0" />
</joint>
```

Tree Image source: http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file

Robot Description – Link Geometry Origin

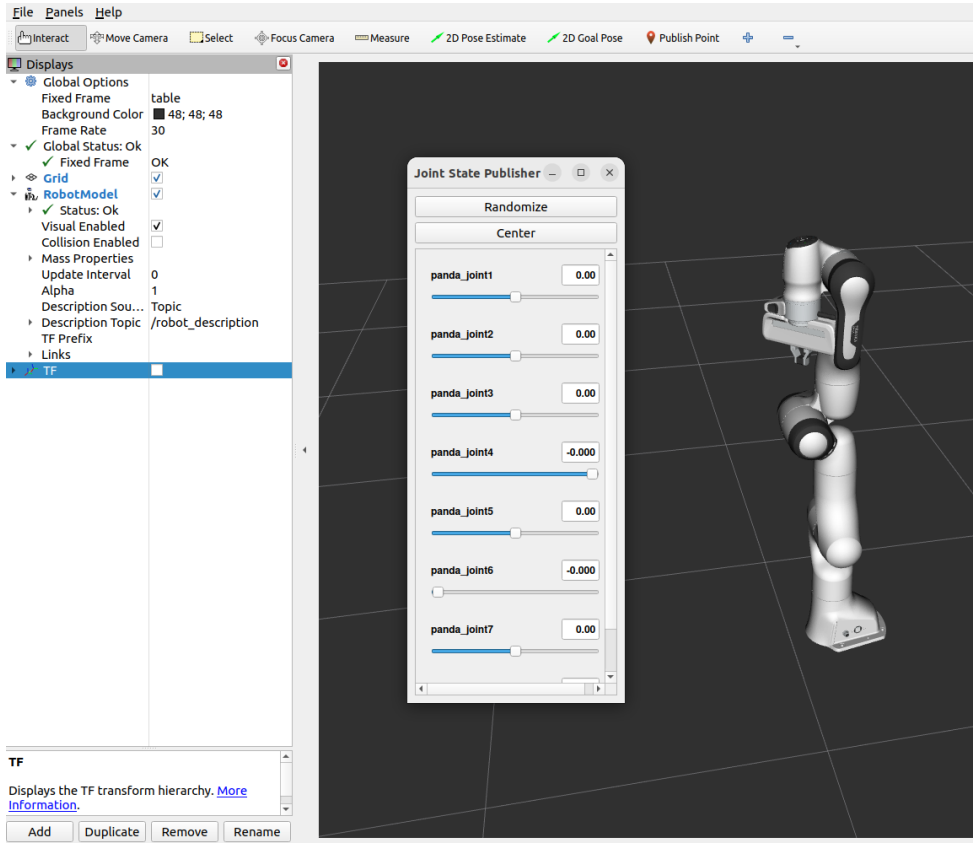
```
<!-- Link Definition for link_0 -->
<link name="Link_0">
  <!-- Visual representation of the link -->
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </visual>
  <!-- Collision representation of the link -->
  <collision>
    <origin xyz="0.15 0 0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>
</link>
```

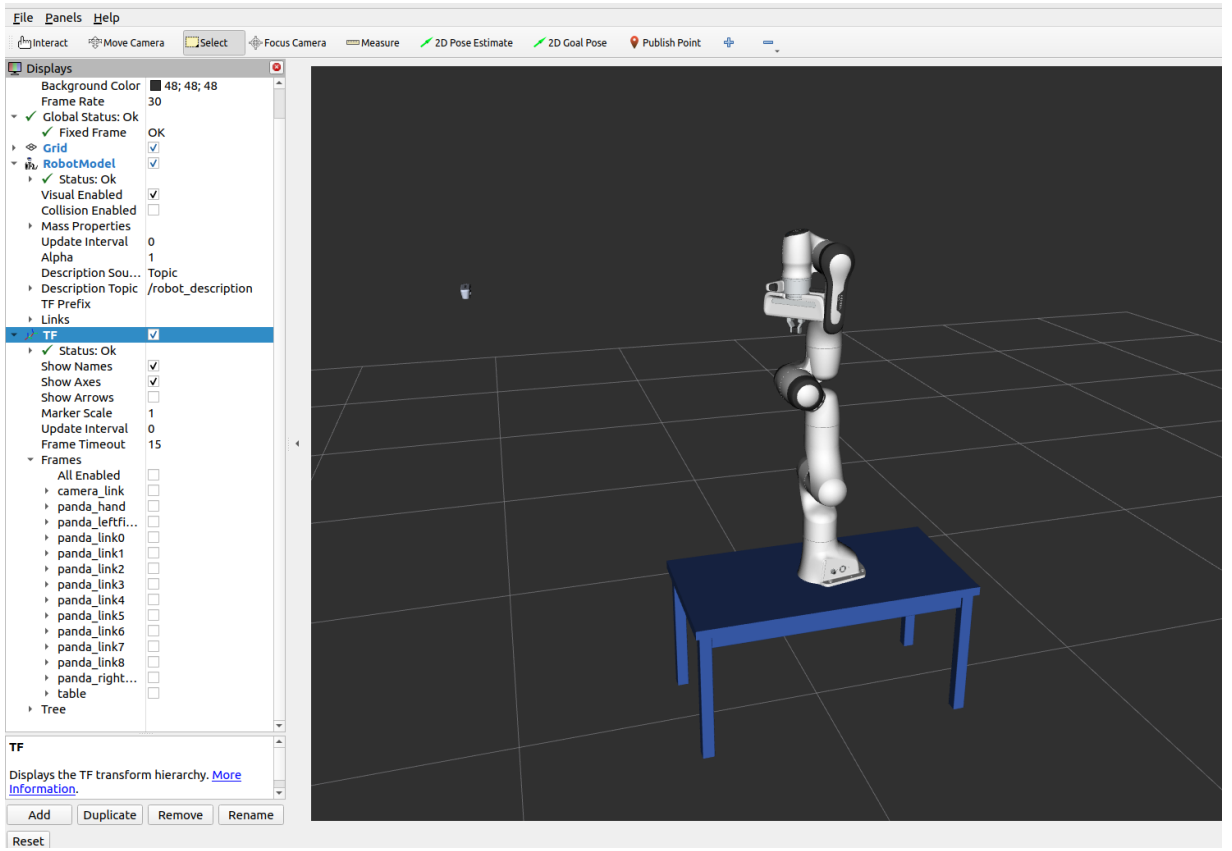


Exercise 1: Visualize Robot Description by adding table and camera

There are two objectives to this exercise

1. **Add the right joint for the gripper and visualize the robot arm in RViz**
2. **Add a table, camera and visualize in RViz**





Safety_limits

ROS-Industrial Manipulation Training - Agenda Day 1

What is Manipulation?
- Concepts
- Applications

Robot Description
- URDF
  - Package Content
  - URDF Components
  - Exercise 1: Visualize Robot Description by adding table and camera

Moveit Setup Assistant
  - Setup Components
  - ROS2 Controllers [High-Level]
  - Moveit Controllers [High-Level]
  - Exercise 2: Create Moveit Config and Launch

Move Group
  - Move Group Capabilities
  - Exercise 3: Motion Planning with Move Group

Perception with Camera and ARUCO Marker
  - Xacro [High-Level]
  - Exercise 4: ARUCO Detection and Spawn Collision Object

Perception with Gazebo Simulation [Take Home (optional)]
  - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo

Advanced Remanufacturing and Technology Centre

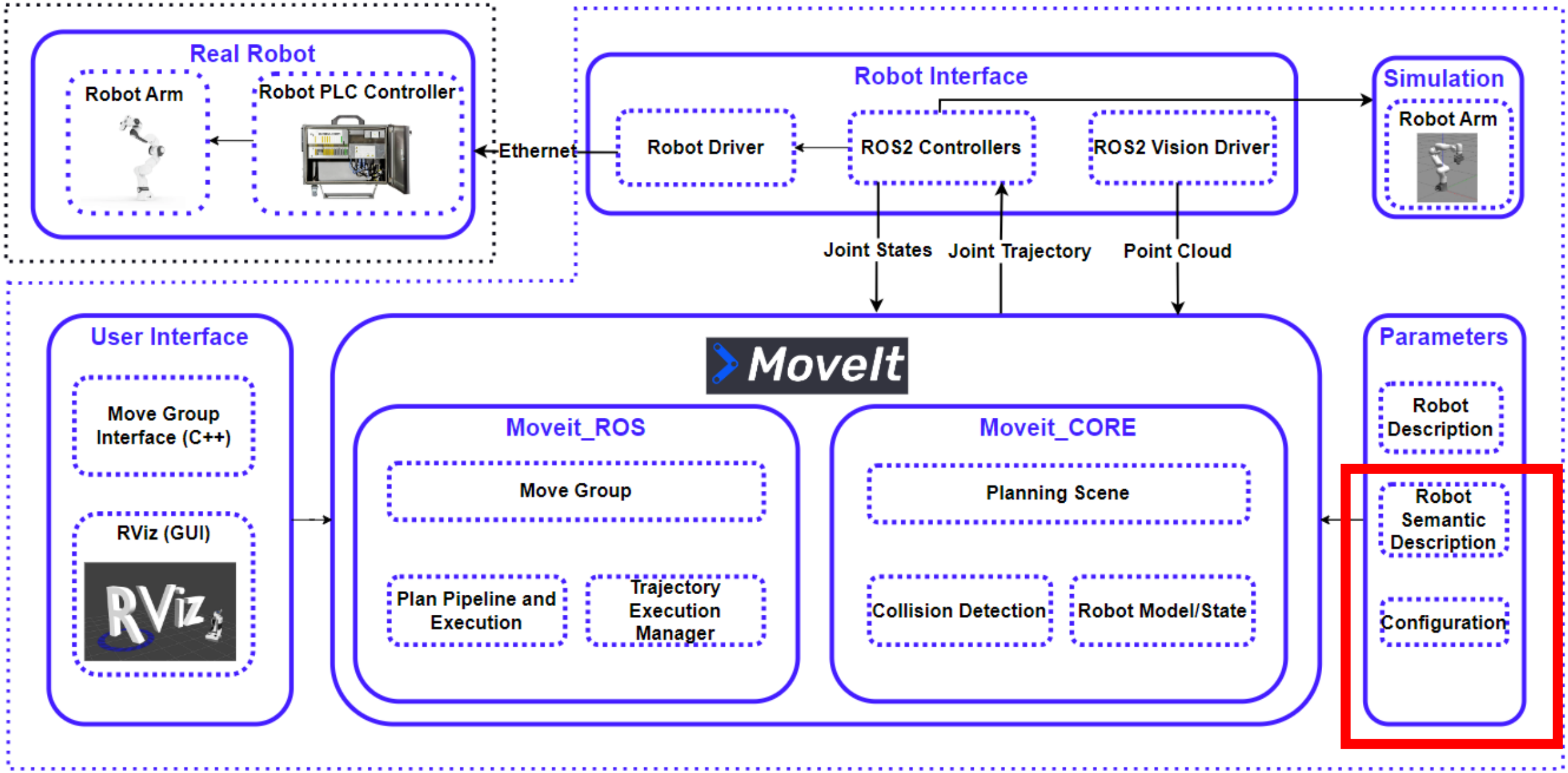
ARTC

MOVEIT SETUP ASSISTANT

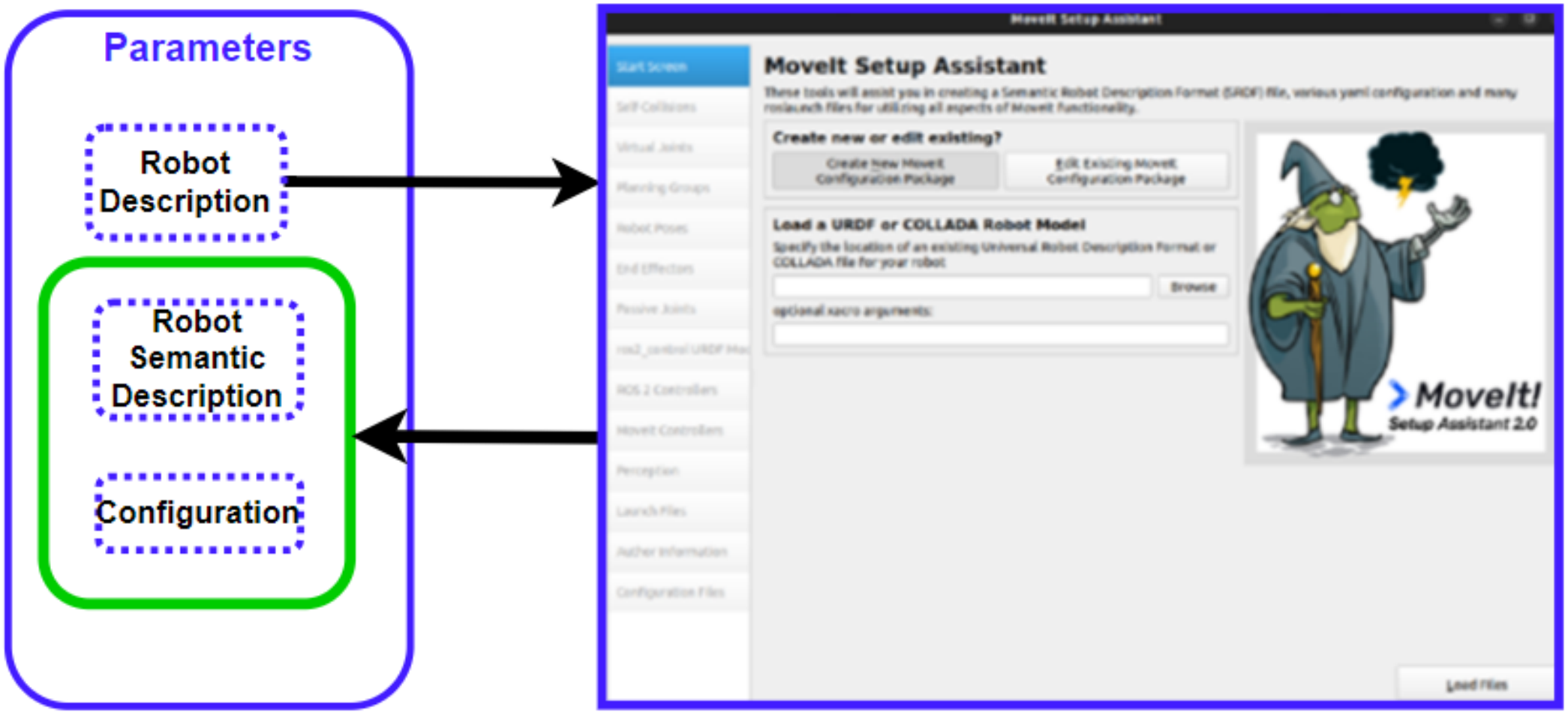
Shalman Khan

4th December 2023

Moveit Setup Assistant

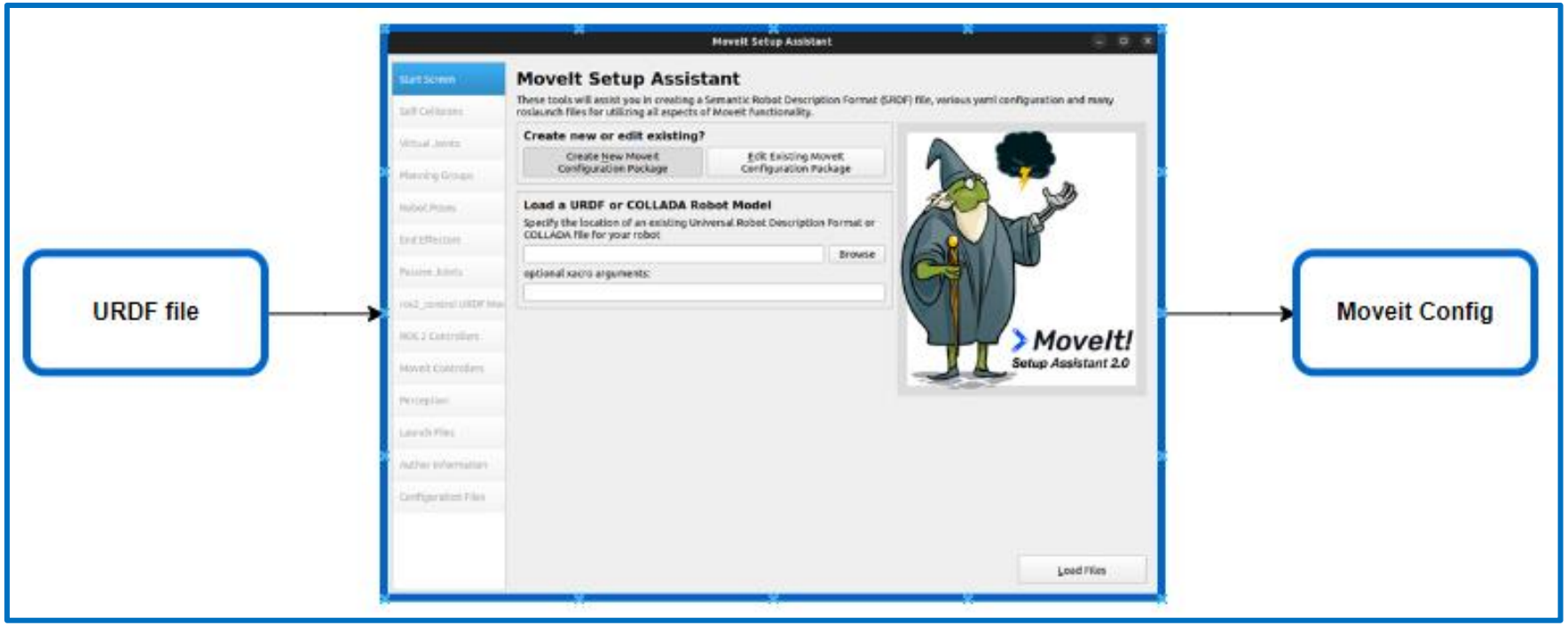


Moveit Setup Assistant – Setup Components



ros2 launch moveit_setup_assistant setup_assistant.launch.py

Moveit Setup Assistant – Setup Components

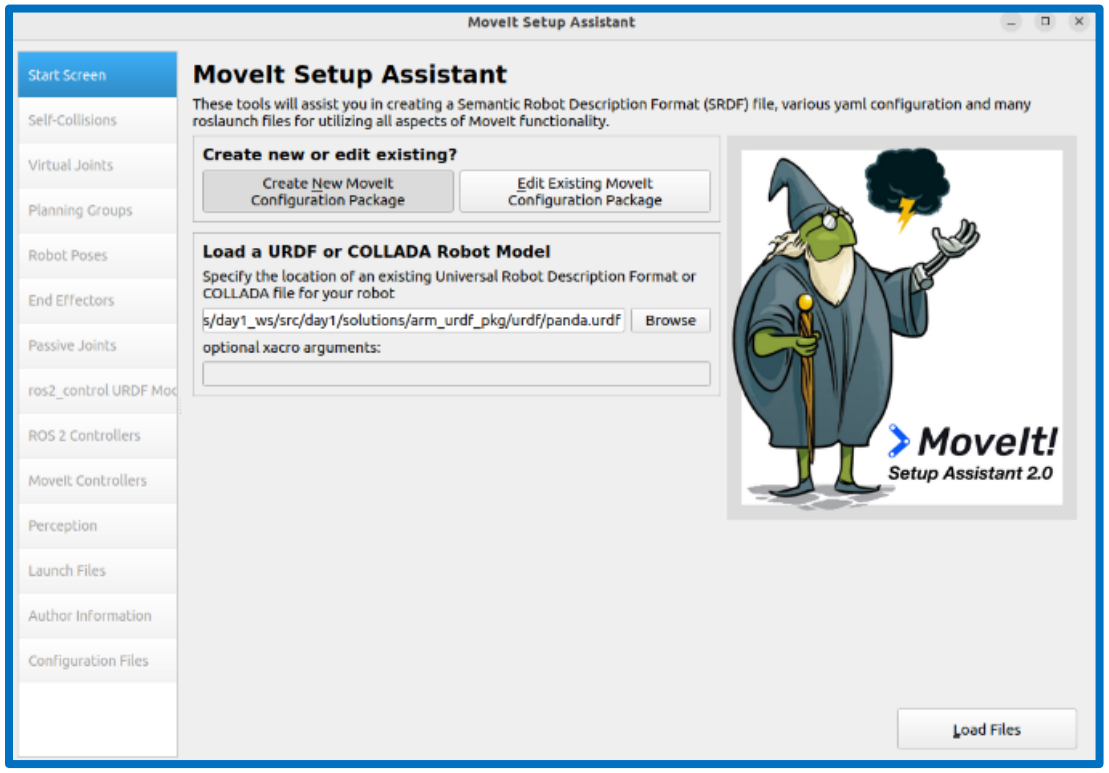


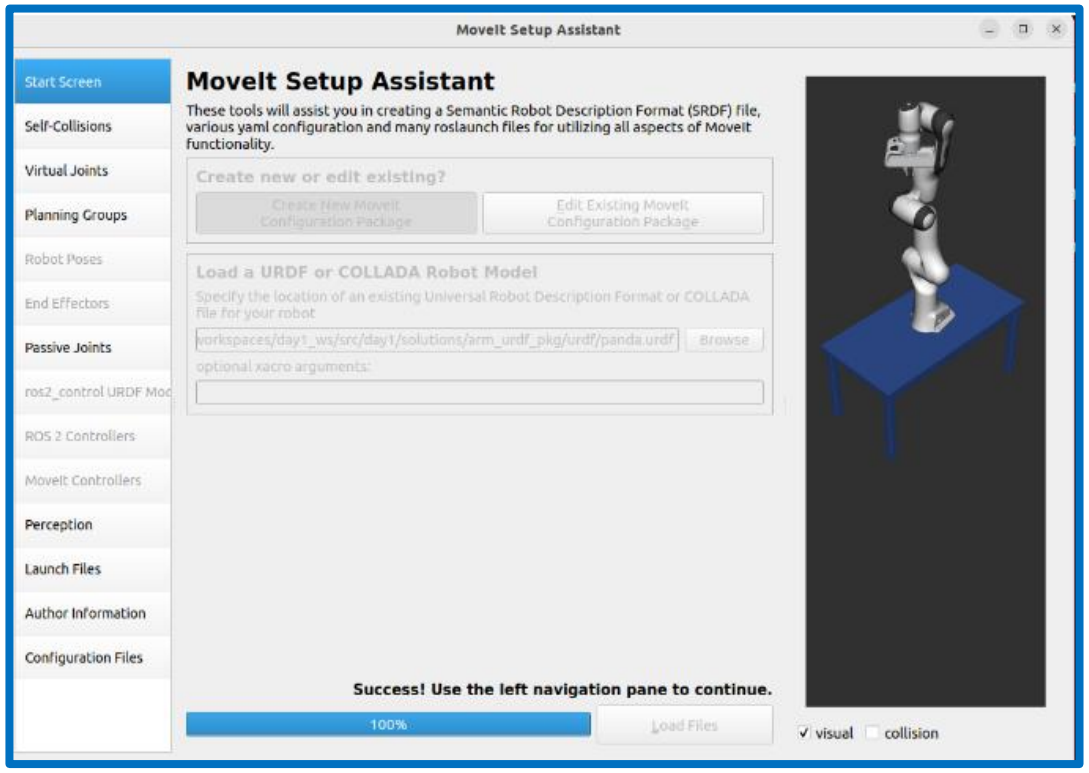
<u>Moveit Setup Assistant</u>

- The **MoveIt Setup Assistant (ROS2 Humble)** serves as a **graphical** interface for the seamless development of **MoveIt configurations for robotic arms.**
- This tool requires only one input: the **URDF** (Universal Robot Description Format) file that defines the specific characteristics of the robot arm.
- The MoveIt Setup Assistant (ROS2 Humble) generates a **Semantic Robot Description Format (SRDF)** file, specifying vital information for MoveIt, such as planning groups and kinematic parameters

ros2 launch moveit_setup_assistant setup_assistant.launch.py

Moveit Setup Assistant – URDF Import

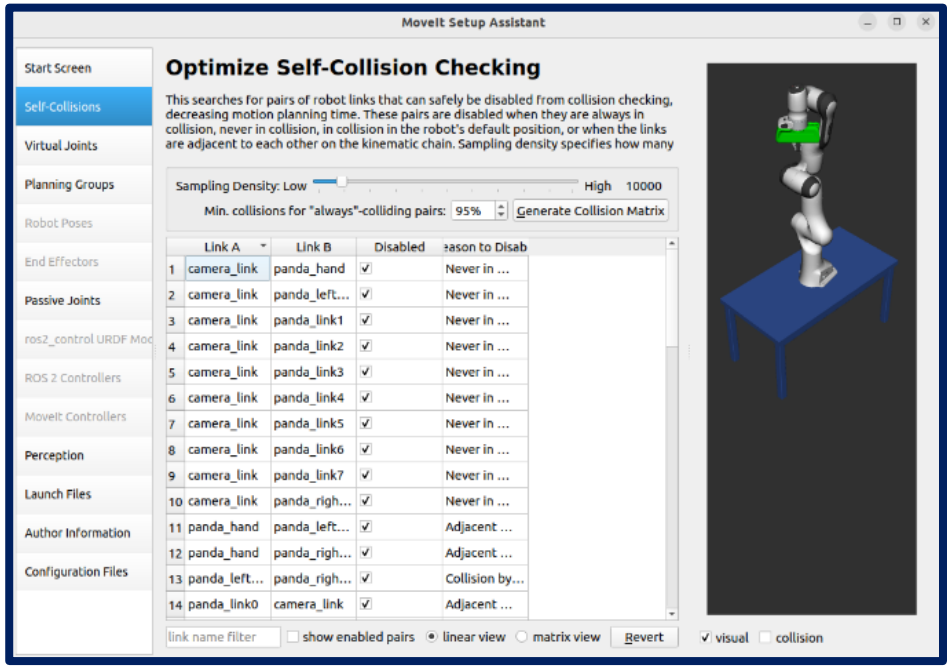




Moveit Setup Assistant Input

- Setup Assistant extracts robot arm specific information from **URDF** (Universal Robot Description Format) file.
