Final Project Report (Group 2)

ENPM 663: Building a Manufacturing Robotic Software System

Abraruddin Syed (UID:120109997)
Dhruv Sharma (UID:119091586)
Mayank Sharma (UID: 119203859)
Yashveer Jain (UID:119252864)

Contents

1 Introduction

The Agile Robotics for Industrial Automation Competition (ARIAC) Agility Challenge, organized by the National Institute of Standards and Technology (NIST), is an exciting competition that evaluates robots' capabilities in completing various tasks within a constantly changing environment. The tasks are designed to assess the robot's proficiency in executing pick-and-place operations, assembly tasks, and kitting processes within a virtual warehouse setting. This competition serves as a valuable platform for refining algorithms that have practical applications in real-life manufacturing scenarios.

The ARIAC environment consists of various parts (Pumps, Sensors, Battery, Regulator), trays, and AGVs for shipping the orders. This time, the projects were based on handling the kitting tasks, which involved leveraging the sensors and floor robot to place parts on the tray in the specified quadrant while handling the agility challenges and shipping the order to the warehouse. Some of the challenges that were introduced to test competitor control system (CCS) agility are:

- **High Priority order:** A priority level is applied to each order. When a new, high-priority order is announced while the robots have been working on a low-priority order, they must start focusing on the high-priority order swiftly and then return to the low-priority order.
- **Faulty Parts:** These are the parts that are not in acceptable condition for the order to be completed. In such a scenario, the part has to be discarded and a new part of the same type and color is picked to complete the order.
- **Faulty gripper:** The part picked by the gripper can be dropped at any time during the trial, also referred to as the dropped part challenge. This situation can happen even if the manipulator is not moving. The goal is to test the ability of CCS to check the dropped part and to pick the part of the same type and color if available in the environment.