- Setup Assistant let's user to create a **new moveit configuration package** with URDF or **edit** an already created moveit configuration packages.

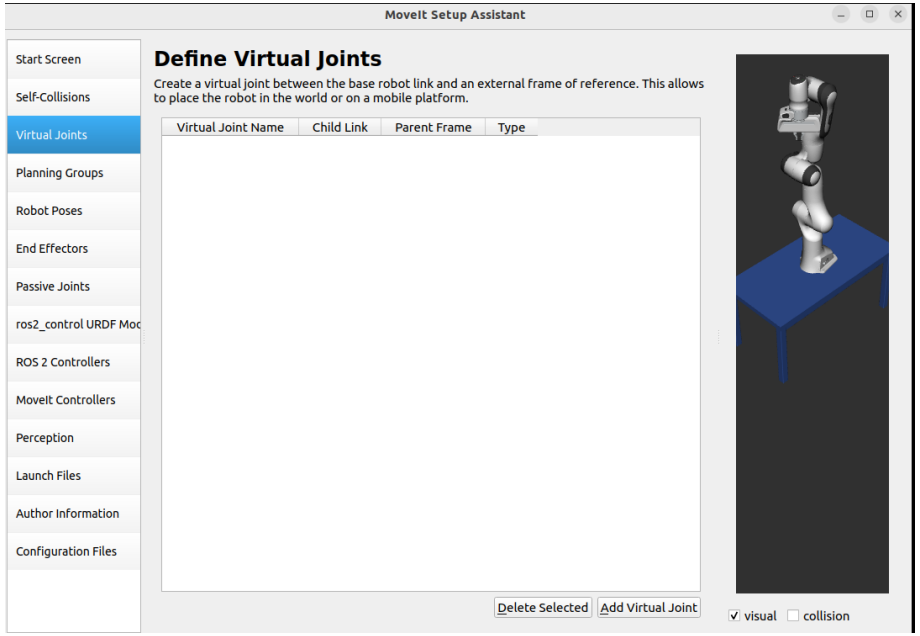
Moveit Setup Assistant – Self Collision Checking



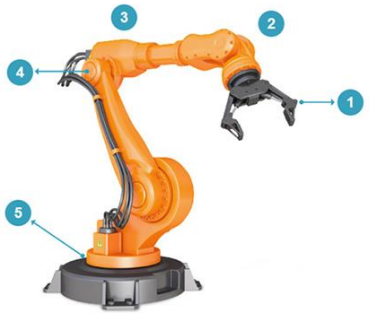
<u>Self Collision Checking</u>

- Setup Assistant has a self collision matrix generation feature which validates collision with each of individual links with other links in the robot arm. It checks ,
  - Pairs which collide
  - Pairs which never collides
  - Adjacent Pairs
- The matrix accuracy is based on Sampling Density.
  - Higher the Sampling Density gets more accurate Self collision matrix.

Moveit Setup Assistant – Virtual Joints



Virtual Joint - Required



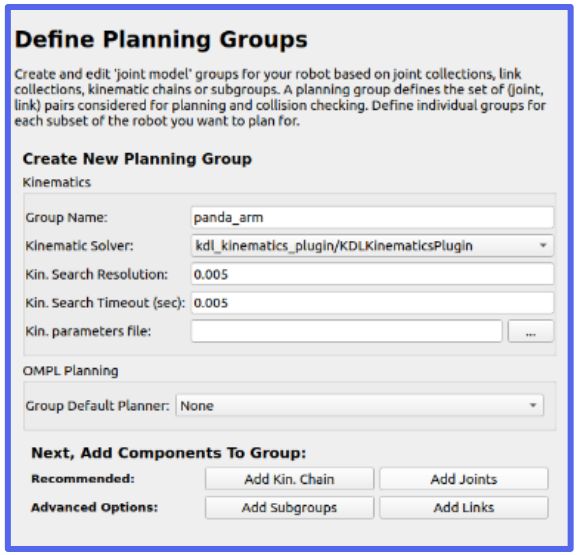
Virtual Joint - Optional

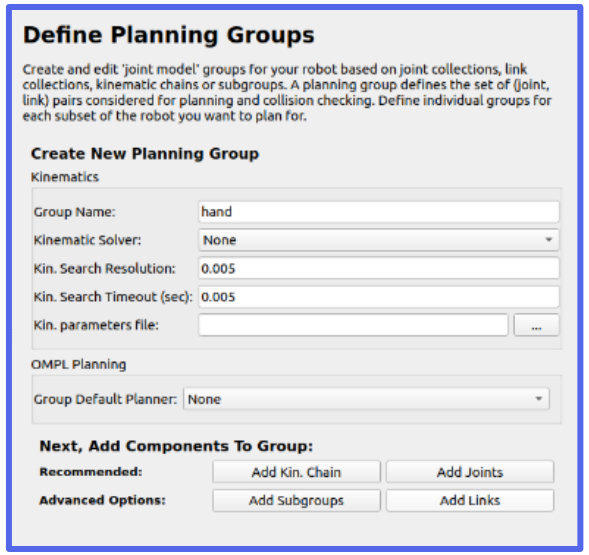
Virtual Joint - Required

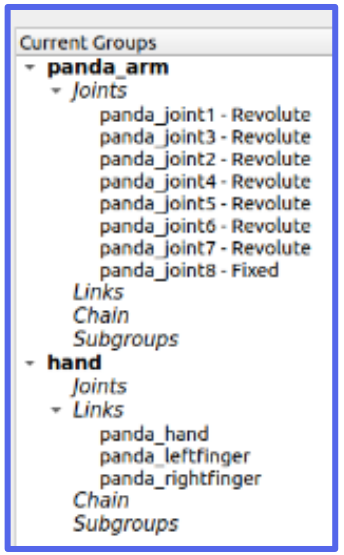
Virtual Joints

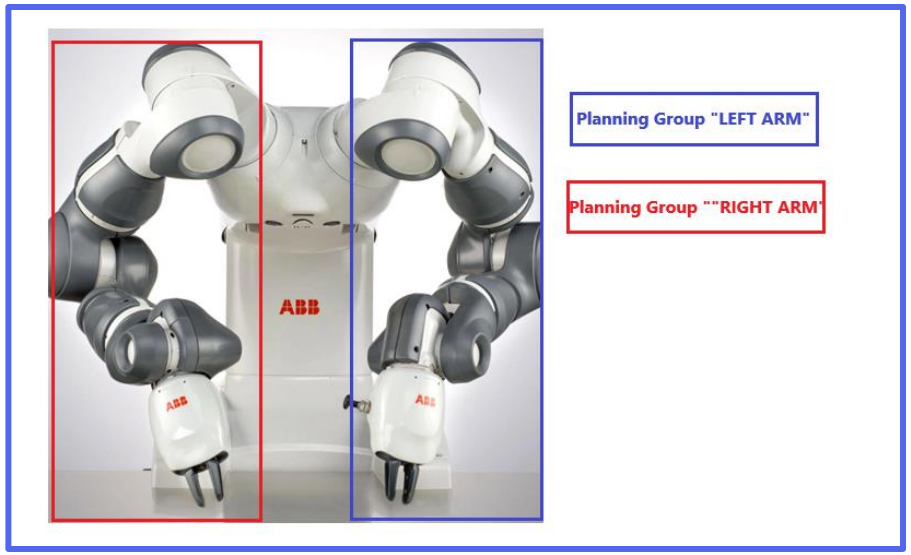
- Virtual Joints are reference to connect robot arms to the world.
- The Robot Arm with Fixed Base as its first link doesn't require virtual joint
- Robot Arm on mobile bases and Robot Arm with additional axis required virtual joint to be specified.

Moveit Setup Assistant – Planning Group





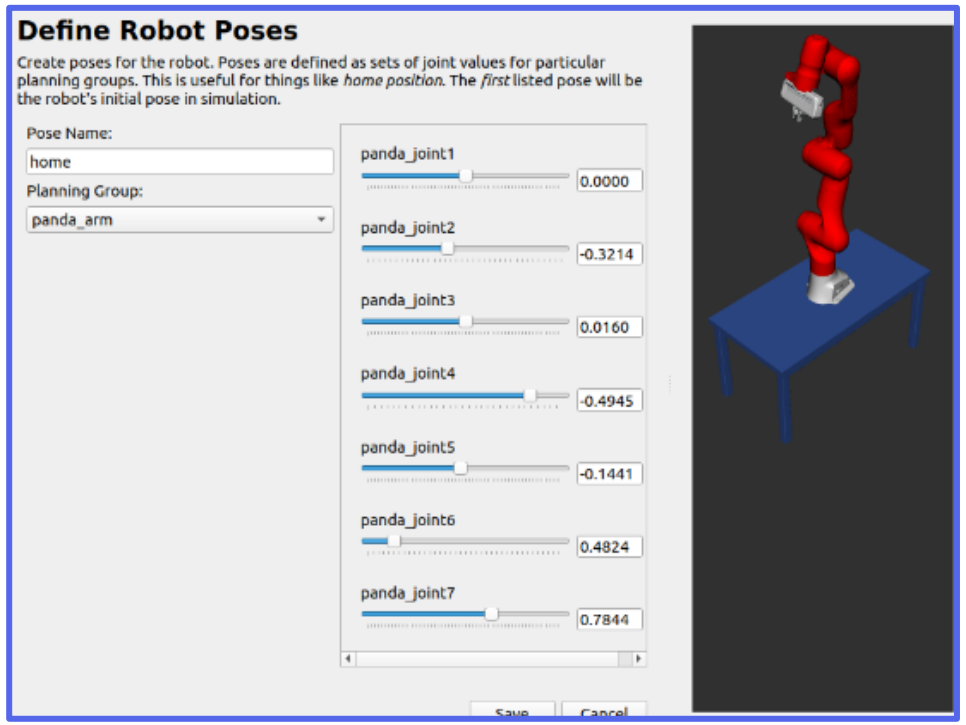


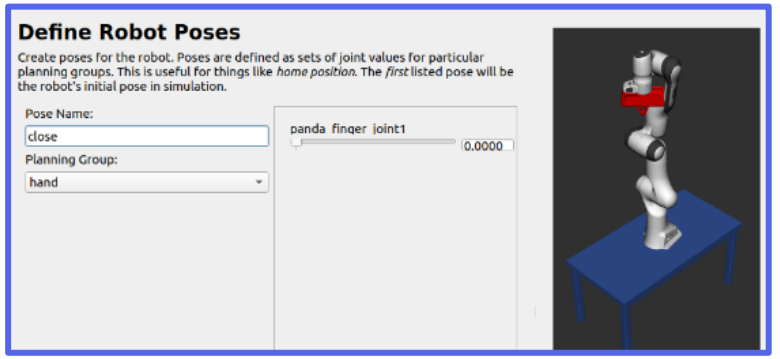


Planning Groups

- Setup Assistant lets user to define specific planning group for specific set of desired joints.
- This in turn facilitates move group to perform motion planning for specific set of joints at a time.
- Planning groups can be categorized for left and right arm of Dual Robot Arms, or to separate Robot arm motion from Grippers.

Moveit Setup Assistant – Robot Poses

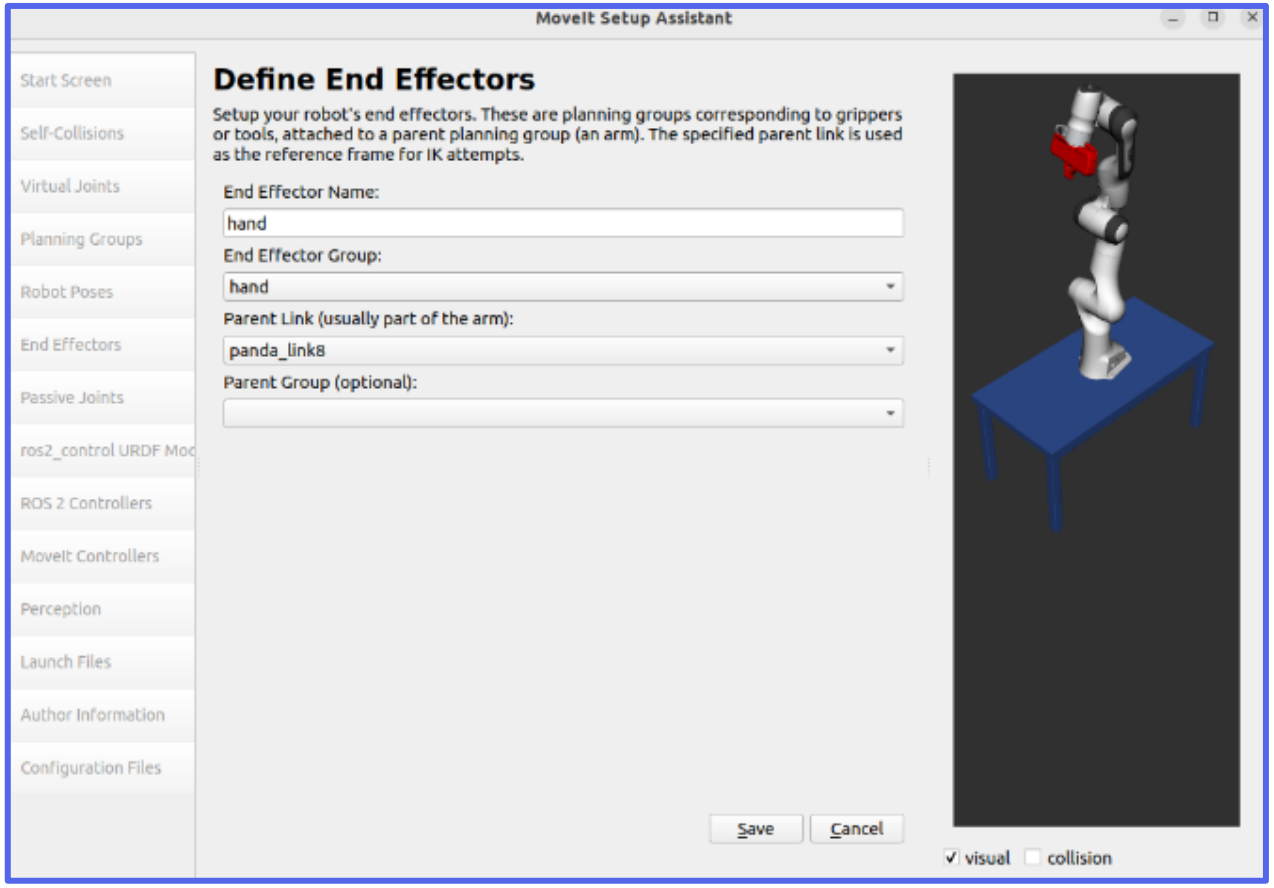




Robot Poses:

- Setup Assistant lets user to define predefined poses based on different planning groups.
- The user can define a specific desired pose for the robot arm and later move group can be used to call robot to plan and execute the desired pose.
- The user can also define open and close poses of gripper for grasping related tasks.

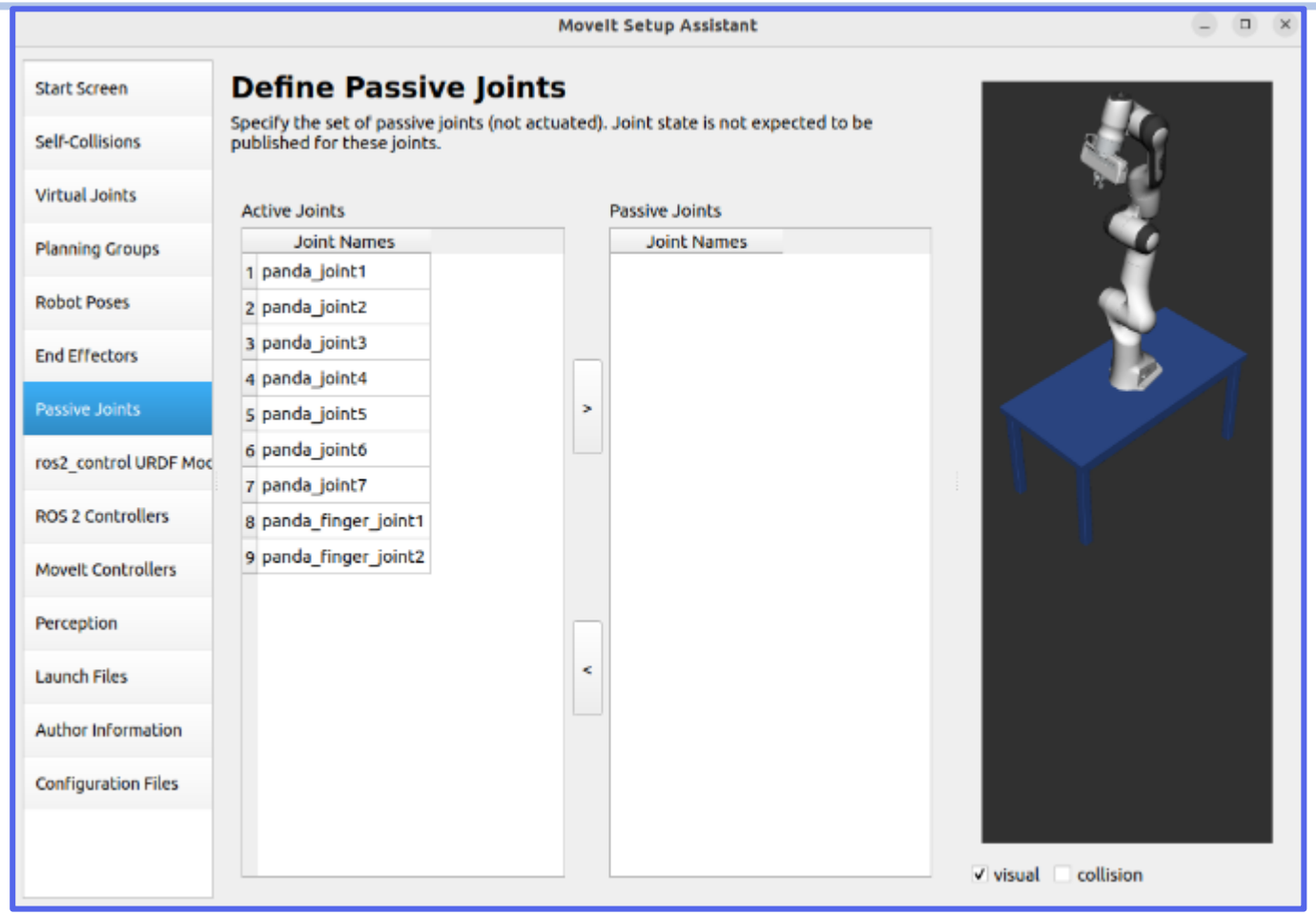
Moveit Setup Assistant – End Effectors



End Effectors:

- Setup Assistant lets user to configure End Effector based on defined planning group to configure end effector related operations like pick and place.

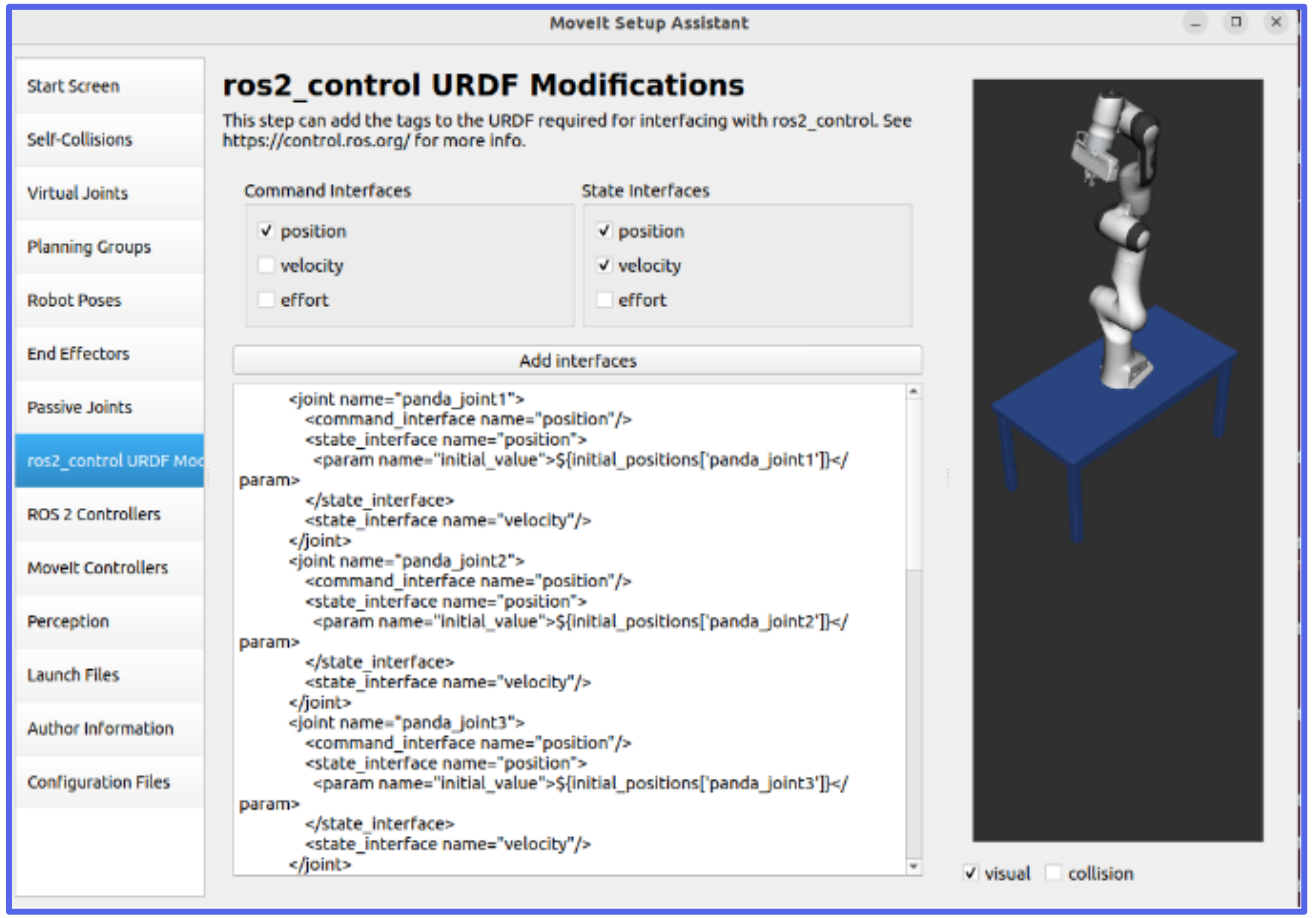
Moveit Setup Assistant – Passive Joints



Passive Joints:

- Setup Assistant lets user to define unactuated passive joints which may exist in the robot arm.
- Existing Passive joints must be defined to avoid unaware collisions during planning operation.

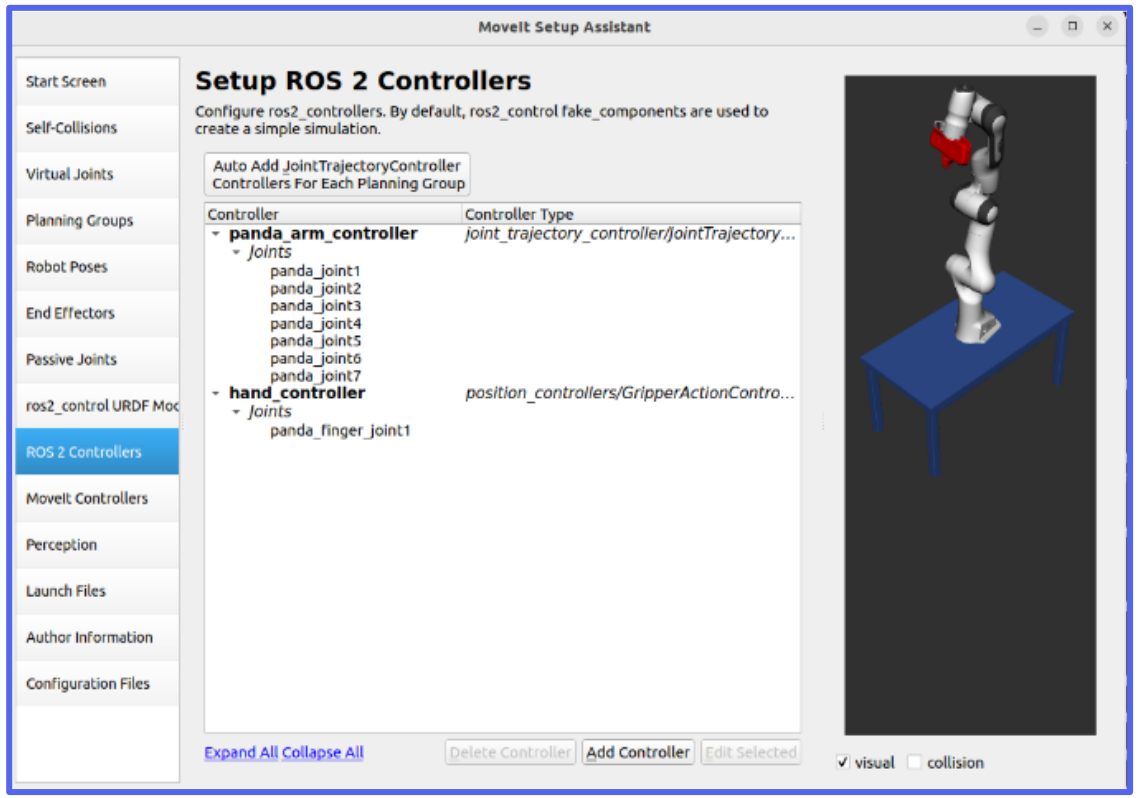
Moveit Setup Assistant – ROS2 Control URDF



ROS2 Control URDF

- The ROS2 Control URDF interfaces ROS2 Control to perform low level control (command) and feedback (state) interface operations.
- The User can select the necessary command and state interfaces to perform robot arm control operation.

Moveit Setup Assistant – ROS2 Controllers

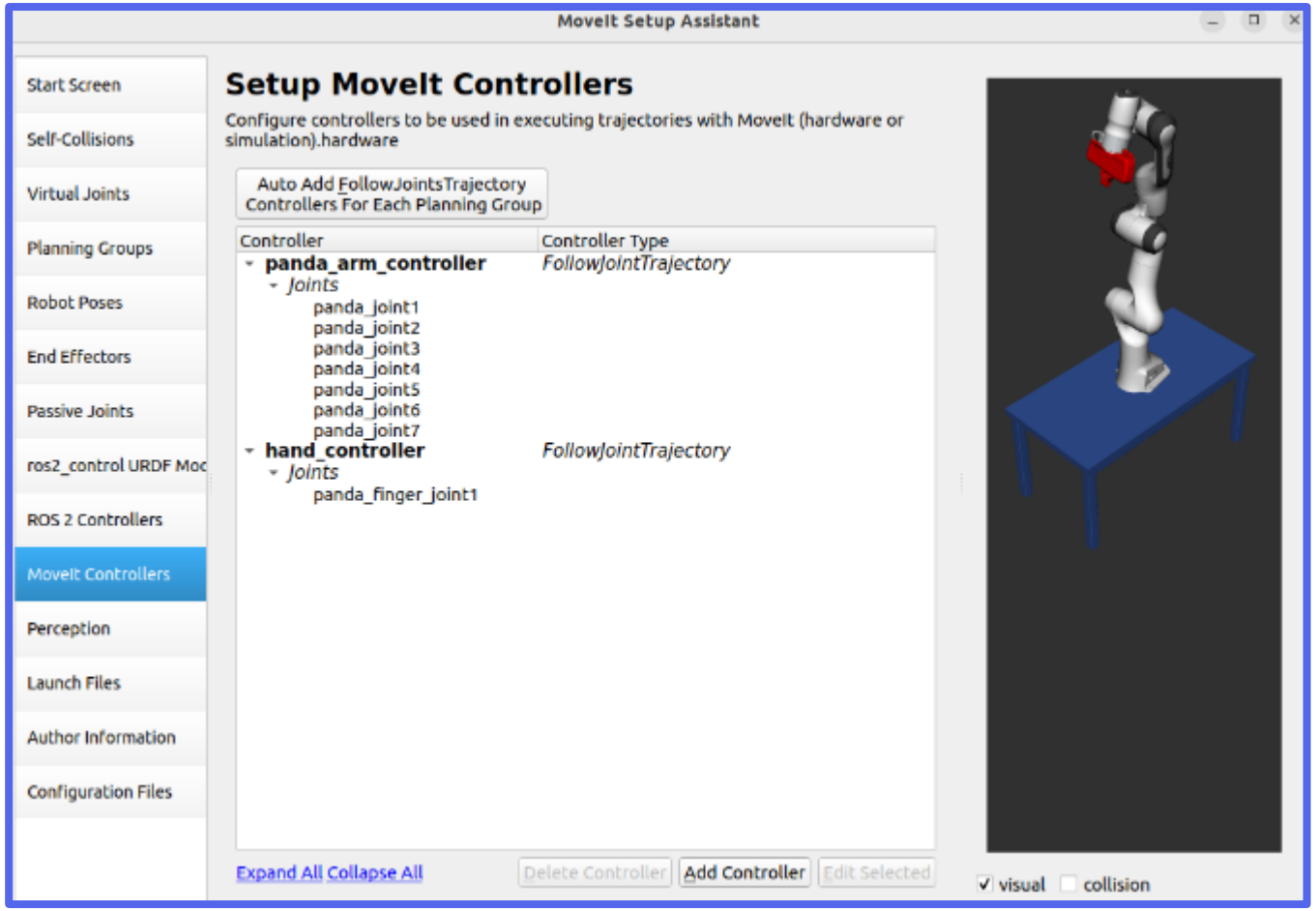


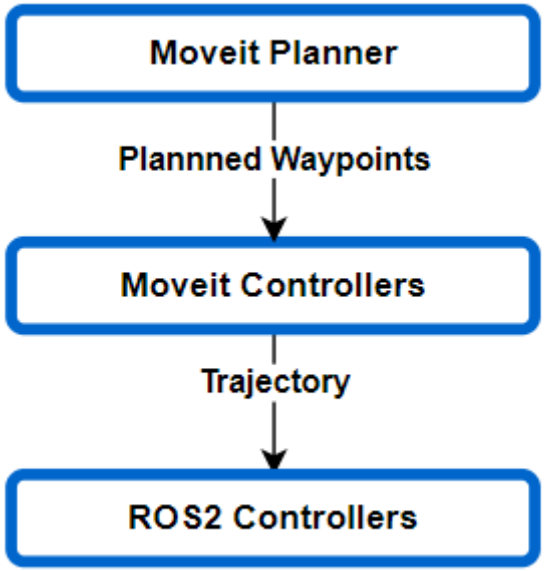
ROS2 Controllers

- ROS 2 Control is a framework for real-time control of robots, designed to manage and simplify the integration of new robot hardware.
- The ROS2 control focuses on different types of controller which focuses on command (control), state(feedback), user interface based joint control, passthrough controllers and IO controllers.

https://github.com/ros-controls/ros2_controllers

Moveit Setup Assistant – Moveit Controllers

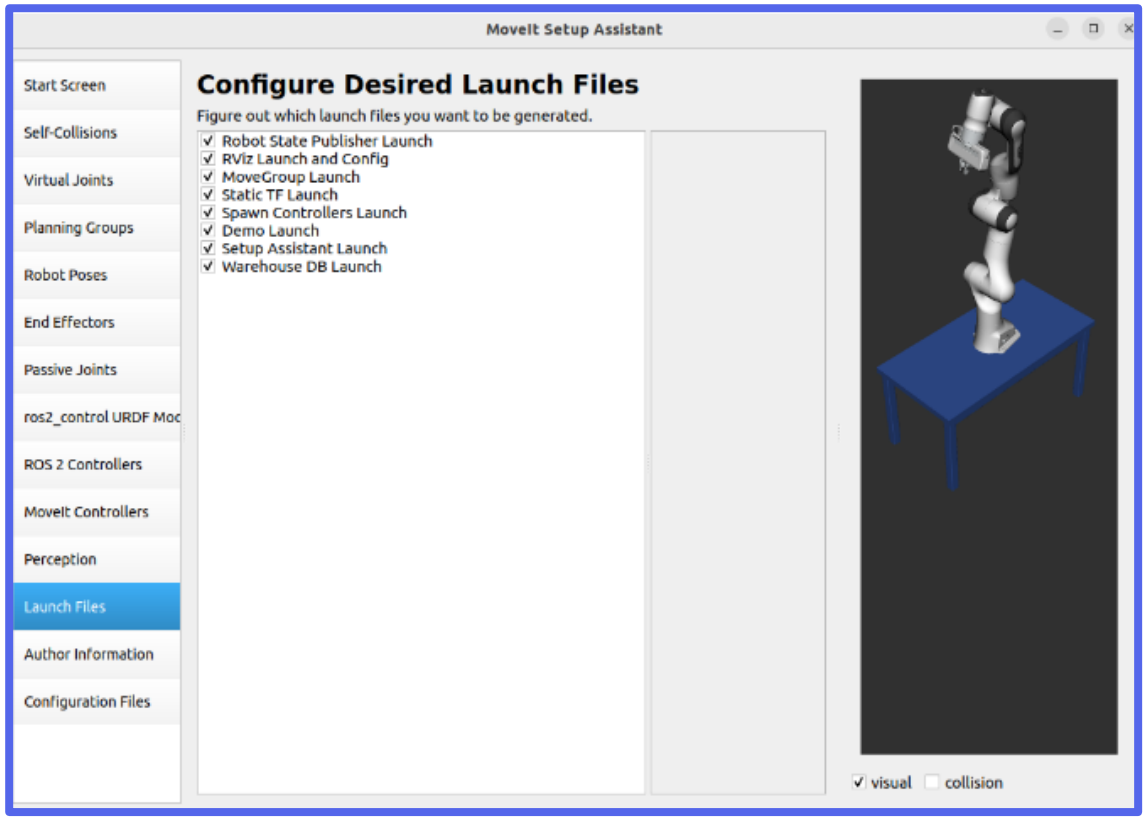


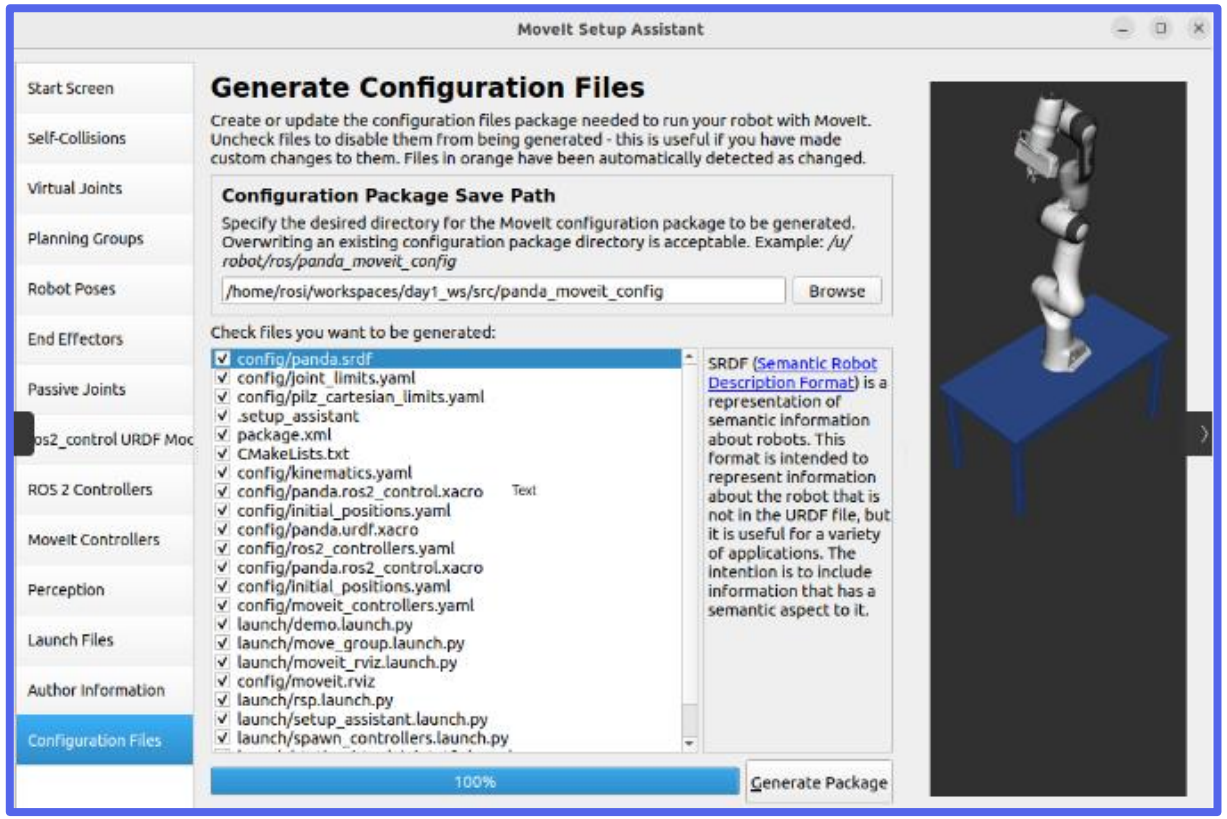


Moveit Controllers

- Moveit Controllers are high level controllers which generate trajectories to be executed based on the plan generated from the moveit planners. This trajectory is passed on to low level ROS2 controllers to perform execution.

Moveit Setup Assistant – Launch Files and Configurations





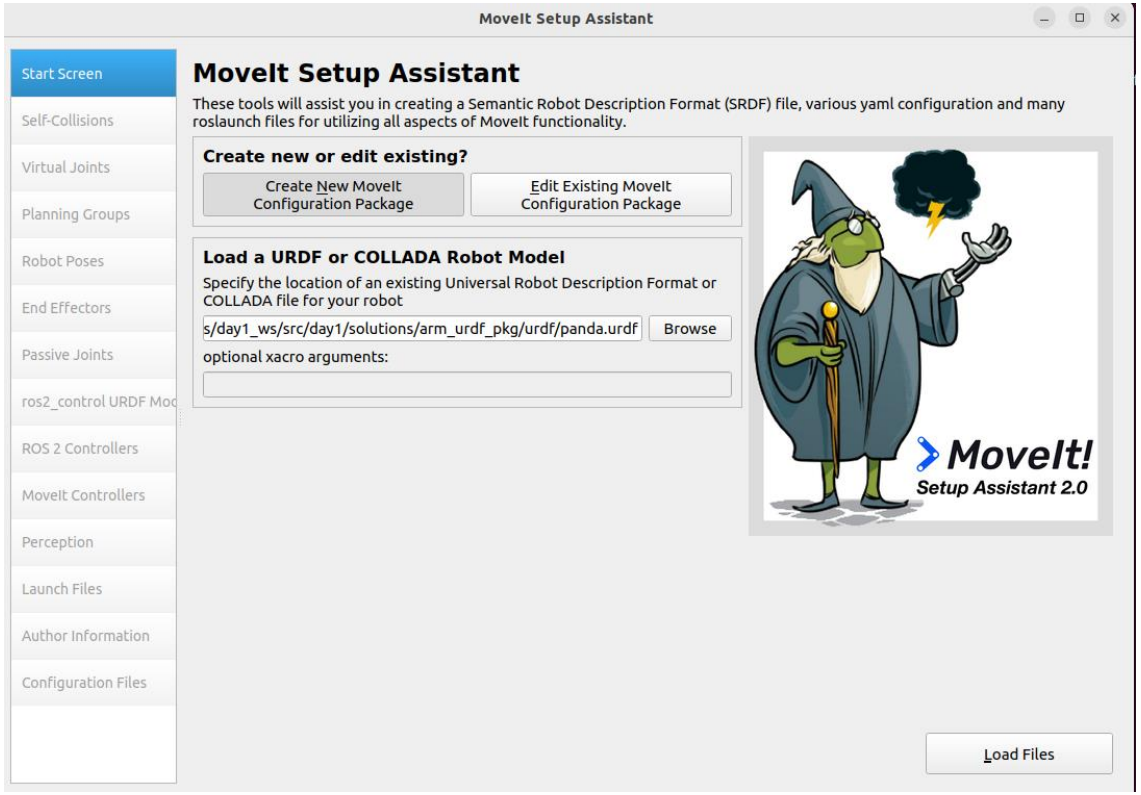
Launch Files and Configuration

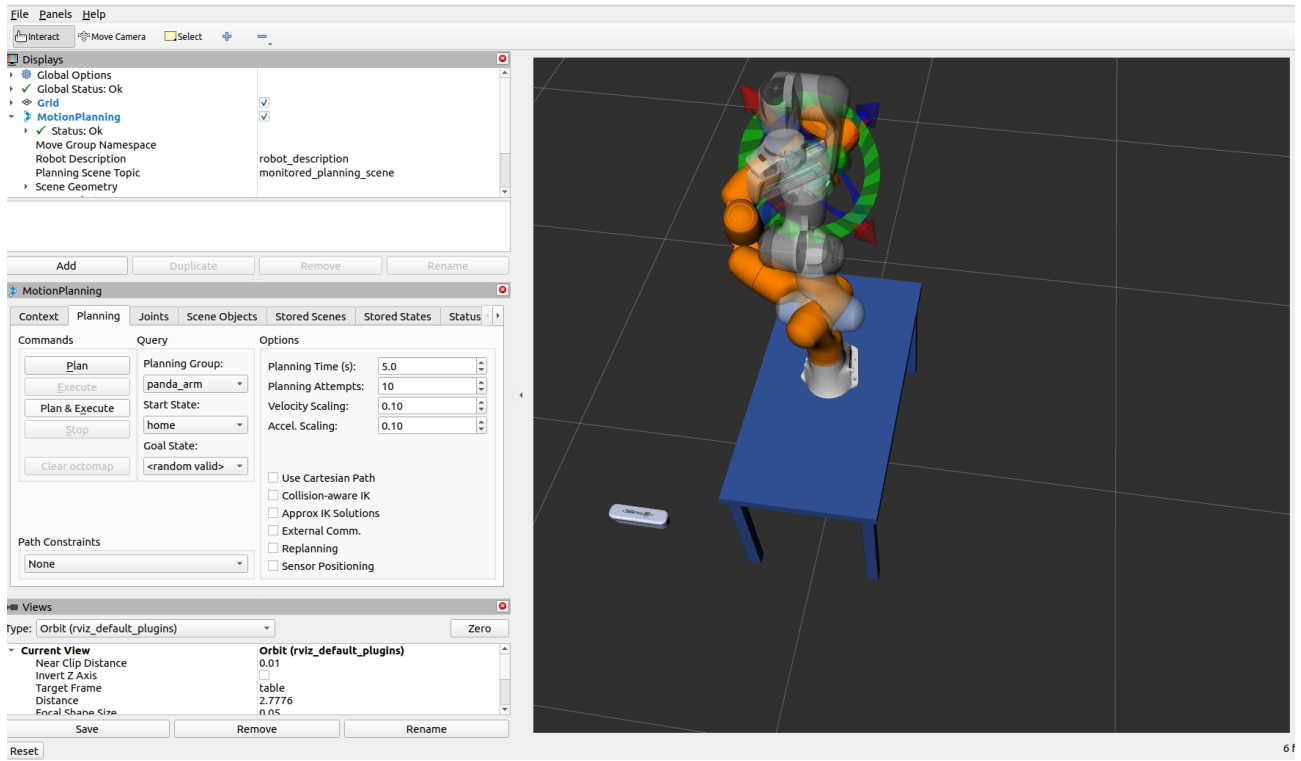
- The Setup Assistant generates necessary launch files and semantic robot configurations files to perform motion planning. This information is later used by move group as an input to perform seamless motion planning.

Exercise 2: Use Moveit Setup Assistant to Develop Moveit Config Package

For this exercise,

1. **Moveit Setup Assistant** will be used to create Moveit Config





Tree Image source: http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file

ROS-Industrial Manipulation Training - Agenda Day 1

- What is Manipulation?
  - Concepts
  - Applications
- Robot Description
  - URDF
    - Package Content
    - URDF Components
    - Exercise 1: Visualize Robot Description by adding table and camera
- Moveit Setup Assistant
    - Setup Components
    - ROS2 Controllers [High-Level]
    - Moveit Controllers [High-Level]
    - Exercise 2: Create Moveit Config and Launch
- Move Group
    - Move Group Capabilities
    - Exercise 3: Motion Planning with Move Group
- Perception with Camera and ARUCO Marker
    - Xacro [High-Level]
    - Exercise 4: ARUCO Detection and Spawn Collision Object
- Perception with Gazebo Simulation [Take Home (optional)]
    - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo

Advanced Remanufacturing and Technology Centre

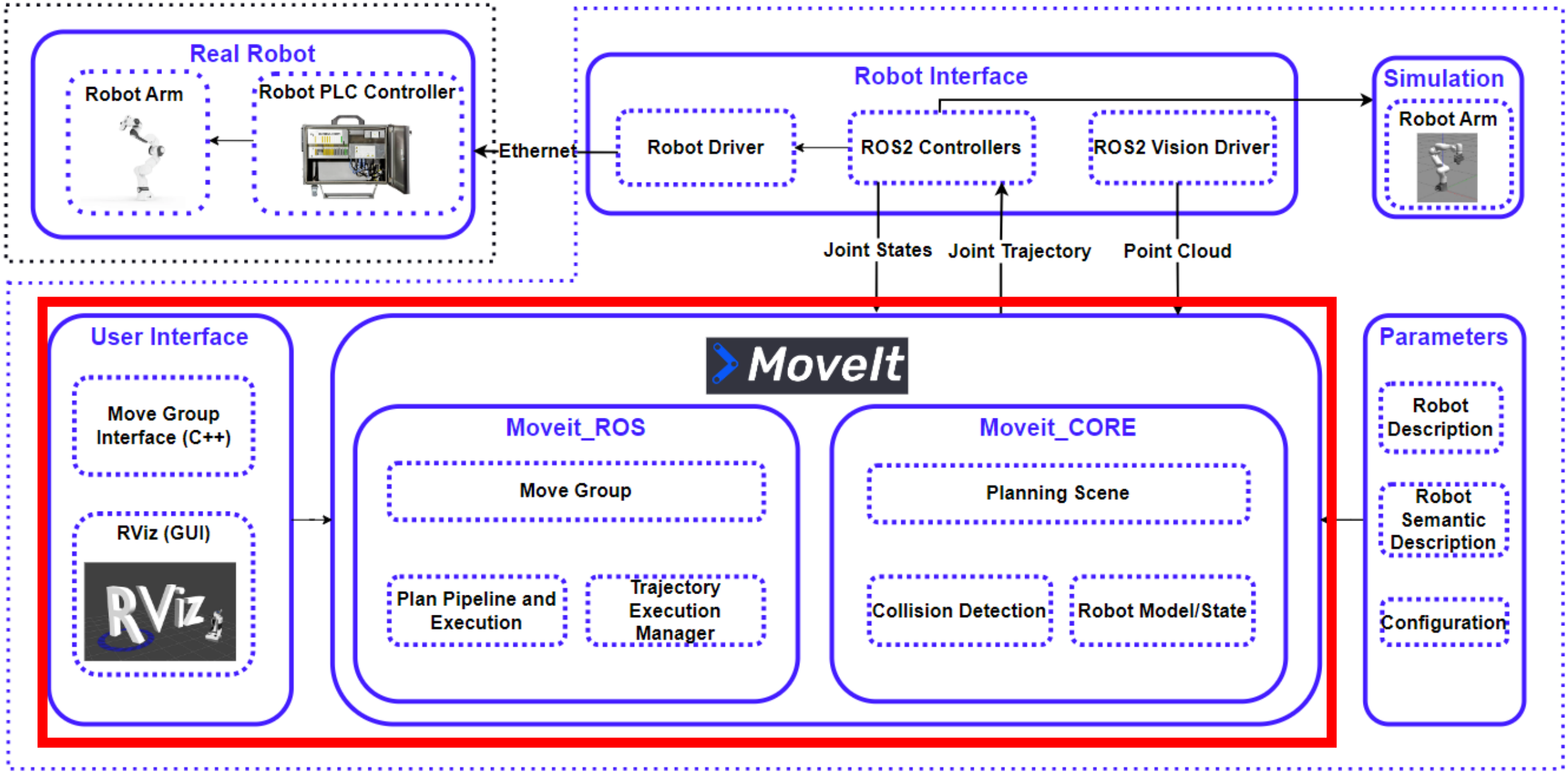
ARTC

MOVE GROUP

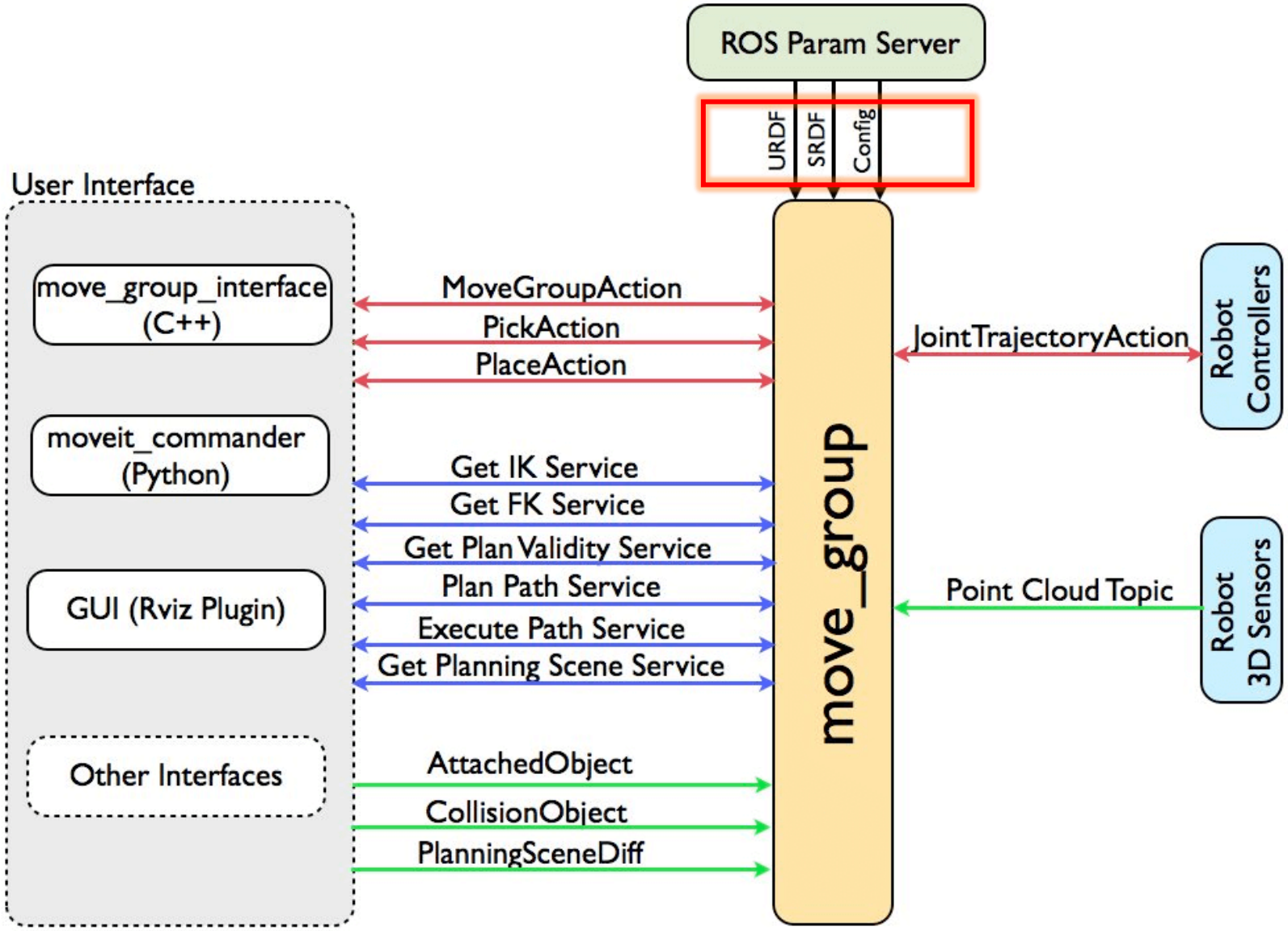
Shalman Khan

4th December 2023

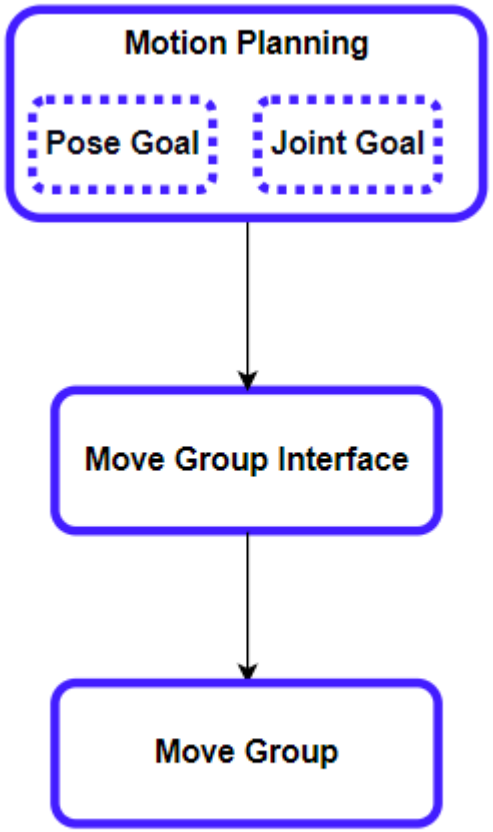
Moveit Setup Assistant



Move Group – Architecture

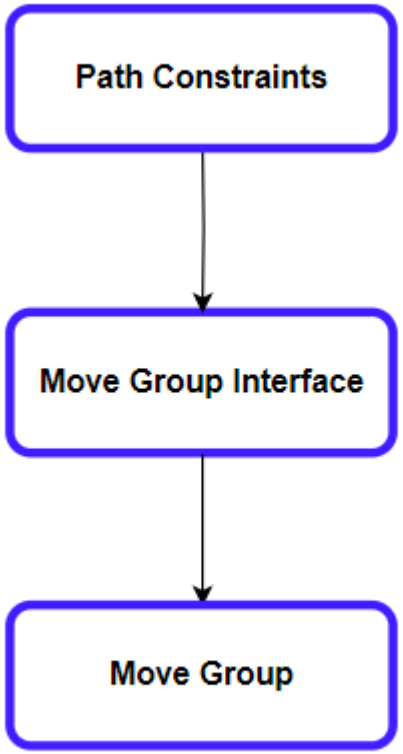


Move Group – Capabilities



Motion Planning

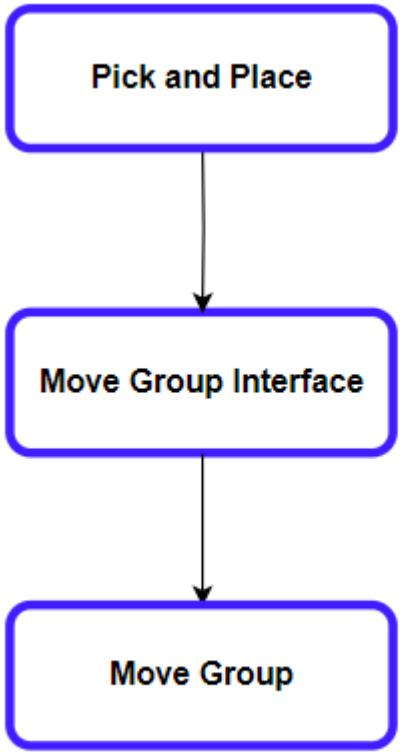