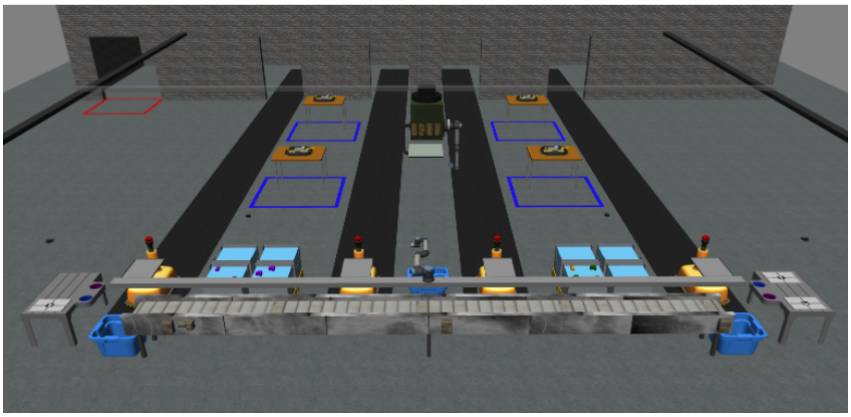

Figure 1: ARIAC Environment

2 Architecture

2.1 Type of architecture

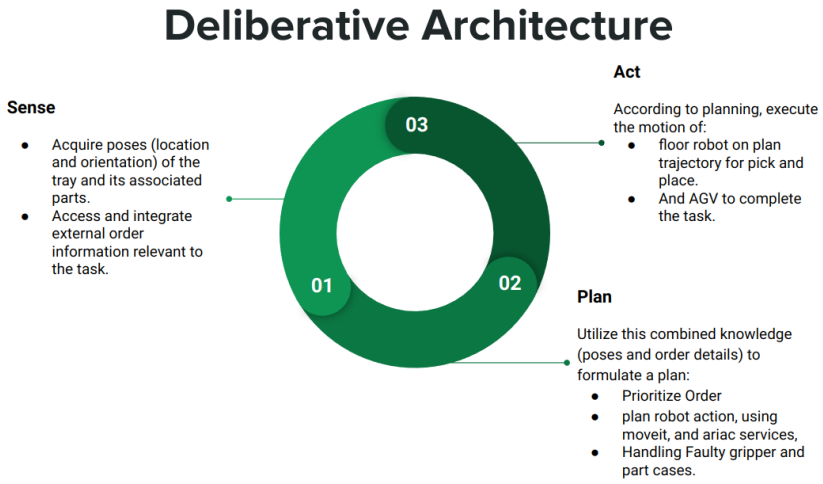

Figure 2: Hierarchical/Deliberative Architecture

The Hierarchical/Deliberative Architecture seen in Figure 3 for the kitting task presents several distinct benefits. Firstly, its strategic planning capabilities leverage high-level intelligence to enable the system to make informed decisions, particularly advantageous for handling complex tasks efficiently. Additionally, its predictive actions feature anticipates potential issues, enabling proactive adjustments to plans, thereby enhancing adaptability and overall performance. Moreover, the system's environmentally aware approach ensures a thorough understanding of the surrounding environment before planning and executing actions, contributing to optimal task execution and resource utilization. These combined attributes highlight the architecture's effectiveness in addressing challenging tasks while emphasizing its adaptability and efficiency in dynamic environments.

2.2 Kitting task architecture

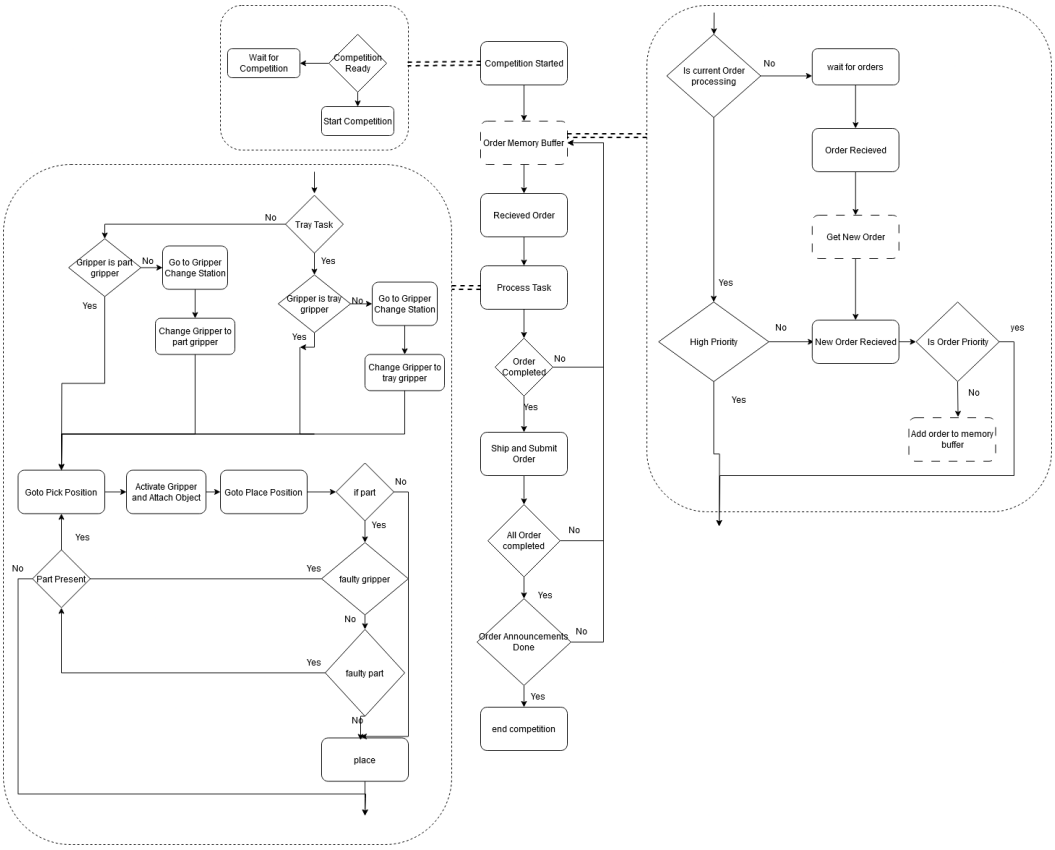

Figure 3: Kitting task Architecture

2.3 Package structure

A folder "Final_ group2" contains two packages - robot_ commander_ msgs and final_2. The former consists of custom-defined services used by the nodes in the final_2 package. The latter consists of the implementation of CCS logic combining the advantages of C++ and Python, to handle the task flow of the kitting task from reading orders to submitting them. Our project leverages the Ament CMake package to manage the dependencies, making it robust and reliable. The tree given below represents the exact package structure.