- Move Group provides high-level interfaces for motion planning. It can compute collision-free paths for the robot's end-effector to reach a specified goal position.
- Move Group supports both Joint Space goal positions and end effector-based goal positions



Path Constraints

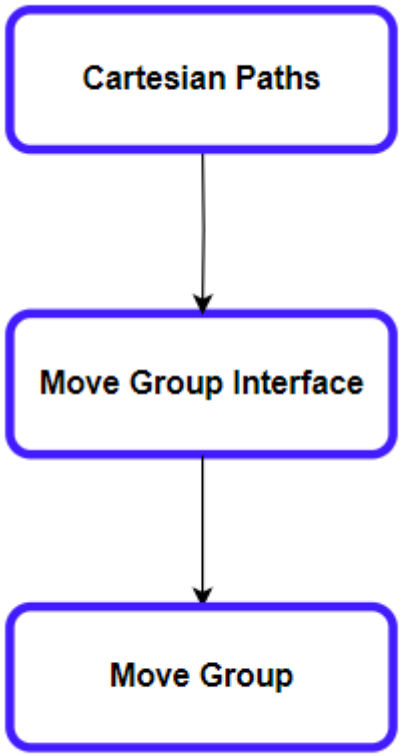
- Move Group allows you to specify path constraints, such as keeping certain joints fixed or maintaining a specific distance between two parts of the robot. This is crucial for tasks where the robot must follow a specific trajectory.

Move Group – Capabilities



Pick and Place

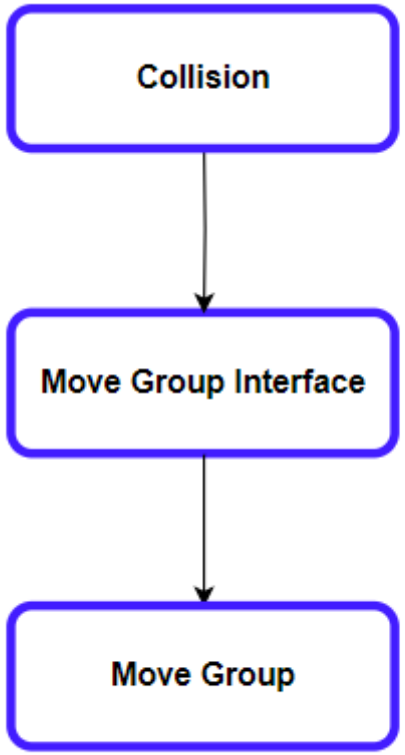
- Move Group supports pick and place operations, allowing the robot to grasp objects from specific locations and place them in desired positions.
- Move Group also supports Moveit Grasps and Moveit Deep Grasps for pick and place operation.



Cartesian Paths

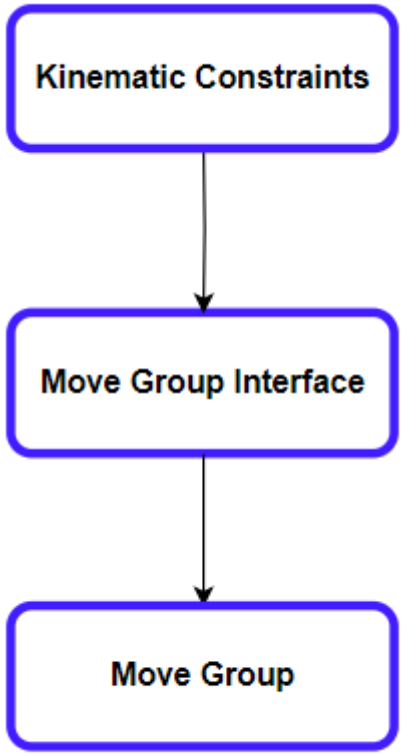
- Move Group can compute Cartesian paths, enabling the robot to follow straight-line paths in the Cartesian space.
- This is useful for applications where the end-effector needs to move linearly.

Move Group – Capabilities



Collision:

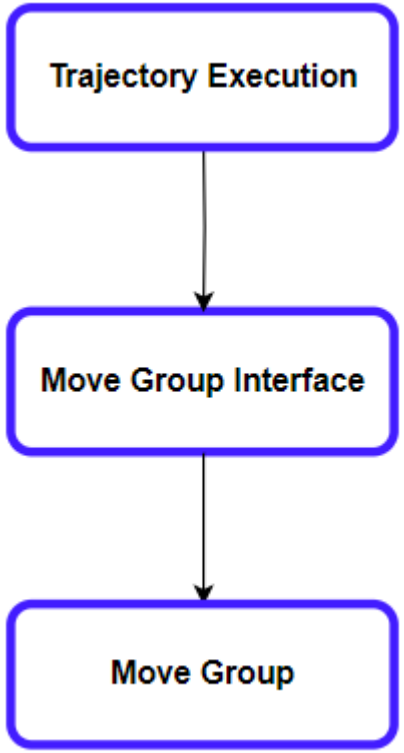
- Move Group performs collision checking to ensure that the planned paths are collision-free. It uses the robot's collision model to validate trajectories.
- Move Group interface can add collision objects as primitives.



Kinematic Constraints:

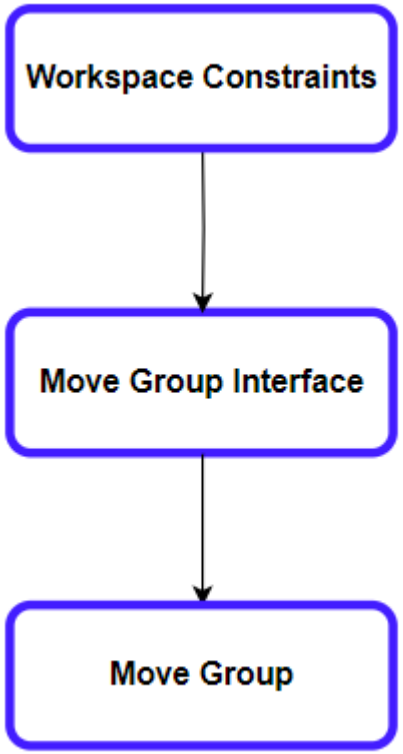
- Move Group can enforce kinematic constraints, ensuring that the robot's joints stay within specified limits during motion.

Move Group – Capabilities



Trajectory Execution:

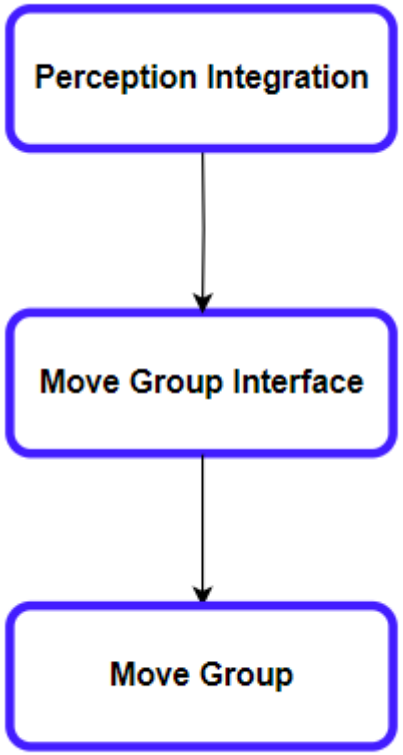
- Move Group can execute planned trajectories on the robot. It interfaces with the robot's hardware interface to control the robot and follow the planned paths.



Workspace Constraints:

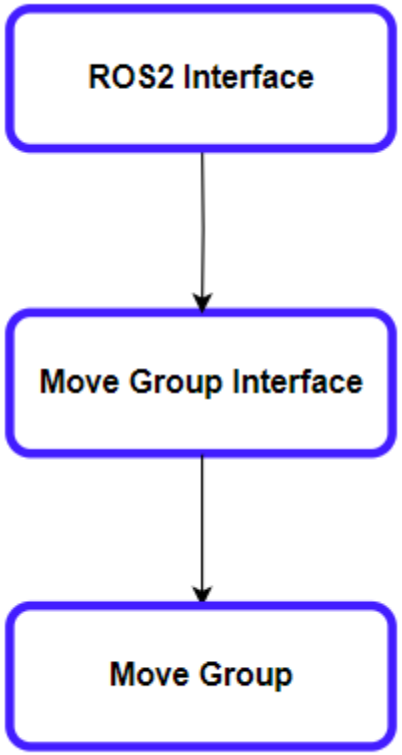
- Move Group allows you to define constraints on the workspace, limiting the robot's motion to a specific region. This is useful for applications with constrained workspace.

Move Group – Capabilities



Perception Integration:

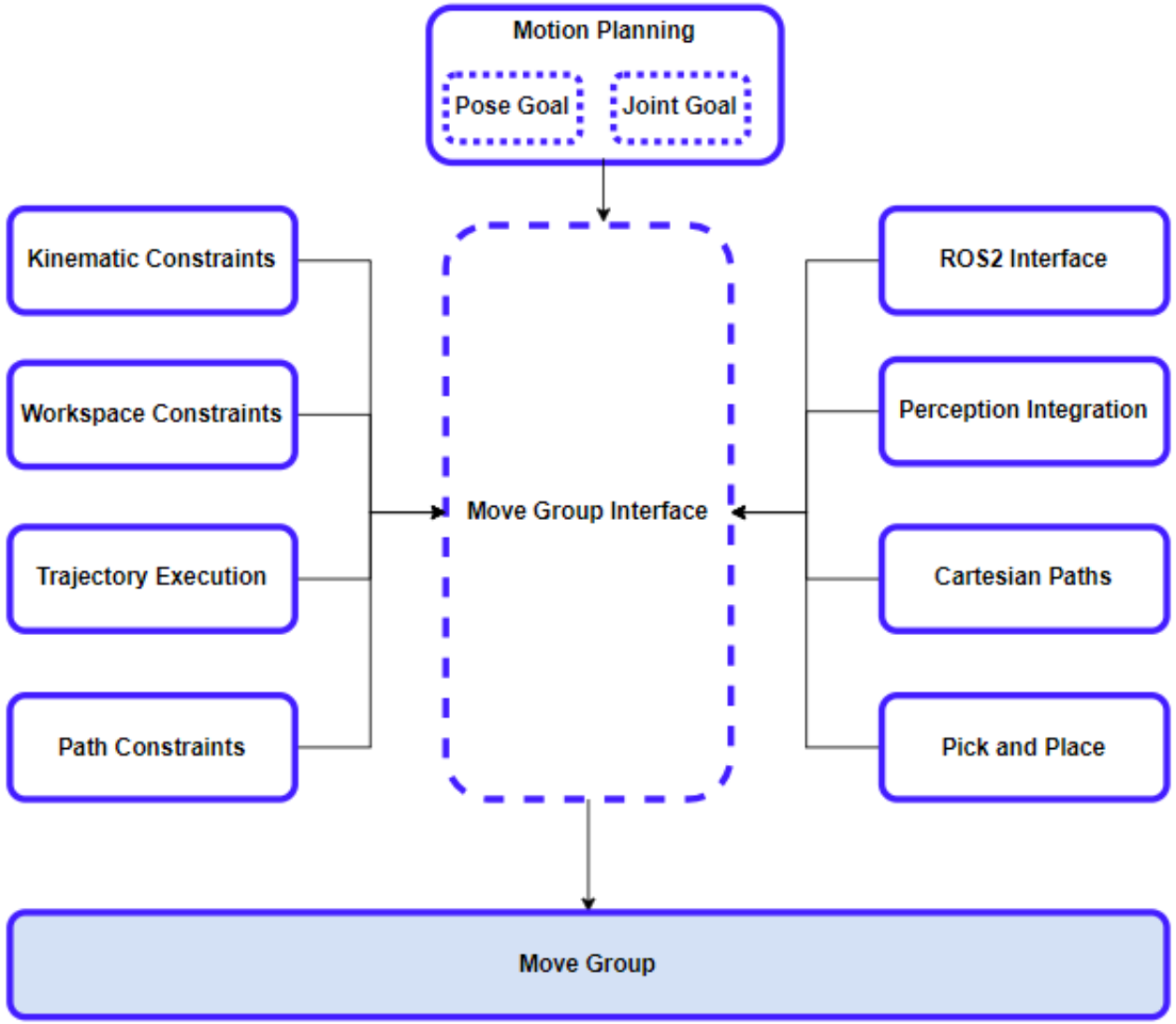
- Move Group can be integrated with perception systems, allowing the robot to plan and execute motions based on real-time sensory input.



ROS2 Interface

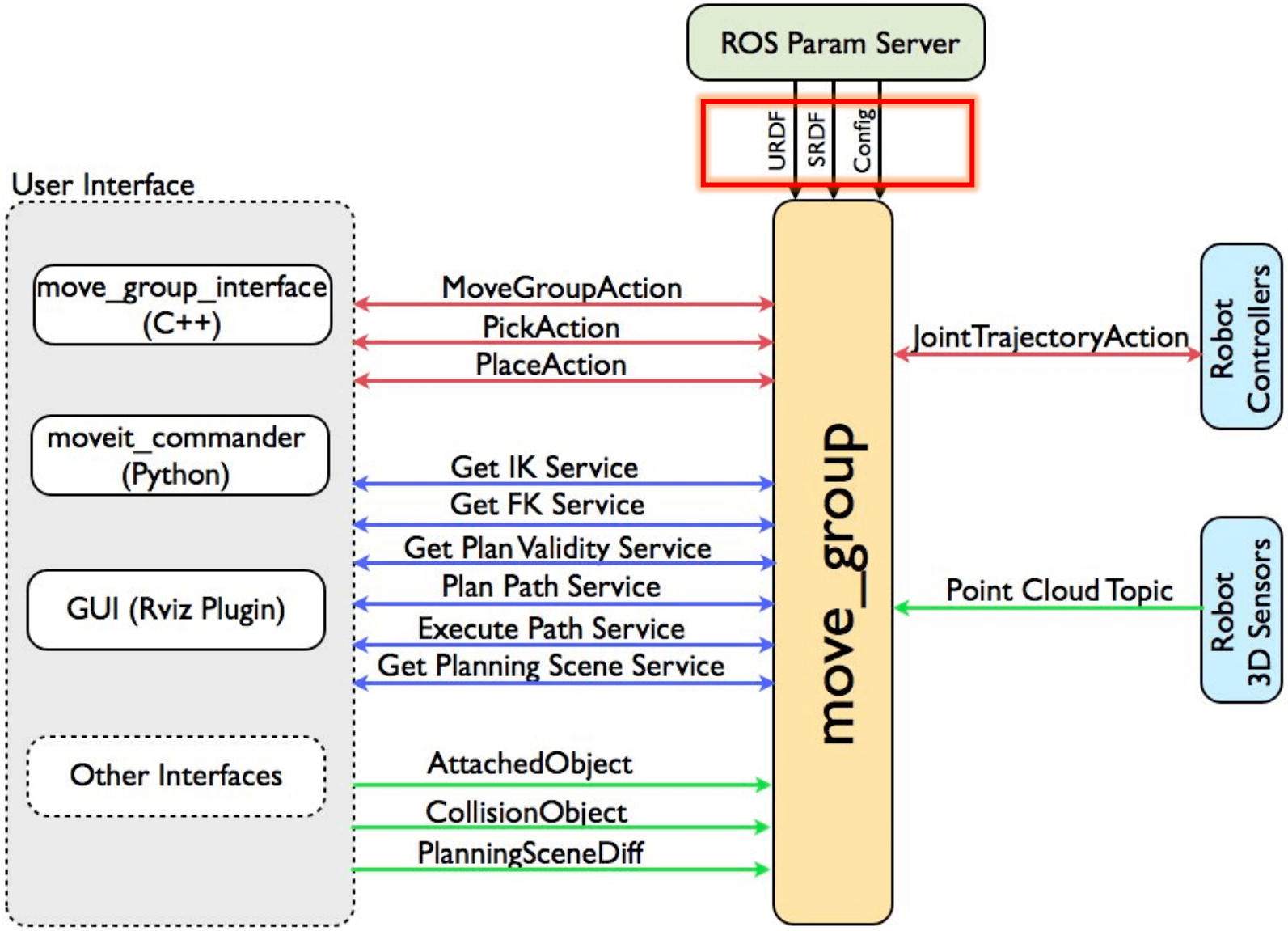
- Move Group provides ROS interfaces, allowing you to control the robot using ROS messages and services. It integrates seamlessly into the ROS ecosystem.

Move Group – Capabilities

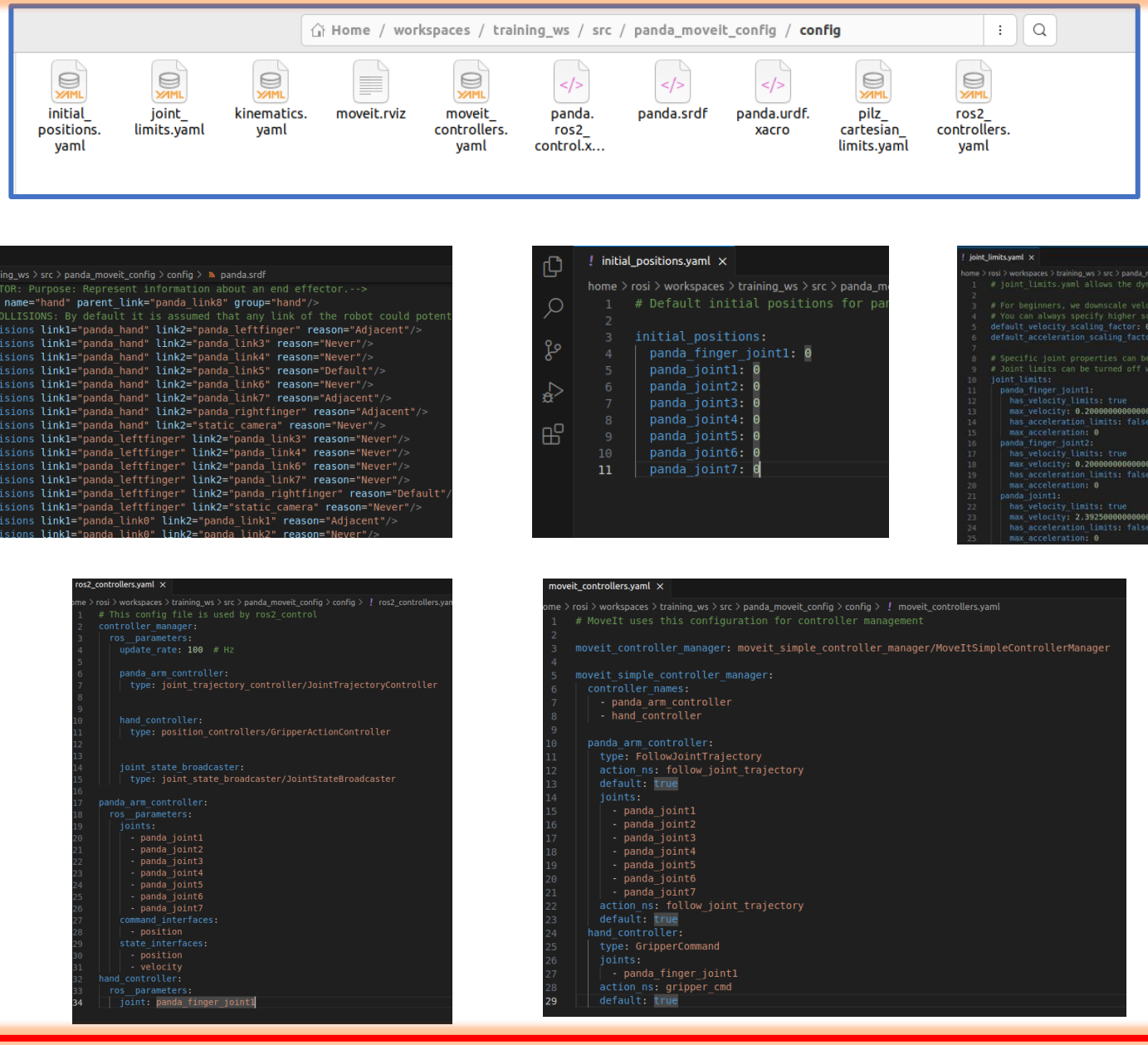


interface_guide

Move Group – Architecture



Move Group – Param Inputs



Move Group Interface – Initialization

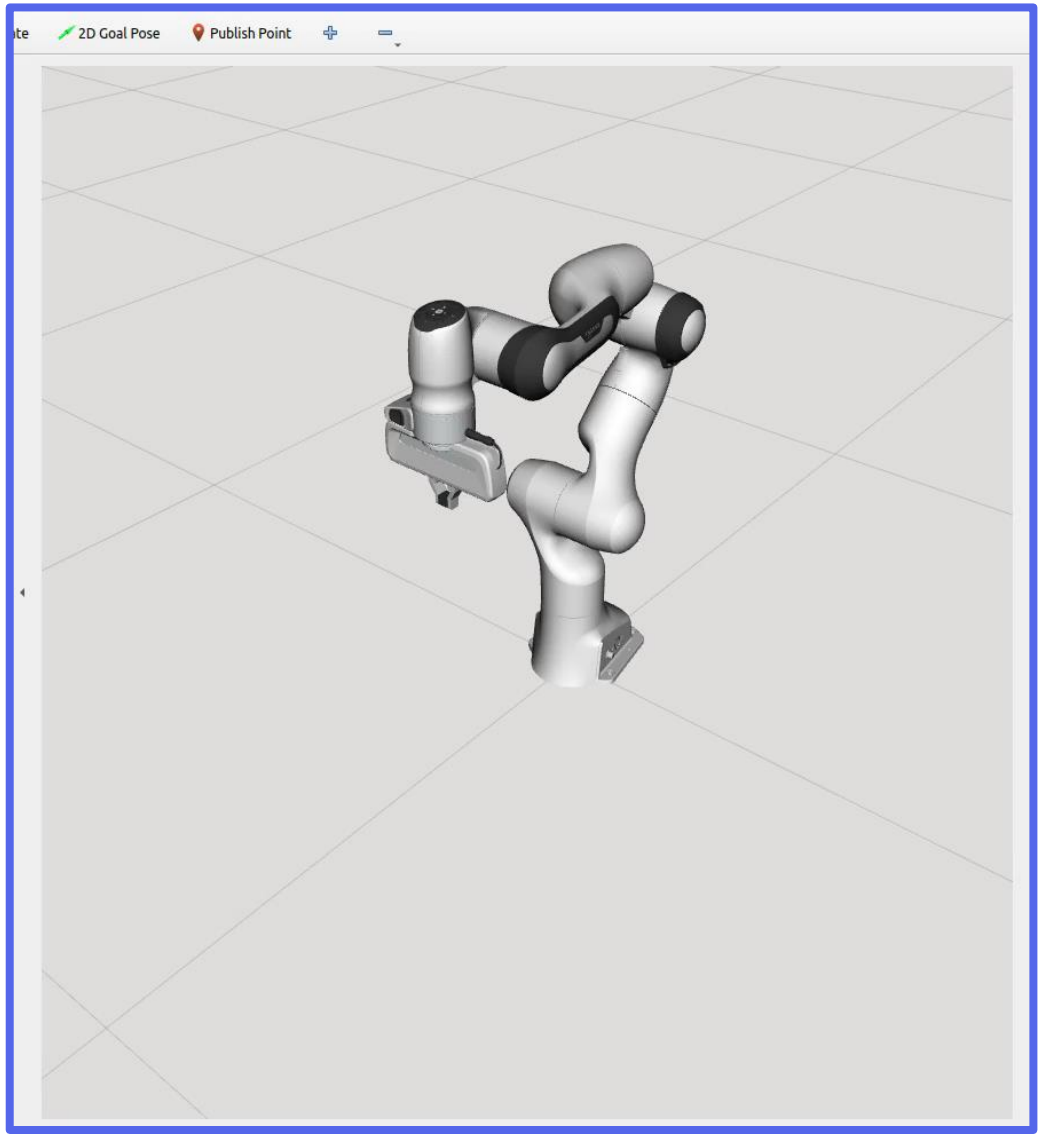
```cpp
//  Initializes the ROS C++ client library
rclcpp::init(argc, argv);

// Options to customize the behavior of the node
rclcpp::NodeOptions node_options;

// Sets an option for the node to automatically declare parameters from overrides
node_options.automatically_declare_parameters_from_overrides(true);
auto move_group_node = rclcpp::Node::make_shared("movet_exec_interface", node_options);

// Spins up a SingleThreadedExecutor for the current state monitor to get information
// about the robot's state.
rclcpp::executors::SingleThreadedExecutor executor;

// Adds the node to executor
executor.add_node(move_group_node);

// Detach makes the thread to run individually in the background
std::thread([&executor]() { executor.spin(); }).detach();
```

Move Group Interface – Move Group Interface and Planning Initialization

```
// Define Planning Group
static const std::string PLANNING_GROUP = "panda_arm";

// Add Planning Group the Move Group Interface which Communicates with the Move Group
moveit::planning_interface::MoveGroupInterface move_group(move_group_node, PLANNING_GROUP)

// Define Joint Model Group to extract the set of joints to control from specific planning
const moveit::core::JointModelGroup* joint_model_group =
    move_group.getCurrentState()->getJointModelGroup(PLANNING_GROUP);

// Class to describe the Planning Scene [Helps to add objects to the planning environment]
moveit::planning_interface::PlanningSceneInterface planning_scene_interface;

// Current set of Joint Values
moveit::core::RobotStatePtr current_state = move_group.getCurrentState(10);

// Current set of Pose Values
auto current_pose = move_group.getCurrentPose();
```

Move Group Interface – Setting a Pose Goal

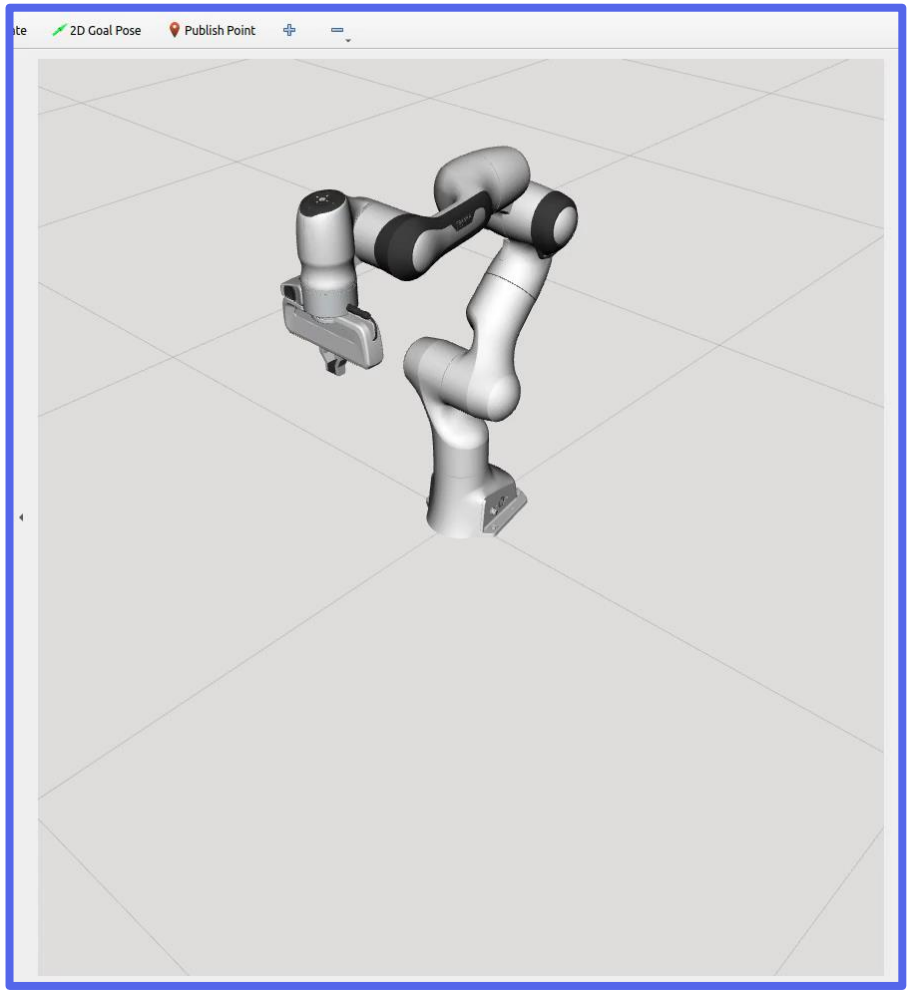
```cpp
// Creating an instance of the Pose message from the geometry_msgs package
geometry_msgs::msg::Pose target_pose;

// Setting the orientation of the target pose
target_pose.orientation.w = 0.0;   // Quaternion's w component
target_pose.orientation.x = -1.0;  // Quaternion's x component
target_pose.orientation.y = 0.0;   // Quaternion's y component
target_pose.orientation.z = 0.0;   // Quaternion's z component

// Setting the position of the target pose
target_pose.position.x = 0.6;   // X coordinate
target_pose.position.y = 0.0;   // Y coordinate
target_pose.position.z = 0.5;   // Z coordinate

// Set the defined target using move group interface
move_group.setPoseTarget(target_pose);
```



interface_guide

Move Group Interface – Setting a Joint Goal

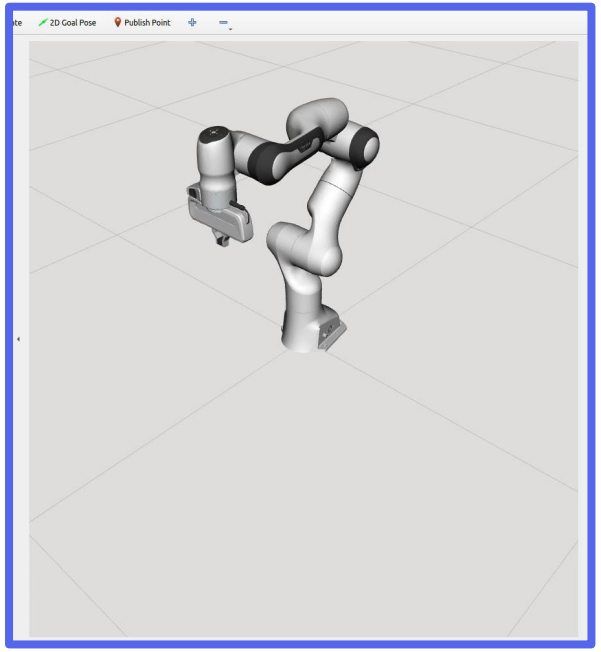
```cpp
// Creating an instance of the Joint message
std::vector<double> joint_group_positions;
current_state->copyJointGroupPositions(joint_model_group, joint_group_positions);

// Setting the target joint angle for each joint
joint_group_positions[0] = 2.3541267464420828;  // radians
joint_group_positions[1] = -1.7049057277868336; // radians
joint_group_positions[2] = -0.8000612682072692; // radians
joint_group_positions[3] = -1.6239137302647184; // radians
joint_group_positions[4] = -1.7064713521395218; // radians
joint_group_positions[5] = 0.7989745787375141;  // radians
joint_group_positions[6] = 0.7260783211238884;  // radians

// Set the defined joint target using move group interface
move_group.setJointValueTarget(joint_group_positions);
```



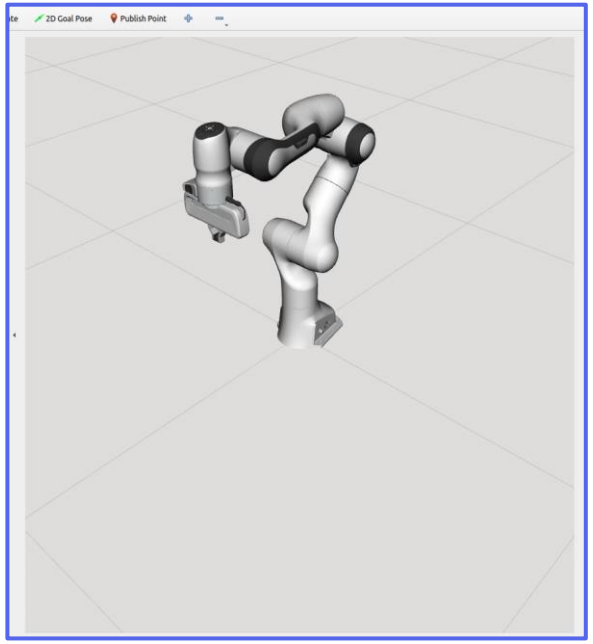
Move Group Interface – Initiate Plan and Move

```cpp
// Defining and Assgning plan to move group interface
moveit::planning_interface::MoveGroupInterface::Plan plan;
auto success = (move_group.plan(plan) == moveit::core::MoveItErrorCode::SUCCESS);

// Moves to the target after the plan
move_group.move();

// Executes to the target with the plan [Blocking]
move_group.execute(plan);
```

Without move() or execute()

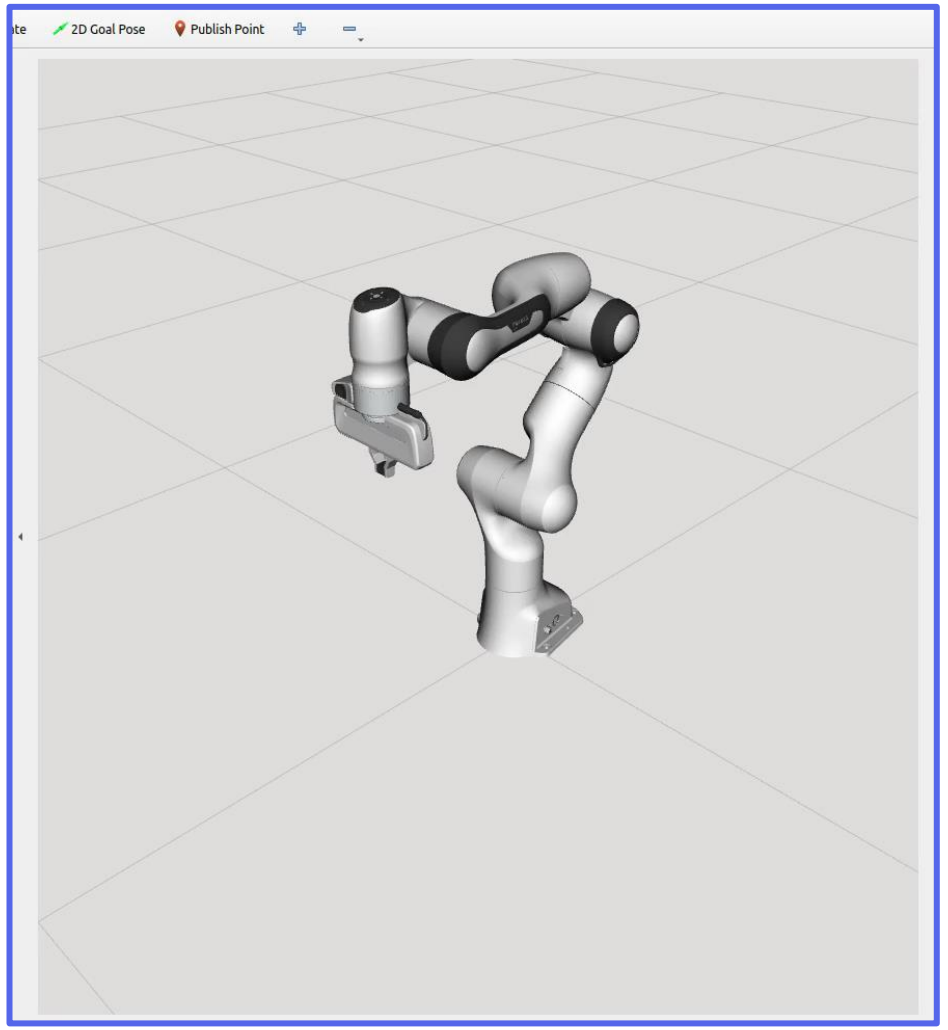


With move() or execute()



Move Group Interface – Add Standard Collision Object

```cpp
// Creating an instance of the Collision object
moveit_msgs::msg::CollisionObject collision_object;
collision_object.header.frame_id = move_group.getPlanningFrame();

// The id of the object is used to identify it.
collision_object.id = "collision_box";

// Define the size and shape primitive
shape_msgs::msg::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[primitive.BOX_X] = 0.02;
primitive.dimensions[primitive.BOX_Y] = 0.3;
primitive.dimensions[primitive.BOX_Z] = 0.3;

// Define the Pose of the collision object with respect to the frame id
geometry_msgs::msg::Pose box_pose;
box_pose.orientation.w = 1.0;
box_pose.position.x = 0.45;
box_pose.position.y = 0.0;
box_pose.position.z = 0.35;

// Push the pose and primitive of the collision object
collision_object.primitives.push_back(primitive);
collision_object.primitive_poses.push_back(box_pose);
collision_object.operation = collision_object.ADD;

// Add collision object to the main collision objects vector
std::vector<moveit_msgs::msg::CollisionObject> collision_objects;
collision_objects.push_back(collision_object);

// Add collision objects to the planning scene
planning_scene_interface.addCollisionObjects(collision_objects);
```

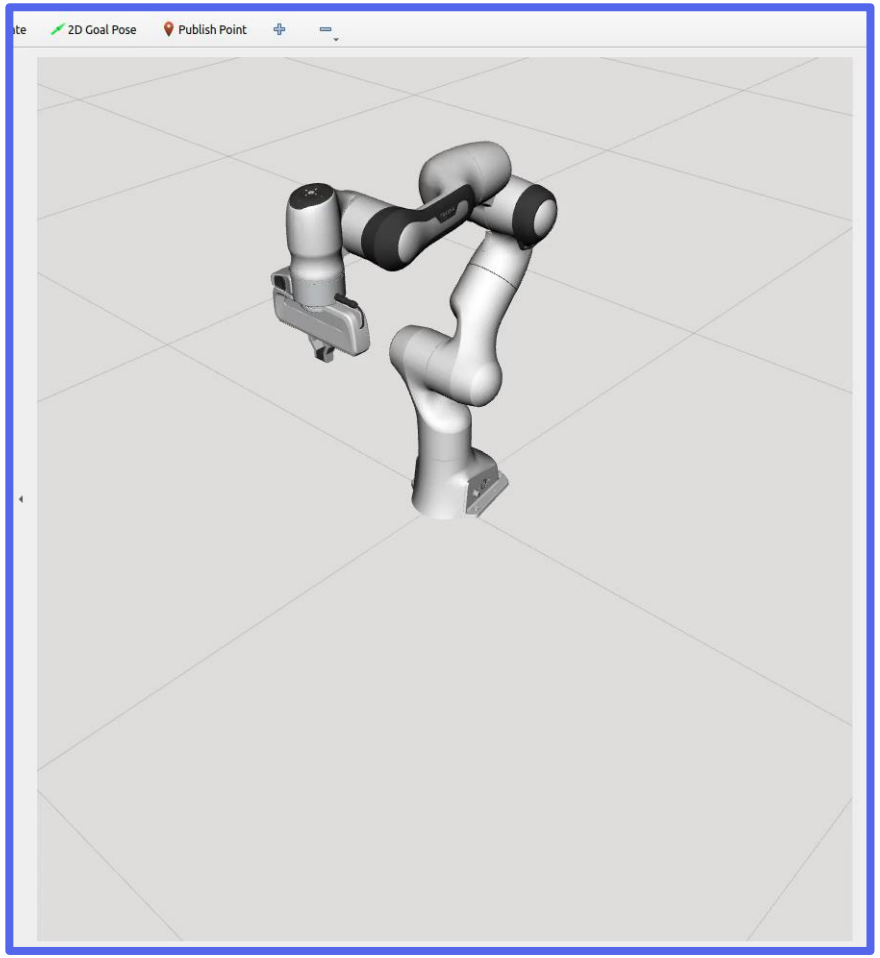


Move Group Interface – Add Mesh Collision Object

```
// Creating an instance of the Mesh Collision object
moveit_msgs::msg::CollisionObject workbench_object;

// Setting the frame ID for the collision object
workbench_object.header.frame_id = "camera_frame_id";

// Assigning an ID to the collision object
workbench_object.id = "workbench";

// Creating a Mesh object from a resource (DAE file)
shapes::Mesh *workbench_import_mesh = shapes::createMeshFromResource("package://description_package/meshes/visual/workbench.dae");

// Converting the Mesh object to a Mesh message
shape_msgs::msg::Mesh workbench_mesh;
shapes::ShapeMsg workbench_mesh_msg;
shapes::constructMsgFromShape(workbench_import_mesh, workbench_mesh_msg);
workbench_mesh = boost::get<shape_msgs::msg::Mesh>(workbench_mesh_msg);

// Adding the Mesh message to the CollisionObject
workbench_object.meshes.push_back(workbench_mesh);

// Adding the pose of the workbench object
workbench_object.mesh_poses.push_back(workbench_collision_detect.obj_pose);

// Setting the operation to ADD, indicating the addition of a new object
workbench_object.operation = workbench_object.ADD;

// Creating a vector to hold the CollisionObjects (in this case, only the workbench)
std::vector<moveit_msgs::msg::CollisionObject> workbench_collision_objects;

// Adding the workbench object to the vector
workbench_collision_objects.push_back(workbench_object);