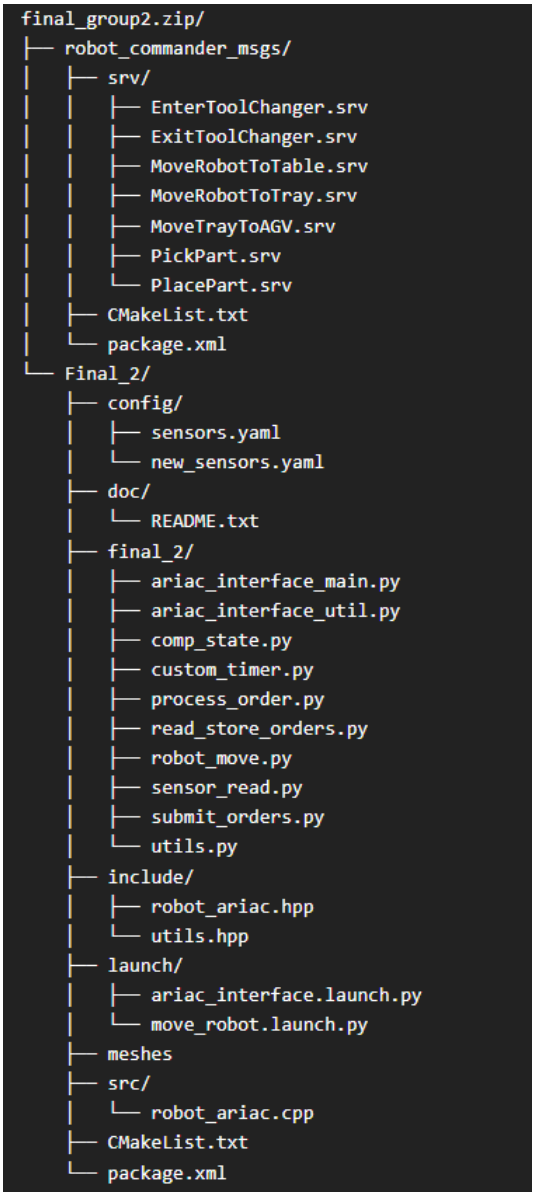

Figure 4: Package Structure

2.4 Class diagram

The competitor control system (CCS) is implemented in Python using the ariac topics to publish and subscribe to order information. Whereas, the motion planning part for the floor robot is implemented using C++ (Server) and Python (Client). Methods, attributes, and the relationship between each class are illustrated in the given class diagram. More details of each class and their corresponding methods are explained in the approach section

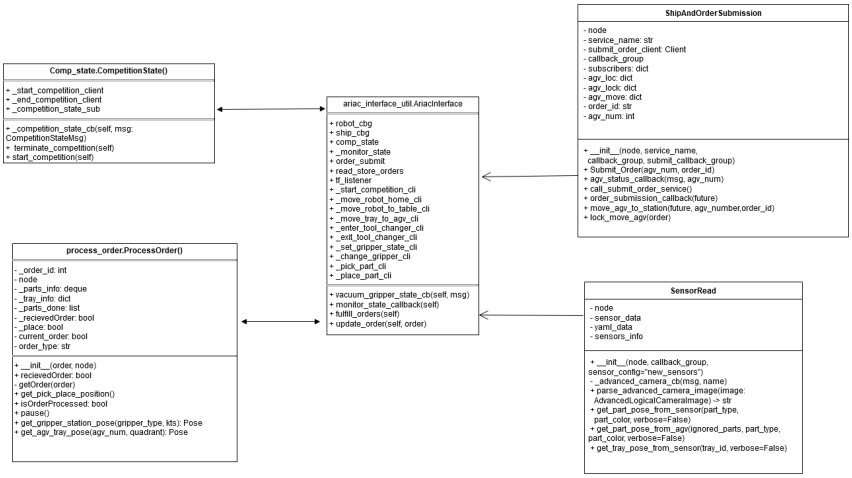

Figure 5: Class diagram

3 Approach

The project timeline, as depicted in Figure 12, delineates the sequential stages of the project, commencing with familiarization with the ARIAC environment and culminating in the successful resolution of its challenges. The group systematically navigated the final project, adhering to a structured approach aimed at achieving proficiency in completing the kitting task, both with and without the utilization of advanced logical cameras. Furthermore, the team addressed the agility challenges in subsequent phases, as elaborated in forthcoming sections.