// Adding the CollisionObjects to the Planning Scene Interface
planning_scene_interface.addCollisionObjects(workbench_collision_objects);
```

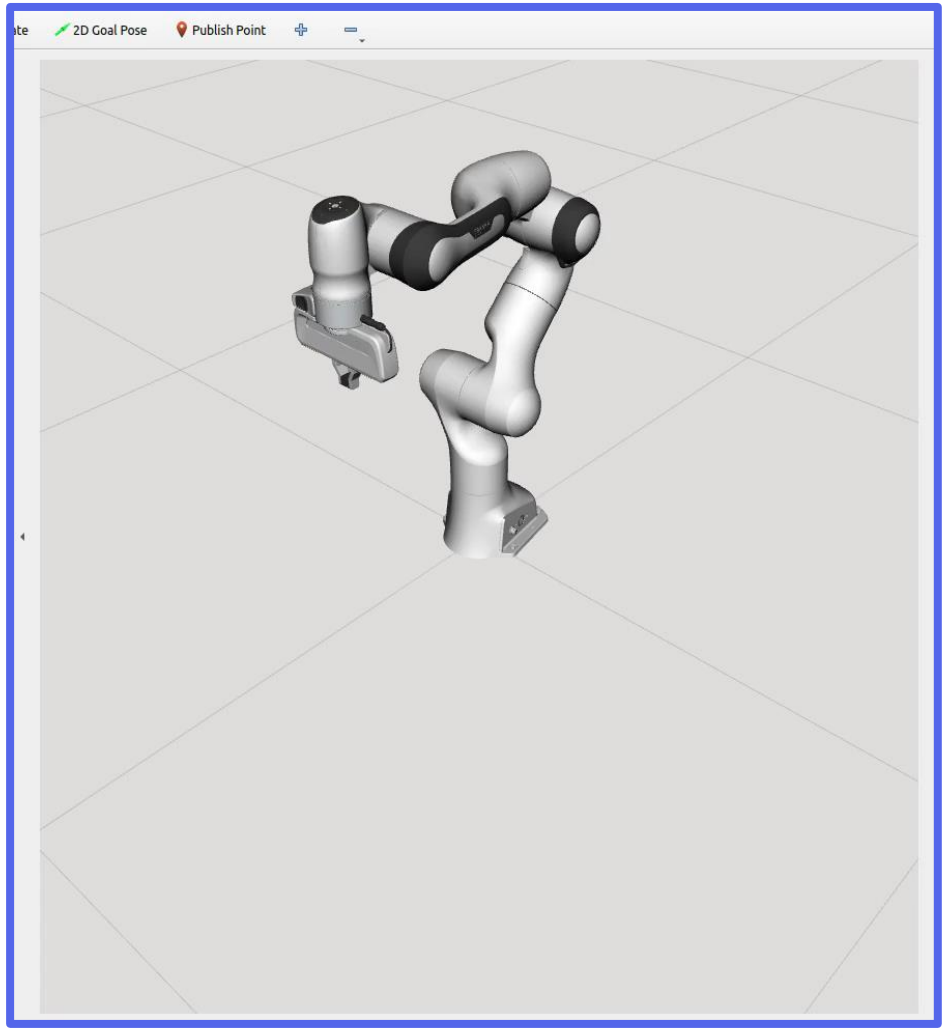
Move Group Interface – Add Mesh Collision Object

```cpp
// Creating an instance of the Mesh Collision object
moveit_msgs::msg::CollisionObject workbench_object;

// Setting the frame ID for the collision object
workbench_object.header.frame_id = "camera_frame_id";

// Assigning an ID to the collision object
workbench_object.id = "workbench";

// Creating a Mesh object from a resource (DAE file)
shapes::Mesh *workbench_import_mesh = shapes::createMeshFromResource("package://description_package/meshes/visual/workbench.dae");

// Converting the Mesh object to a Mesh message
shape_msgs::msg::Mesh workbench_mesh;
shapes::ShapeMsg workbench_mesh_msg;
shapes::constructMsgFromShape(workbench_import_mesh, workbench_mesh_msg);
workbench_mesh = boost::get<shape_msgs::msg::Mesh>(workbench_mesh_msg);

// Adding the Mesh message to the CollisionObject
workbench_object.meshes.push_back(workbench_mesh);

// Adding the pose of the workbench object
workbench_object.mesh_poses.push_back(workbench_collision_detect.obj_pose);

// Setting the operation to ADD, indicating the addition of a new object
workbench_object.operation = workbench_object.ADD;

// Creating a vector to hold the CollisionObjects (in this case, only the workbench)
std::vector<moveit_msgs::msg::CollisionObject> workbench_collision_objects;

// Adding the workbench object to the vector
workbench_collision_objects.push_back(workbench_object);

// Adding the CollisionObjects to the Planning Scene Interface
planning_scene_interface.addCollisionObjects(workbench_collision_objects);
```



Move Group Interface – Attach and Detach Objects

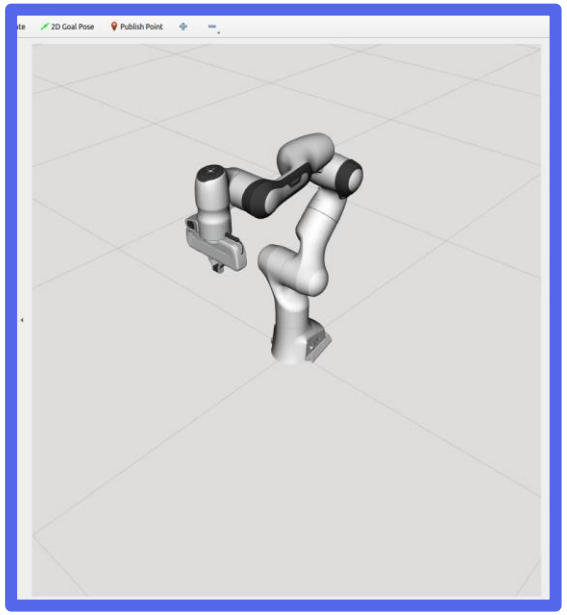
```cpp
// Creating a CollisionObject for a cylinder to be attached
moveit_msgs::msg::CollisionObject object_to_attach;
object_to_attach.id = "cylinder1";

// Creating a SolidPrimitive representing a cylinder
shape_msgs::msg::SolidPrimitive cylinder_primitive;
cylinder_primitive.type = shape_msgs::msg::SolidPrimitive::CYLINDER;
cylinder_primitive.dimensions.resize(2);
cylinder_primitive.dimensions[shape_msgs::msg::SolidPrimitive::CYLINDER_HEIGHT] = 0.20;
cylinder_primitive.dimensions[shape_msgs::msg::SolidPrimitive::CYLINDER_RADIUS] = 0.04;

// Setting the frame ID for the CollisionObject
object_to_attach.header.frame_id = move_group.getEndEffectorLink();

// Setting the pose for attaching the cylinder
geometry_msgs::msg::Pose grab_pose;
grab_pose.orientation.w = 1.0;
grab_pose.position.z = 0.22;

// Adding the cylinder primitive and grab pose to the CollisionObject
object_to_attach.primitives.push_back(cylinder_primitive);
object_to_attach.primitive_poses.push_back(grab_pose);

// Setting the operation to ADD, indicating the addition of a new object
object_to_attach.operation = object_to_attach.ADD;

// Applying the CollisionObject to the planning scene
planning_scene_interface.applyCollisionObject(object_to_attach);

// Defining a vector of touch links for attaching the object
std::vector<std::string> touch_links;
touch_links.push_back("panda_rightfinger");
touch_links.push_back("panda_leftfinger");

// Attaching the object to the robot's hand with specified touch links
move_group.attachObject(object_to_attach.id, "panda_hand", touch_links);

// Detaching the object from the robot's hand using the object id
move_group.detachObject(object_to_attach.id);
```

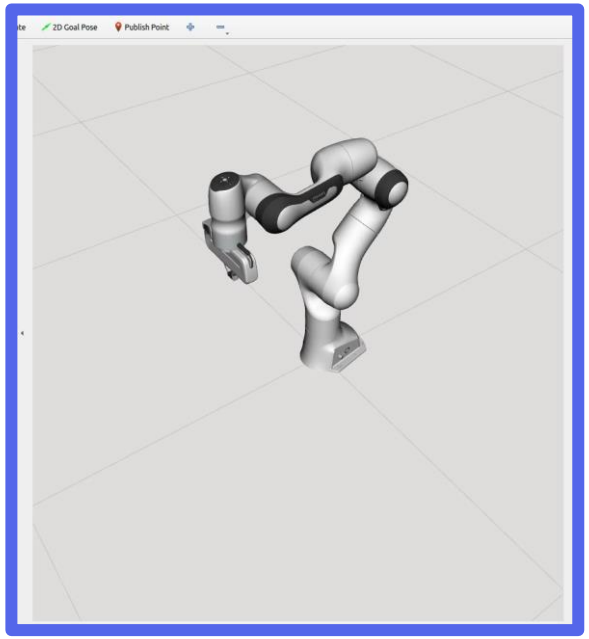


interface_guide

Move Group Interface – Add Workspace and Position Constraints

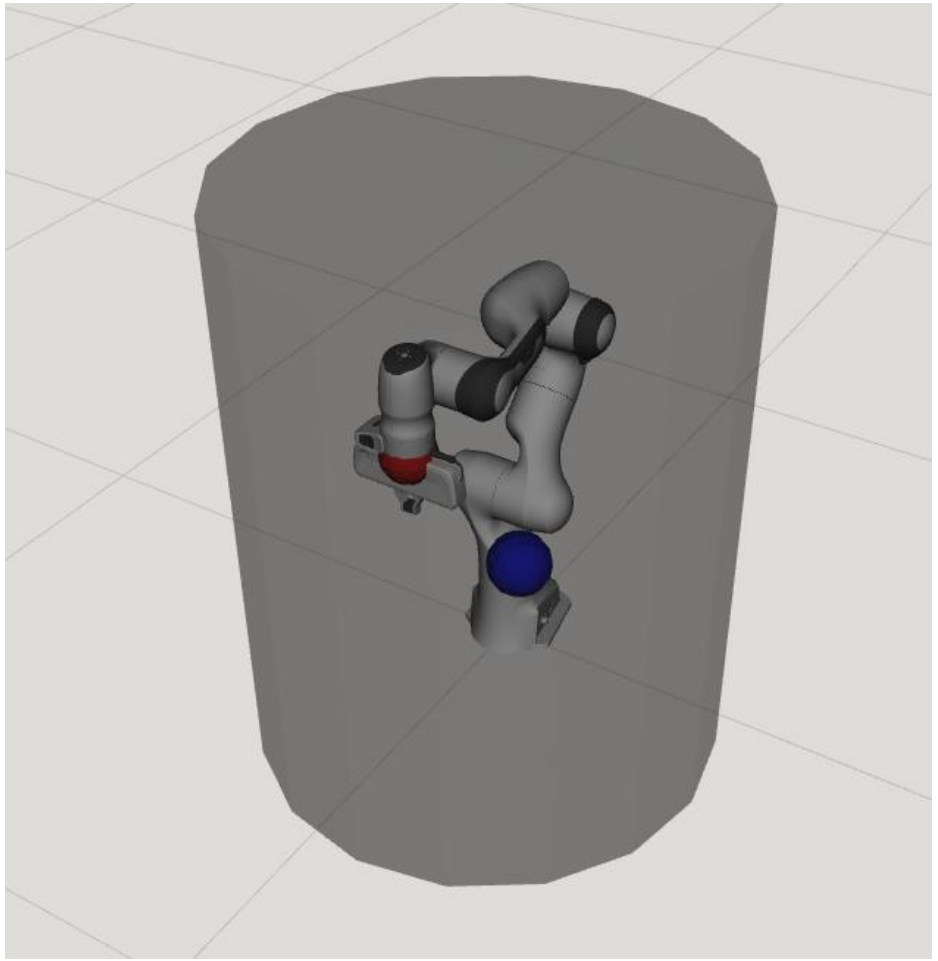
```cpp
// Creating a PositionConstraint for a box in the planning scene
moveit_msgs::msg::PositionConstraint box_constraint;

// Setting the frame ID for the box constraint
box_constraint.header.frame_id = move_group.getPoseReferenceFrame();

// Setting the end effector link for the box constraint
box_constraint.link_name = move_group.getEndEffectorLink();

// Creating a SolidPrimitive representing a box
shape_msgs::msg::SolidPrimitive box;
box.type = shape_msgs::msg::SolidPrimitive::BOX;
box.dimensions = { 0.8, 0.8, 0.8 };
box_constraint.constraint_region.primitives.emplace_back(box);

// Setting the pose of the box in the constraint region
geometry_msgs::msg::Pose box_pose;
box_pose.position.x = current_pose.pose.position.x;
box_pose.position.y = 0.15;
box_pose.position.z = current_pose.pose.position.z;
box_pose.orientation.w = 1.0;
box_constraint.constraint_region.primitive_poses.emplace_back(box_pose);

// Setting the weight for the constraint (1.0 for full importance)
box_constraint.weight = 1.0;

// Creating Constraints message to hold the position constraint
moveit_msgs::msg::Constraints constraints_list;

// Adding the box constraint to the position constraints in the Constraints message
constraints_list.position_constraints.emplace_back(box_constraint);

// Setting constraints with the listed constraints
move_group.setPathConstraints(constraints_list);
```





Move Group Interface – Workspace (Cylinder) Constraints

```cpp
// Creating a PositionConstraint for a cylinder in the planning scene
moveit_msgs::msg::PositionConstraint cylinder_constraint;

// Setting the frame ID for the cylinder constraint
cylinder_constraint.header.frame_id = move_group.getPoseReferenceFrame();

// Setting the end effector link for the cylinder constraint
cylinder_constraint.link_name = move_group.getEndEffectorLink();

// Creating a SolidPrimitive representing a cylinder
shape_msgs::msg::SolidPrimitive cylinder;
cylinder.type = shape_msgs::msg::SolidPrimitive::CYLINDER;
cylinder.dimensions = { 1.0, 0.8 };  // Radius and height of the cylinder
cylinder_constraint.constraint_region.primitives.emplace_back(cylinder);

// Setting the pose of the cylinder in the constraint region
geometry_msgs::msg::Pose cylinder_pose;
cylinder_pose.position.x = current_pose.pose.position.x;
cylinder_pose.position.y = 0.15;
cylinder_pose.position.z = current_pose.pose.position.z;
cylinder_pose.orientation.w = 1.0;
cylinder_constraint.constraint_region.primitive_poses.emplace_back(cylinder_pose);

// Setting the weight for the constraint (1.0 for full importance)
cylinder_constraint.weight = 1.0;


// Creating Constraints message to hold the position constraint
moveit_msgs::msg::Constraints constraints_list;

// Adding the cylinder constraint to the position constraints in the Constraints message
constraints_list.position_constraints.emplace_back(cylinder_constraint);

// Setting constraints with the listed constraints
move_group.setPathConstraints(constraints_list);
```



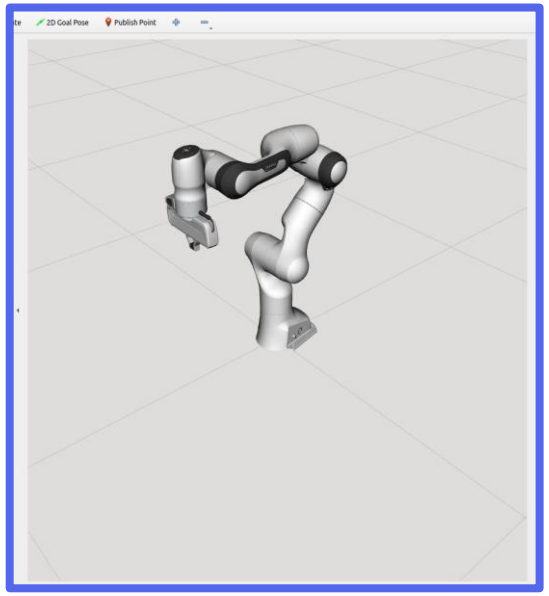
Move Group Interface – Add Orientation Constraints

```
// Creating an OrientationConstraint for the end effector in the planning scene
moveit_msgs::msg::OrientationConstraint orientation_constraint;

// Setting the frame ID for the orientation constraint
orientation_constraint.header.frame_id = move_group.getPoseReferenceFrame();

// Setting the end effector link for the orientation constraint
orientation_constraint.link_name = move_group.getEndEffectorLink();

// Setting the desired orientation for the end effector
orientation_constraint.orientation = current_pose.pose.orientation;

// Setting absolute tolerances for each axis in radians
orientation_constraint.absolute_x_axis_tolerance = 0.5;
orientation_constraint.absolute_y_axis_tolerance = 0.5;
orientation_constraint.absolute_z_axis_tolerance = 0.5;

// Setting the weight for the constraint (1.0 for full importance)
orientation_constraint.weight = 1.0;

// Creating Constraints message to hold the position constraint
moveit_msgs::msg::Constraints constraints_list;

// Adding the orientation constraint to the position constraints in the Constraints messag
constraints_list.orientation_constraints.emplace_back(orientation_constraint);

// Setting constraints with the listed constraints
move_group.setPathConstraints(constraints_list);
```

Move Group Interface – Add Mixed Constraints

```
// Creating Constraints message to hold the position constraint
moveit_msgs::msg::Constraints constraints_list;

// Adding the position and orientation constraint to the position constraints in the Constraints message
constraints_list.position_constraints.emplace_back(box_constraint);
constraints_list.orientation_constraints.emplace_back(orientation_constraint);

// Setting constraints with the listed constraints
move_group.setPathConstraints(constraints_list);
```

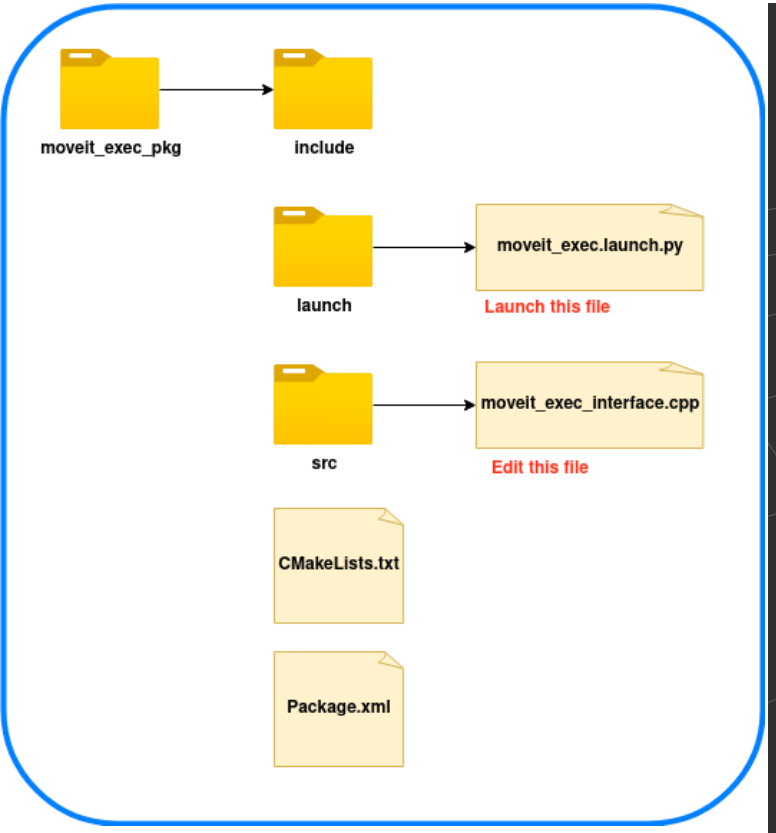
Tight Tolerance

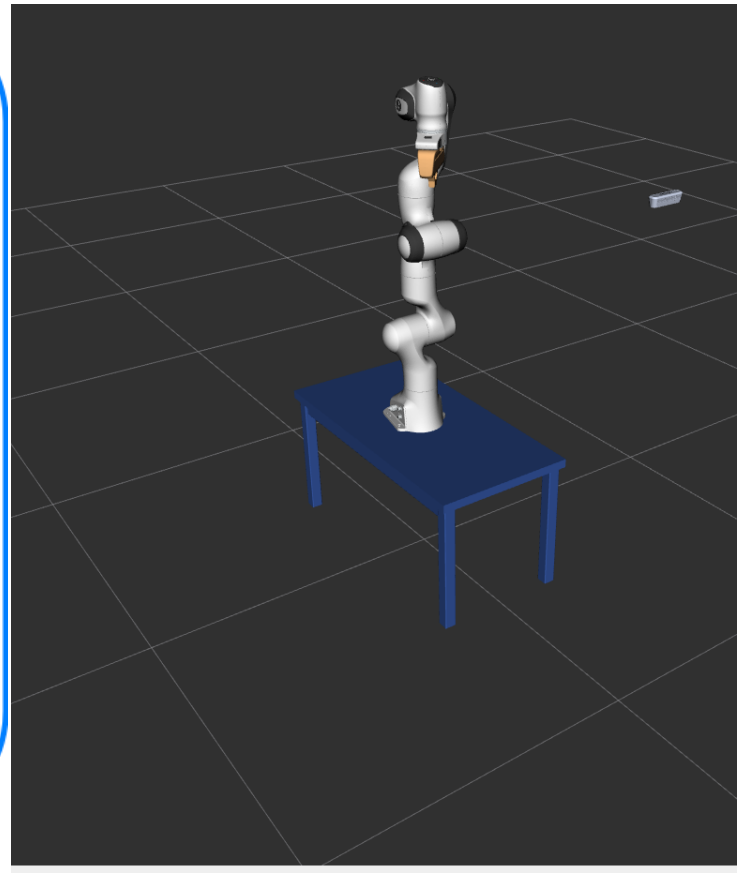


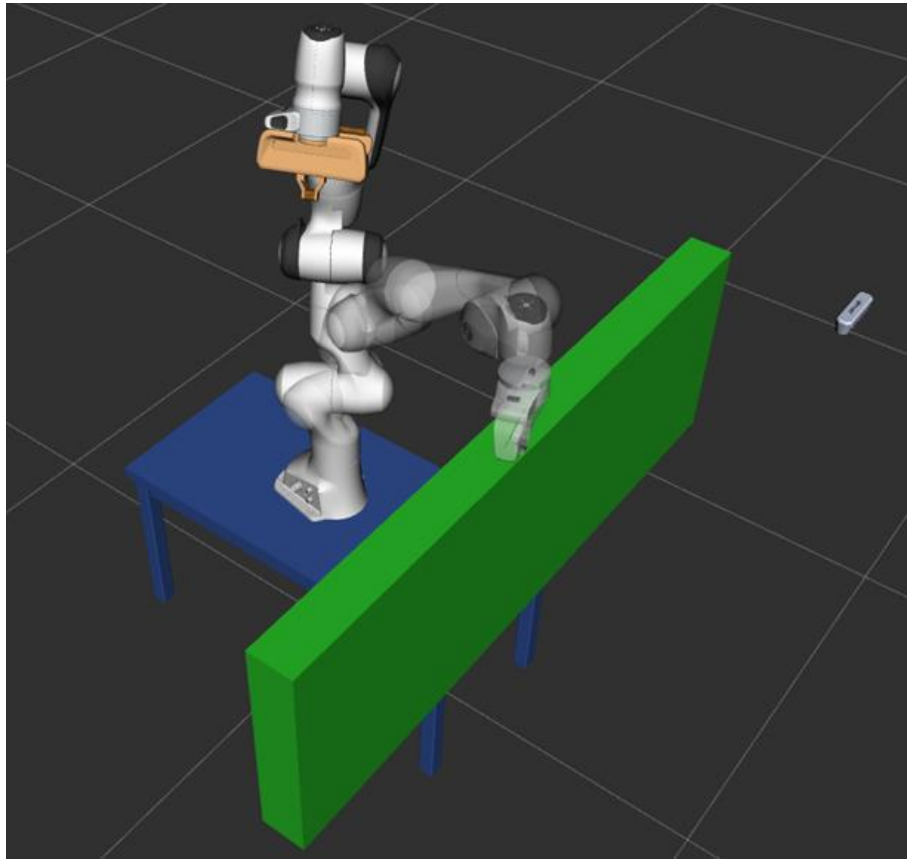
Comparatively More Tolerance

Exercise 3: Use Move Group and Move Group Interface to execute set of motion plan tasks

1.Pose Goal and Plan
2.Add Collision Object and Plan
3.Joint Goal and Plan







ROS-Industrial Manipulation Training - Agenda Day 1

What is Manipulation?
- Concepts
- Applications

Robot Description
- URDF
  - Package Content
  - URDF Components
  - Exercise 1: Visualize Robot Description by adding table and camera

Moveit Setup Assistant
  - Setup Components
  - ROS2 Controllers [High-Level]
  - Moveit Controllers [High-Level]
  - Exercise 2: Create Moveit Config and Launch

Move Group
  - Move Group Capabilities
  - Exercise 3: Motion Planning with Move Group

Perception with Camera and ARUCO Marker
  - Exercise 4: ARUCO Detection and Spawn Collision Object
  - Xacro [High-Level]

Perception with Gazebo Simulation [Take Home (optional)]
  - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo

Advanced Remanufacturing and Technology Centre

ARTC

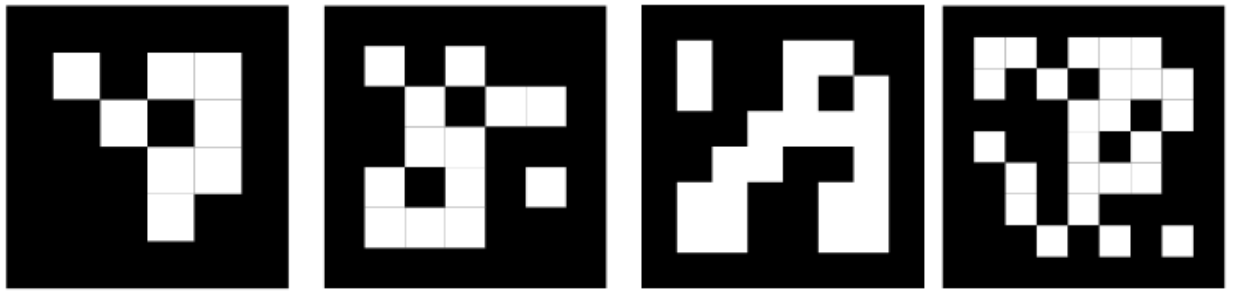
PERCEPTION WITH CAMERA AND ARUCO MARKER

Shalman Khan

4th December 2023

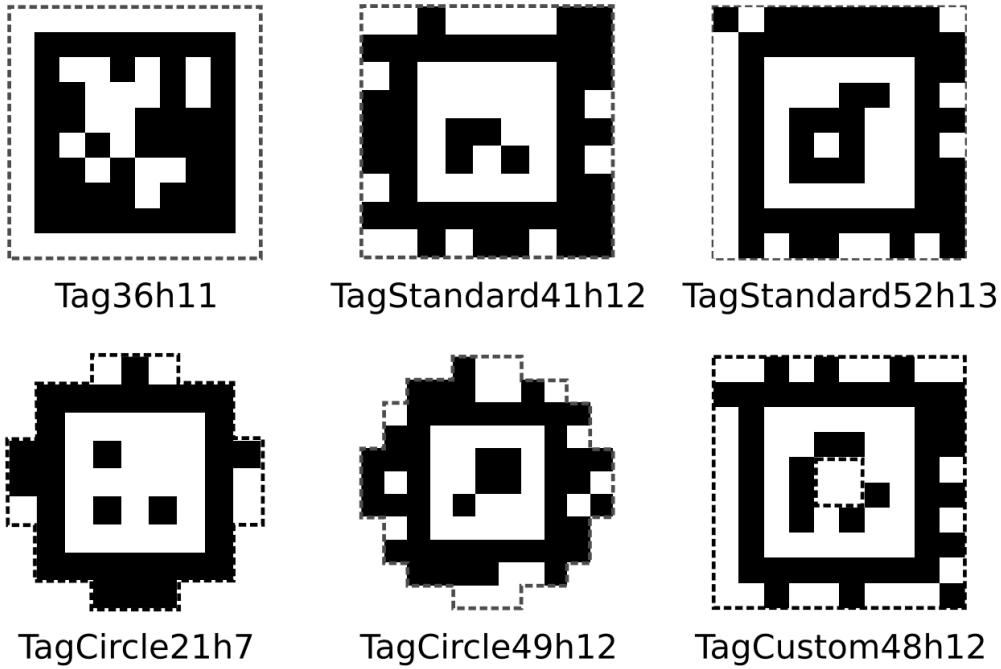
AR Tag Detection – ArUco Marker + April Tag

ArUco Marker



4 x4 5x5 6x6 7x7

April Tag



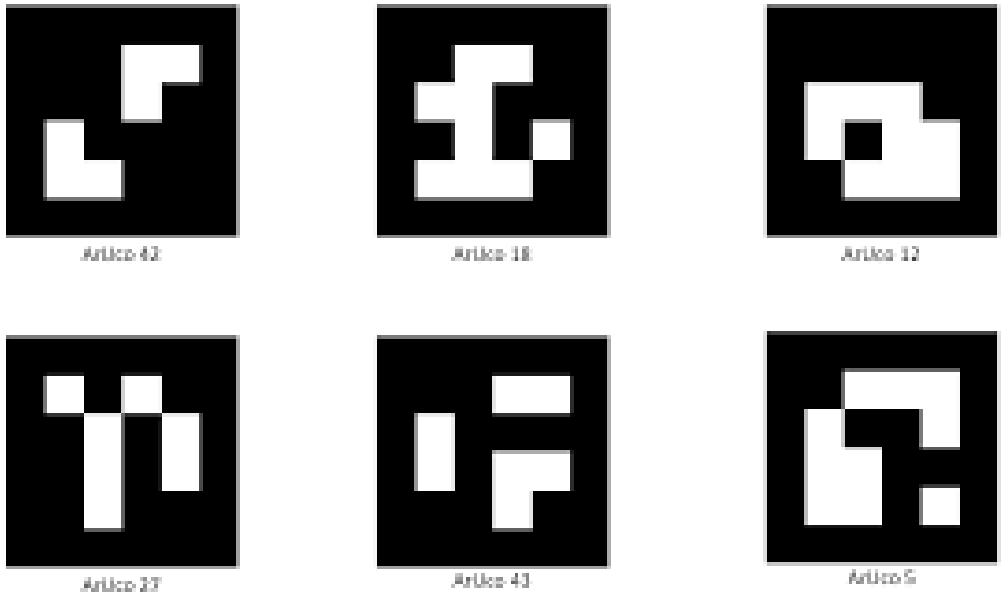
Tag36h11 TagStandard41h12 TagStandard52h13

TagCircle21h7 TagCircle49h12 TagCustom48h12

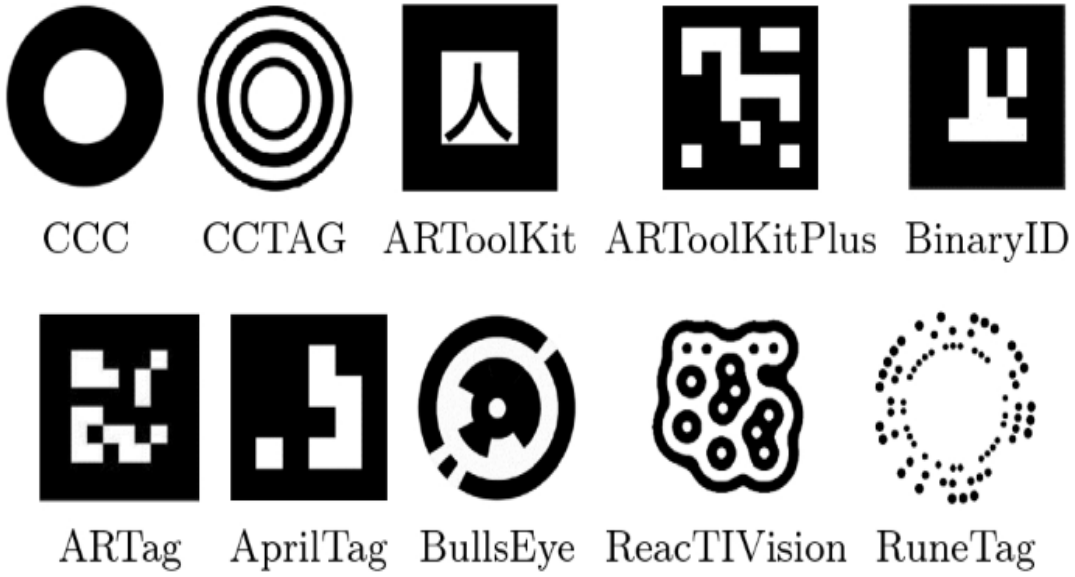
Link 1

Link 2

AR Tag Detection – ID and Other Markers

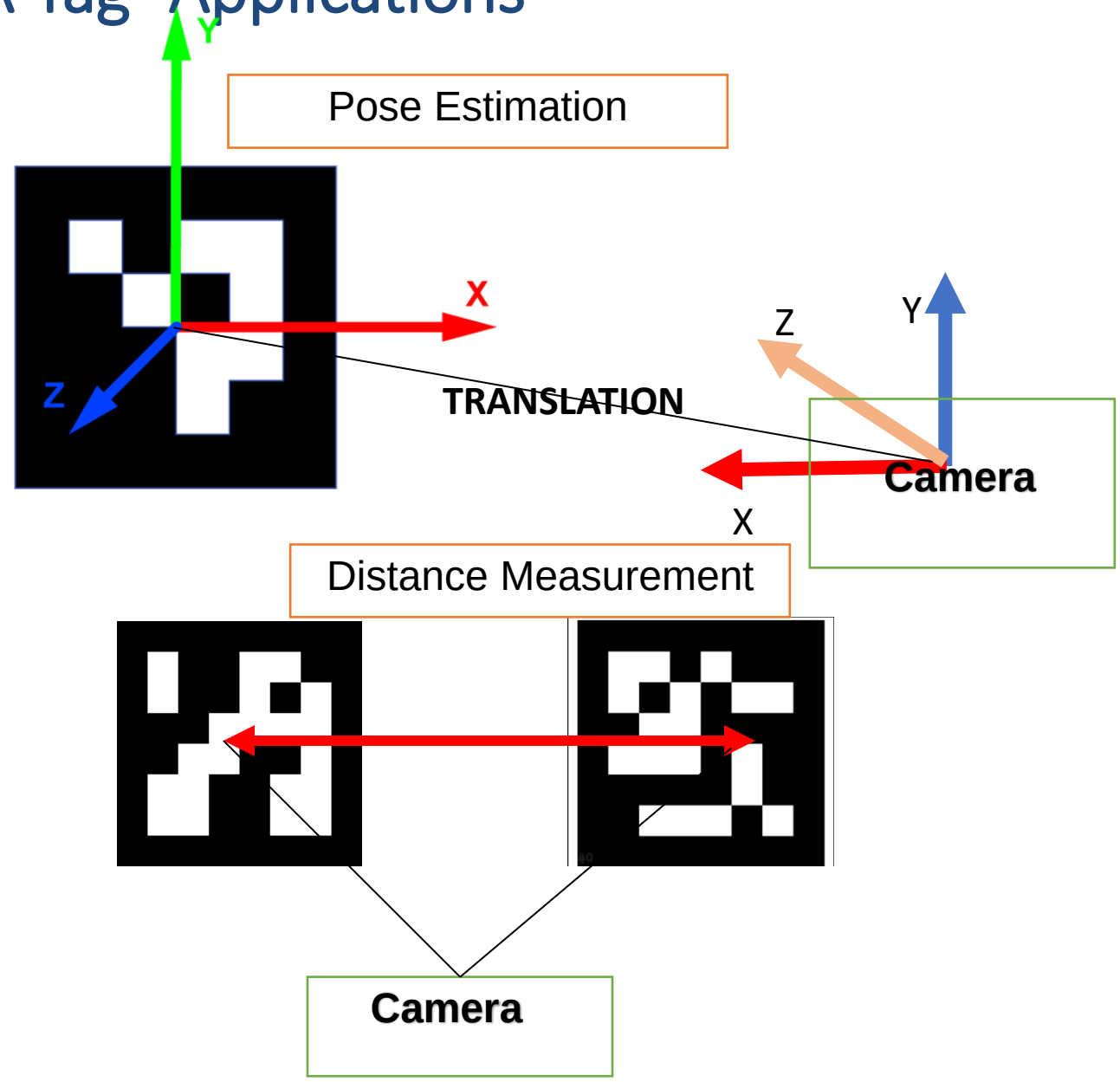


ArUco Marker – Different IDs



AR Tags [Others]

AR Tag- Applications



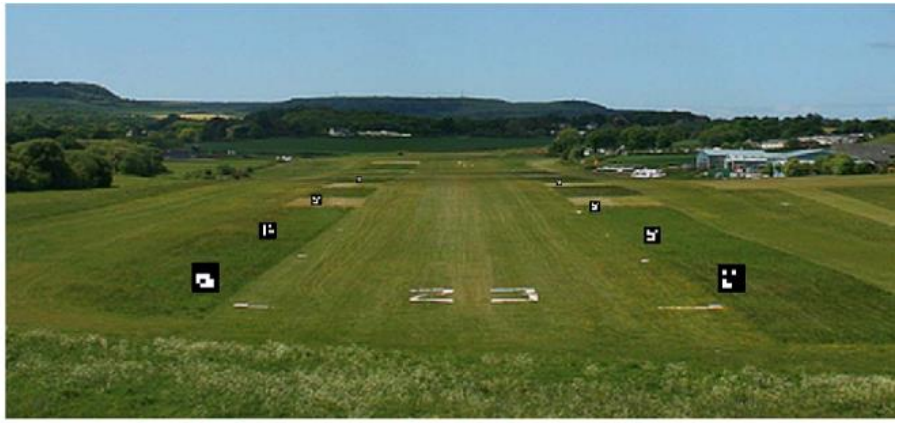
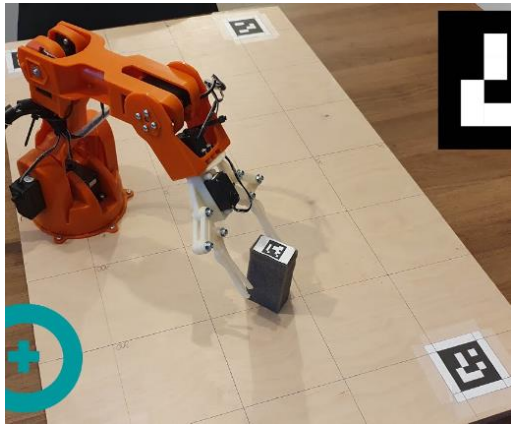
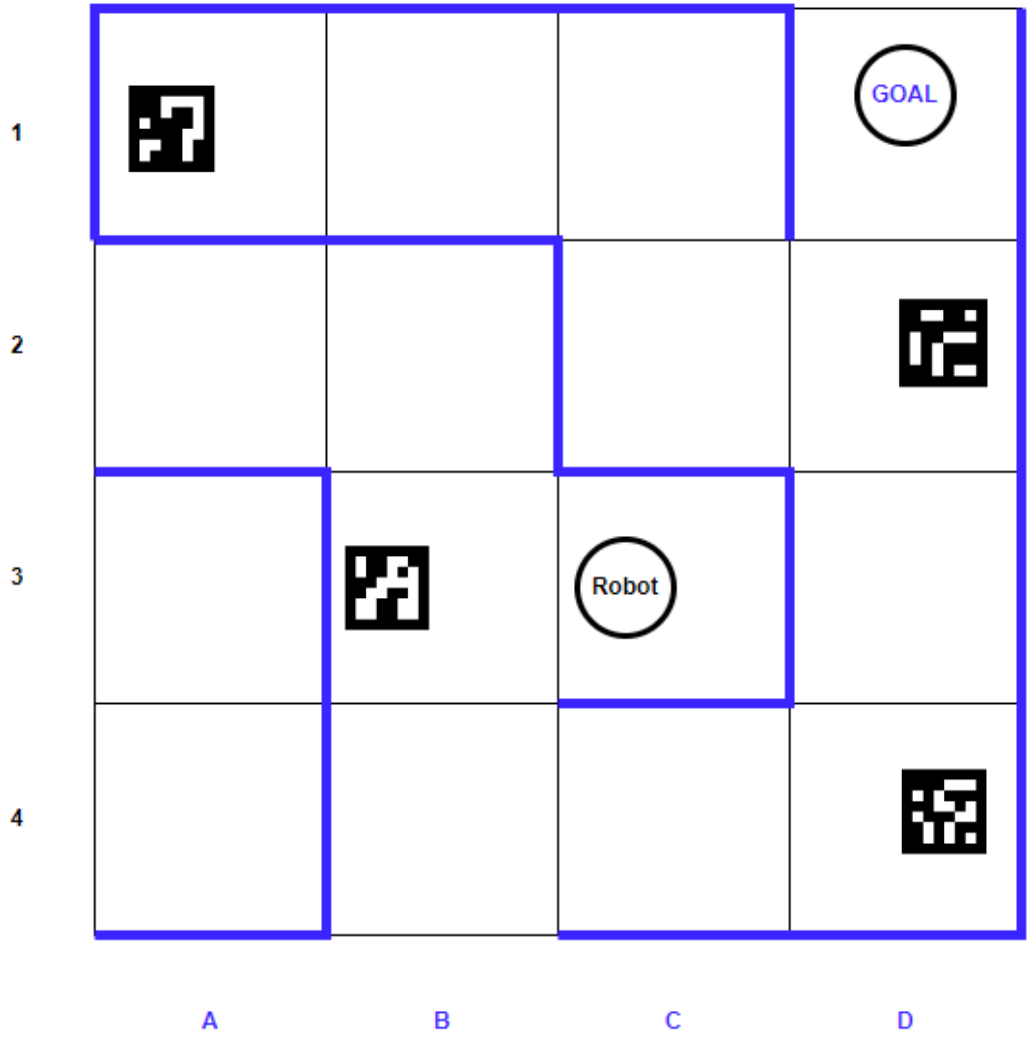
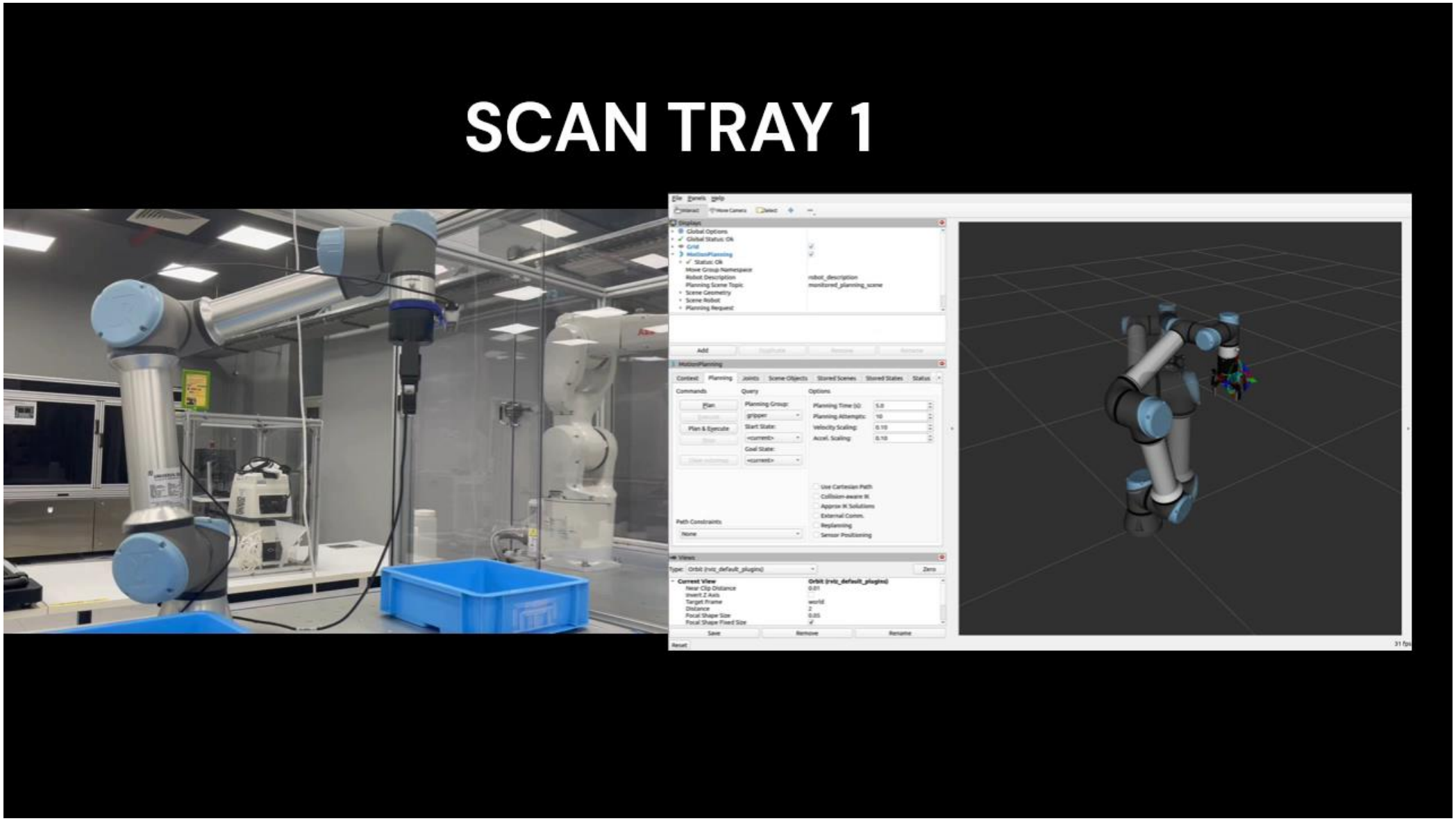


Figure 3. Airfield infrastructure for ArUco based landing aid system



AR Tag- Localization

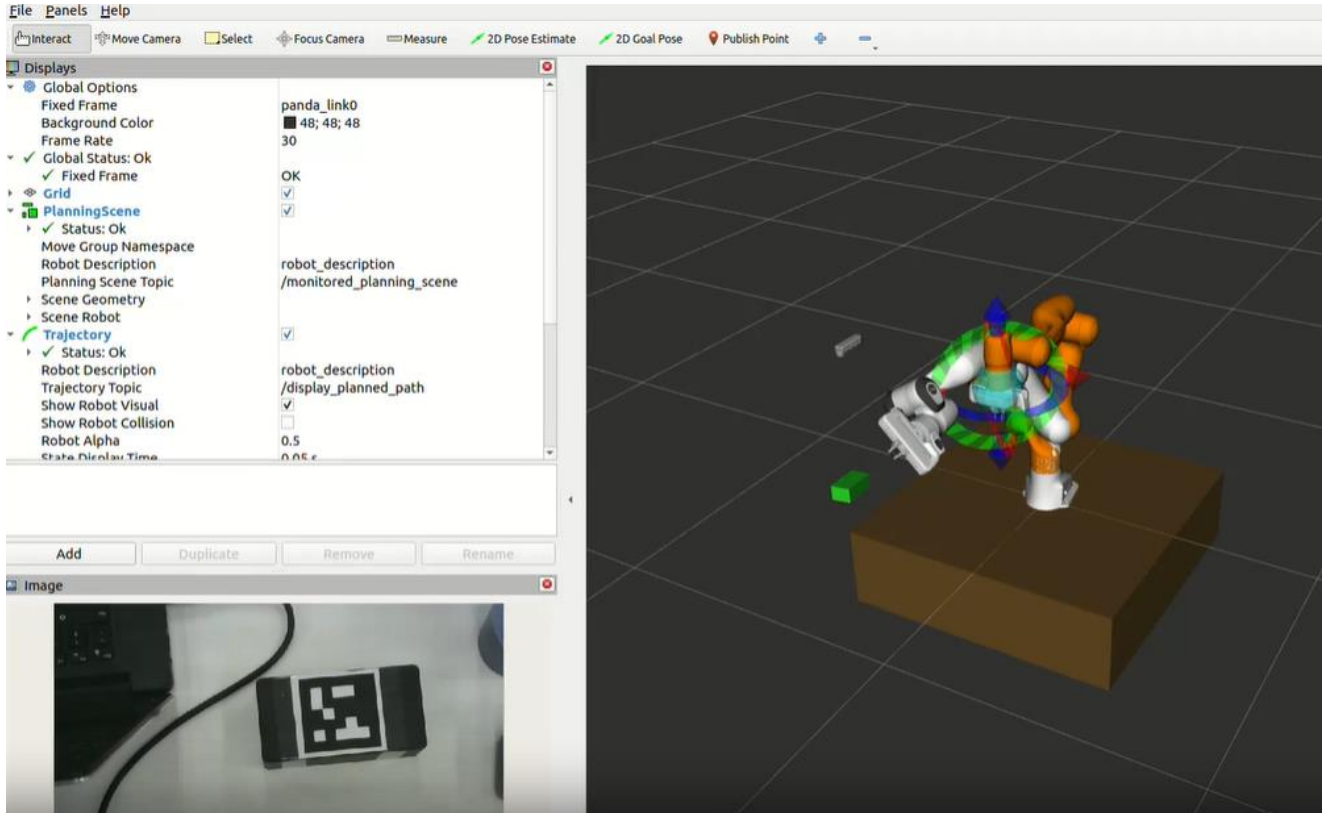



SCAN TRAY 1

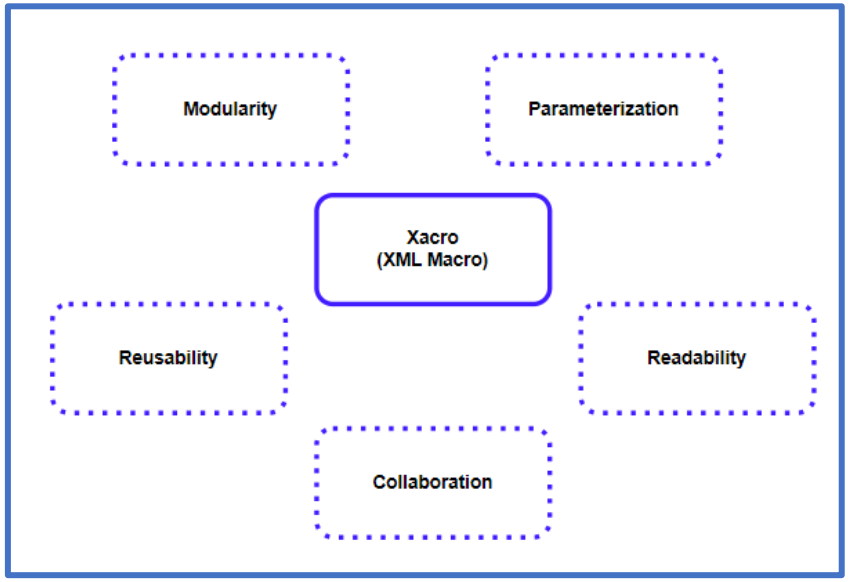
Exercise 4: ARUCO Detection and Spawn Collision Object

There are three objectives to this exercise

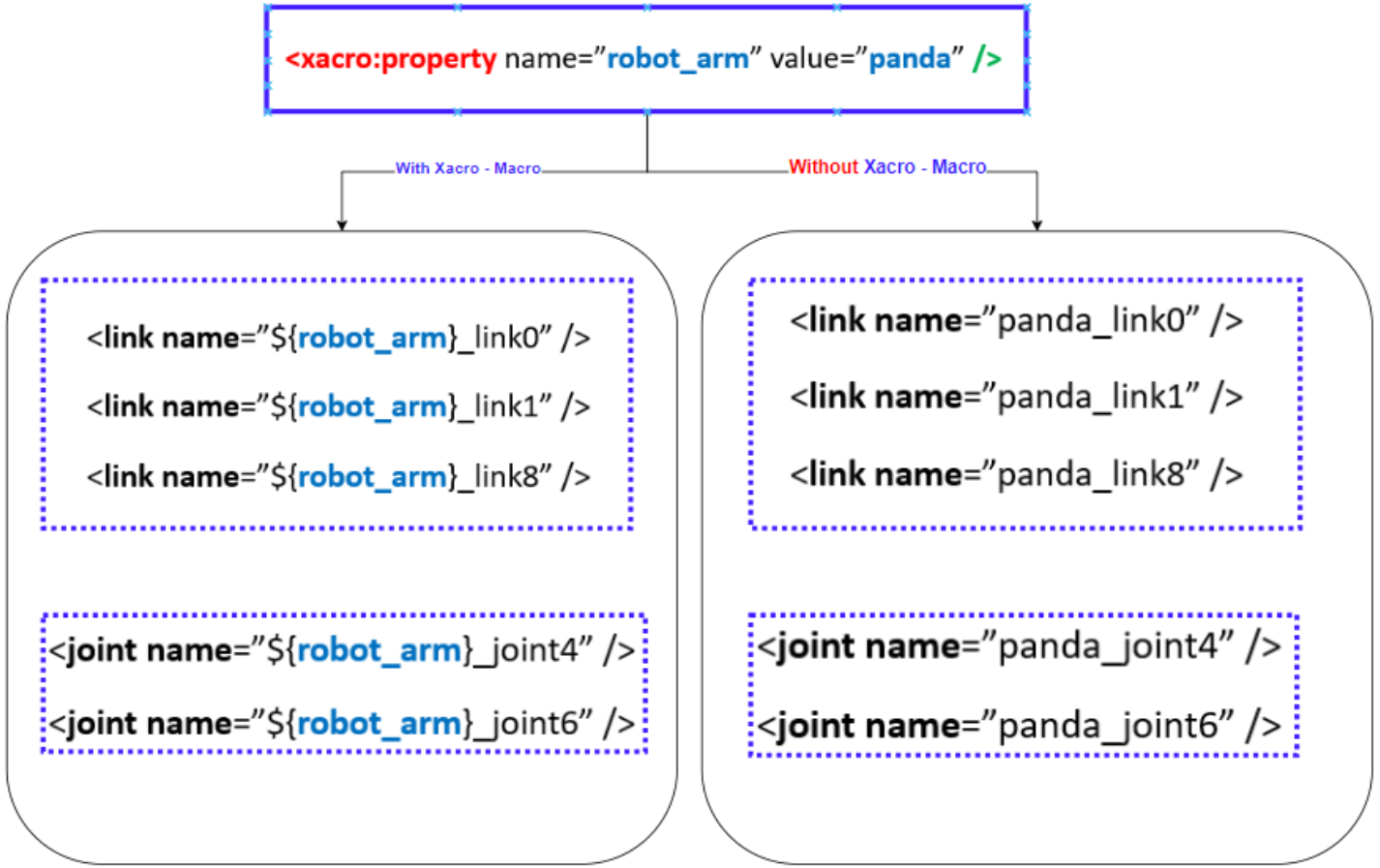
1. **Write a client request function to get ARUCO Marker Position**
2. **Call the detect_collision_object to get Marker pose**
3. **Create a box for following dimensions and use the detected pose as box pose**



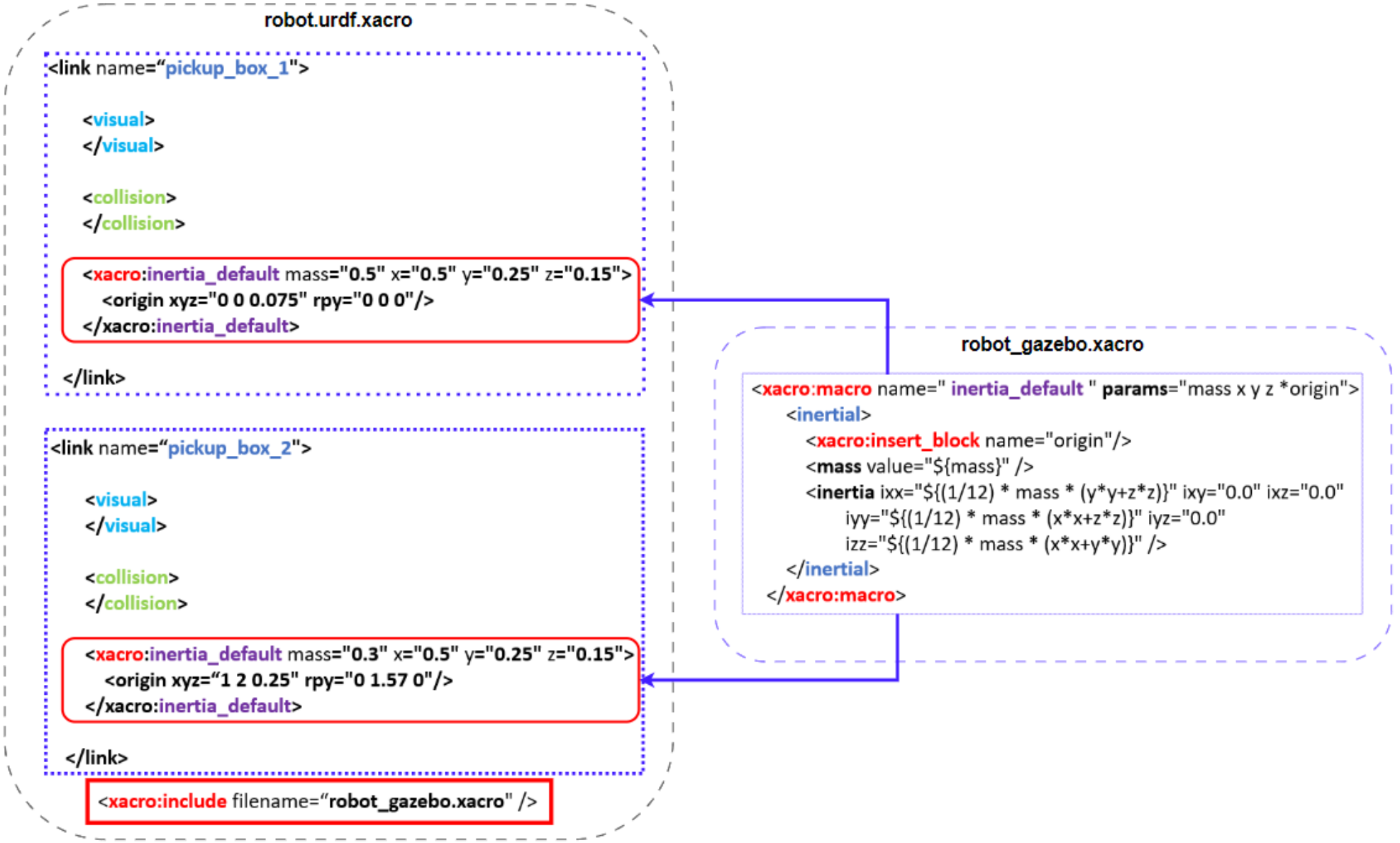
Brief Introduction to Xacro




```
<origin xyz="0.8 0.3  ${Table_Height*(1.0)}" />
```

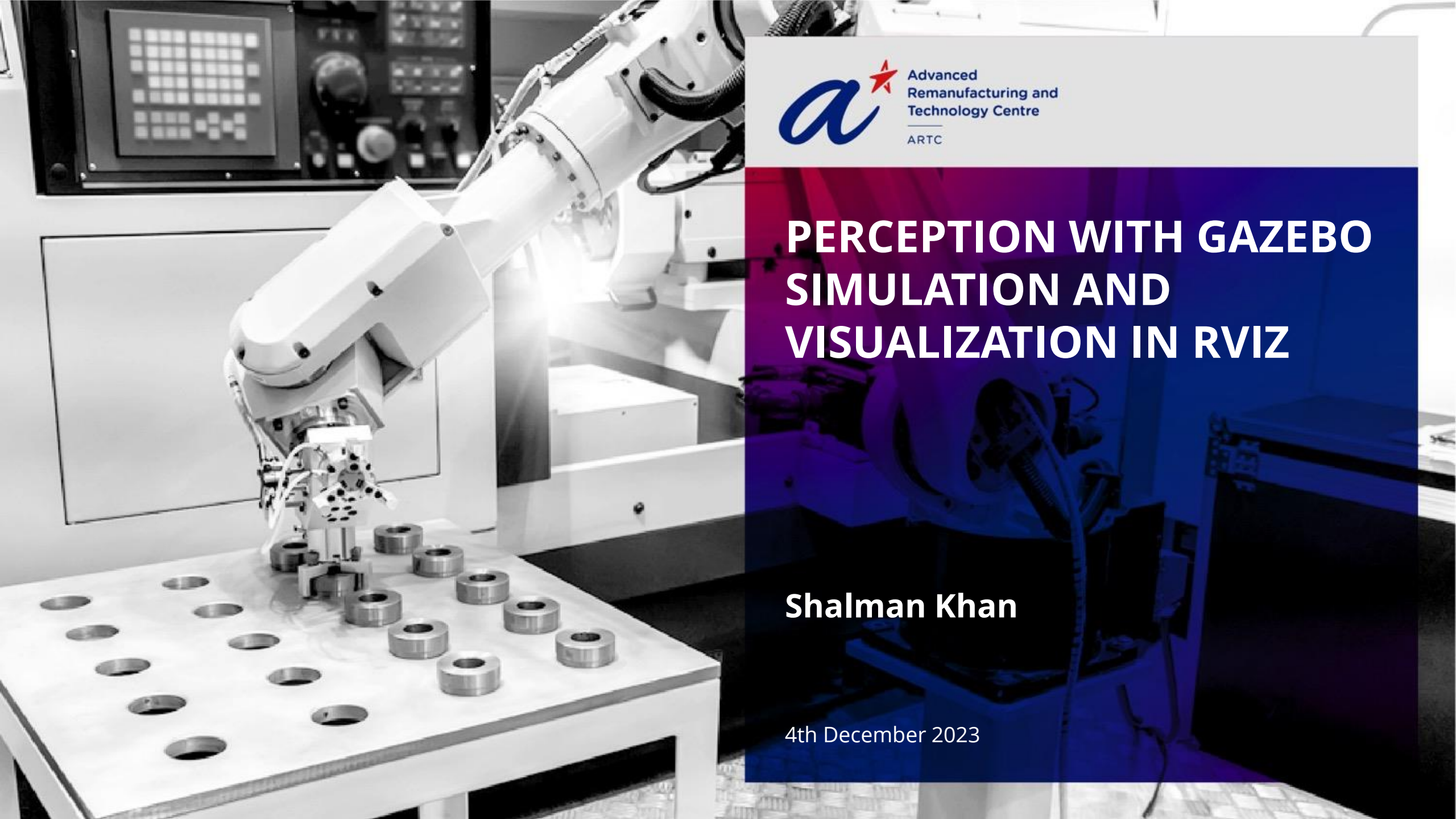


Brief Introduction to Xacro



ROS-Industrial Manipulation Training - Agenda Day 1

- What is Manipulation?
  - Concepts
  - Applications
- Robot Description
  - URDF
    - Package Content
    - URDF Components
    - Exercise 1: Visualize Robot Description by adding table and camera
- Moveit Setup Assistant
    - Setup Components
    - ROS2 Controllers [High-Level]
    - Moveit Controllers [High-Level]
    - Exercise 2: Create Moveit Config and Launch
- Move Group
    - Move Group Capabilities
    - Exercise 3: Motion Planning with Move Group
- Perception with Camera and ARUCO Marker
    - Xacro [High-Level]
    - Exercise 4: ARUCO Detection and Spawn Collision Object
- Perception with Gazebo Simulation [Take Home (optional)]
    - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo



Advanced Remanufacturing and Technology Centre
ARTC

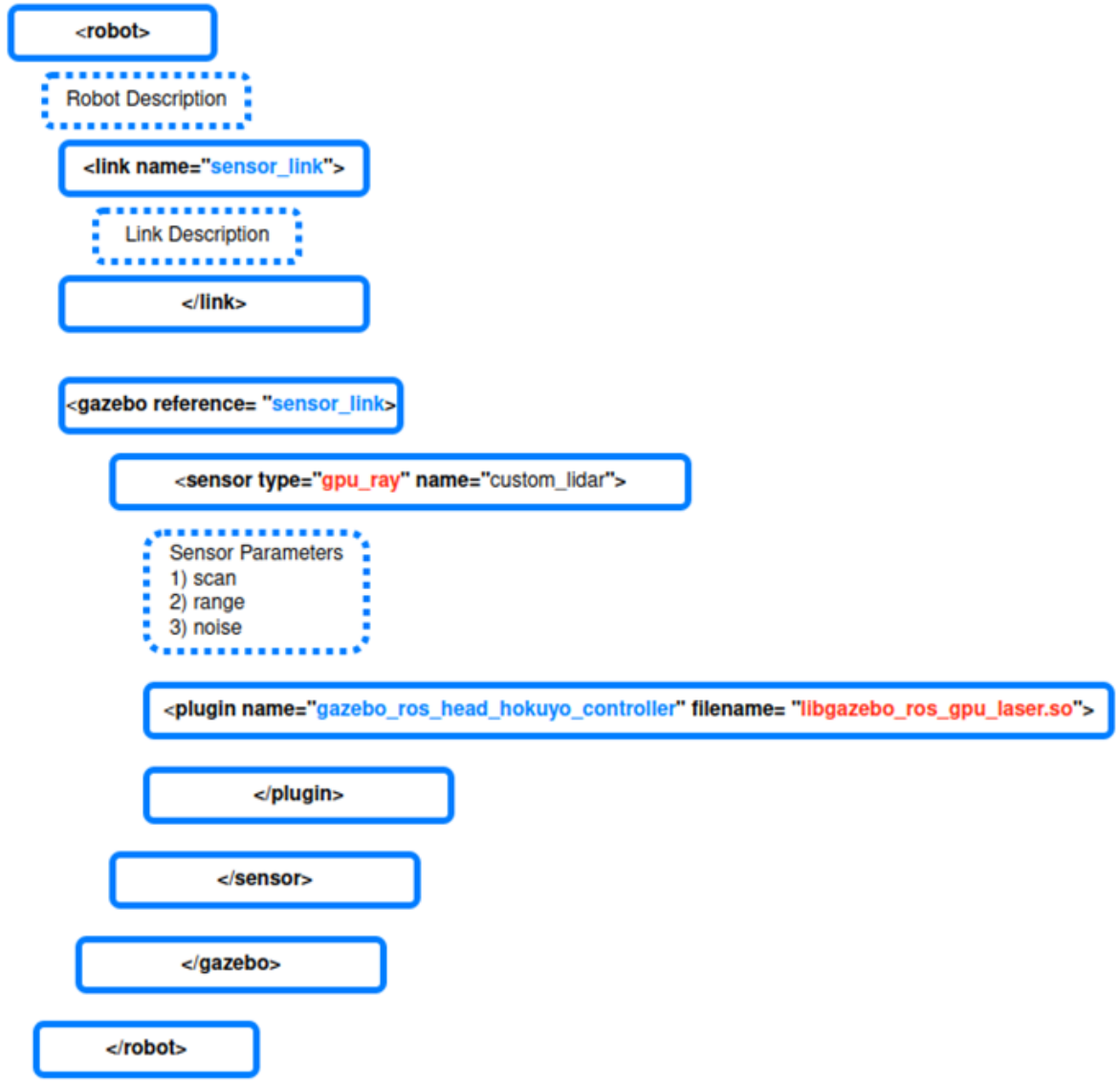
PERCEPTION WITH GAZEBO SIMULATION AND VISUALIZATION IN RVIZ

Shalman Khan

4th December 2023

Perception with Gazebo and RViz – Introduction to Gazebo Sensor Plugins

Structure of URDF with Gazebo Sensor Plugins




```
<!-- hokuyo -->
<gazebo reference="hokuyo_link">
  <sensor type="gpu_ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>false</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.570796</min_angle>
          <max_angle>1.570796</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <!-- Noise parameters based on published spec for Hokuyo laser
             achieving "+-30mm" accuracy at range < 10m.  A mean of 0.0m and
             stddev of 0.01m will put 99.7% of samples within 0.03m of the true
             reading. -->
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_gpu_laser.so">
      <topicName>/rrbot/laser/scan</topicName>
      <frameName>hokuyo_link</frameName>
    </plugin>
  </sensor>
</gazebo>
```

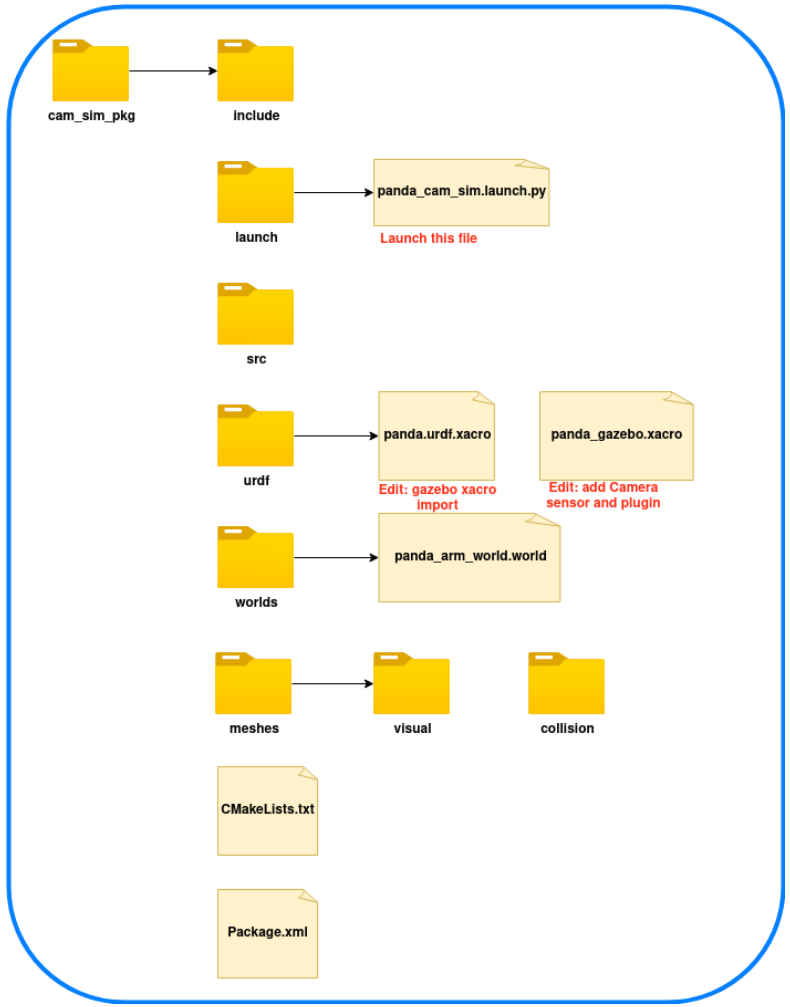
Plugin_integration Link

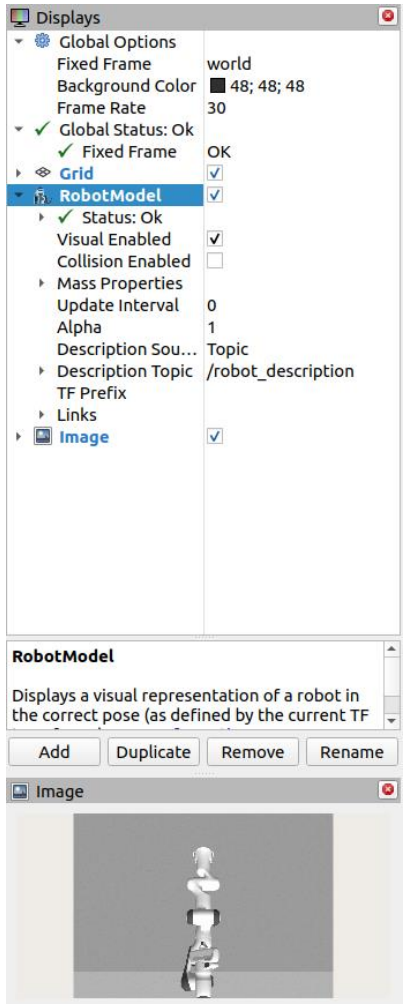
Plugin Link

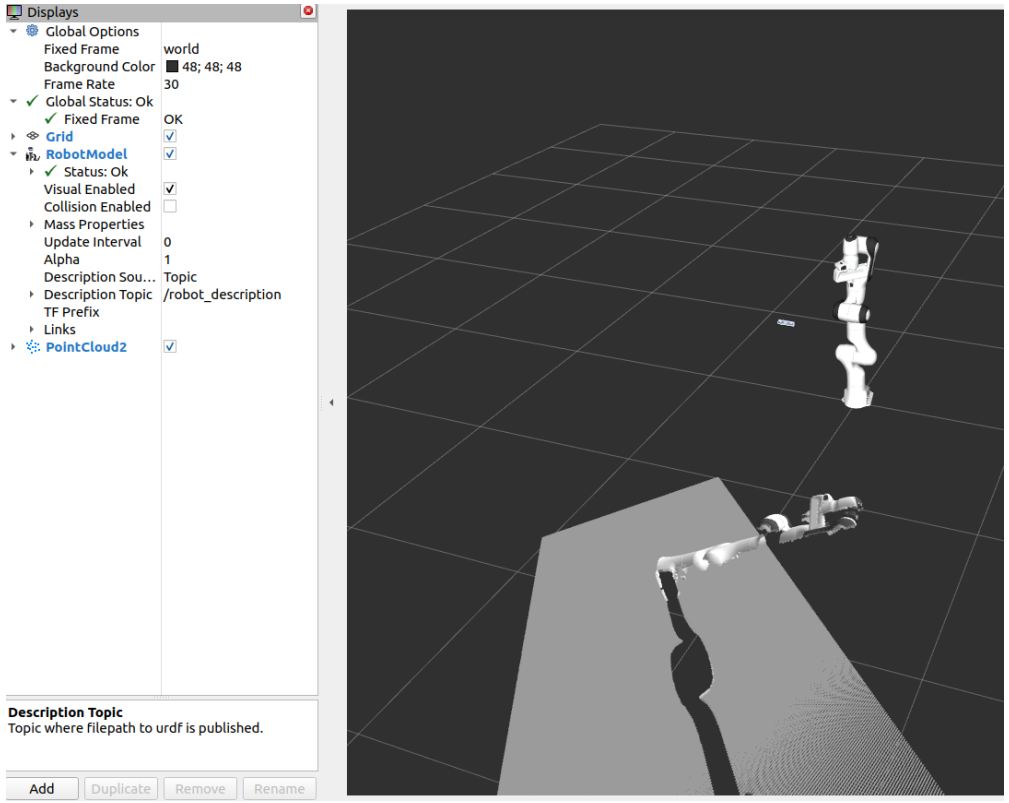
Exercise 5: Visualize image (and) 3D point cloud using camera plugin in Gazebo and RViz

There are three objectives to this exercise

1. **Add Camera Plugin in Gazebo Xacro file**
2. **Include Gazebo Xacro in URDF Xacro file**
3. **Visualize 3D Point cloud and Image**







ROS-Industrial Manipulation Training - Agenda Day 1

- What is Manipulation?
  - Concepts
  - Applications
- Robot Description
  - URDF
    - Package Content
    - URDF Components
    - Exercise 1: Visualize Robot Description by adding table and camera
- Moveit Setup Assistant
    - Setup Components
    - ROS2 Controllers [High-Level]
    - Moveit Controllers [High-Level]
    - Exercise 2: Create Moveit Config and Launch
- Move Group
    - Move Group Capabilities
    - Exercise 3: Motion Planning with Move Group
- Perception with Camera and ARUCO Marker
    - Xacro [High-Level]
    - Exercise 4: ARUCO Detection and Spawn Collision Object
- Perception with Gazebo Simulation [Take Home (optional)]
    - Bonus Exercise [Optional]: Visualize Image and Point Cloud with RViz and Gazebo