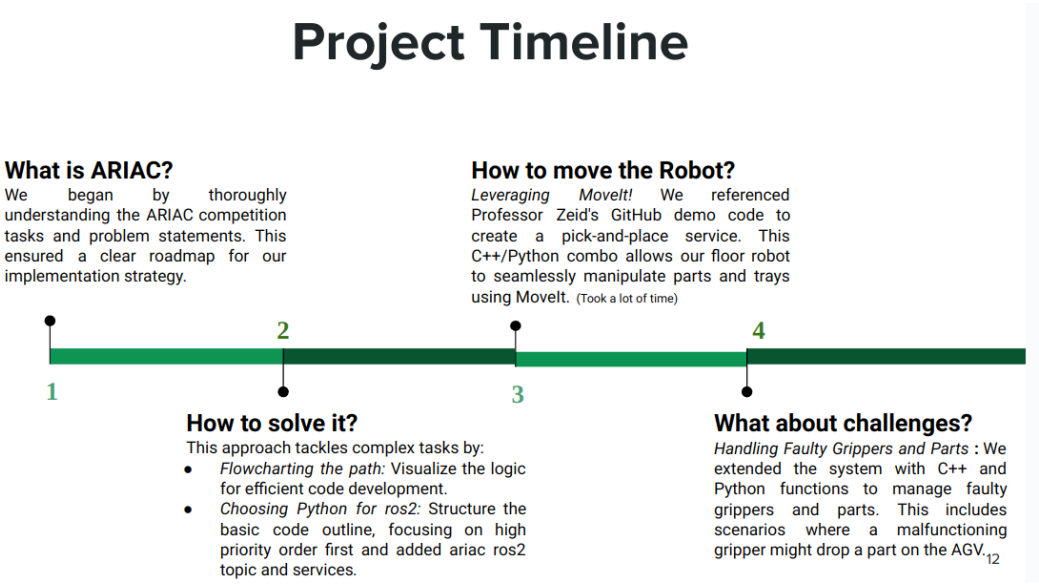

Figure 6: Project Timeline

The approach is divided into the following subtopics:

1. Starting the ARIAC Competition.
2. Read and Store Order.
3. Process the Order.
4. Submit the Order.

3.1 Start ARIAC Competition

Created a subscriber callback to */ariac/start_competition* service that checks the competition state, if the state is ready then the competition starting service is triggered by the client which is in our Python main node. Upon starting the competition the robot controllers, activates sensors, and global challenges present in the trial file; proceeding towards announcing the order.

3.2 Read and Store Orders

As soon as the competition state is ready, orders are announced on the */ariac/orders* topic. The main task in this step is to segregate orders based on priority and continue with the rest of the flow. Double-ended queues prove to be efficient in such a case, because of their insertion and deletion from both ends. So a high-priority order is pushed from the left side of the queue and a regular priority order on the right side. Additionally the use of the default sensor configuration which includes placement of advanced logical cameras over the Tray and bins to retrieve the pose and type of parts in the order, needed for proceeding with kitting.

Figure 7 shows the sensor configuration To successfully read the order part, the

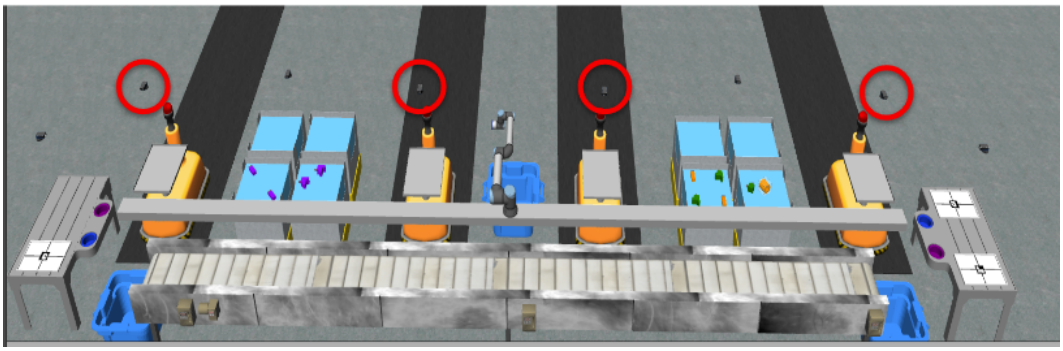

Figure 7: Sensor Configuration

sensor fusion problem had to be perfect so that we get the correct information from the fusion. To store the orders a dequeue data structure was used because of its efficient insertion and deletion and its support for parallelism.

3.3 Process the Order

Upon receiving the first order the floor robot goes and picks the tray gripper and picks the corresponding tray. To move the floor robot to either a tray or pick and place parts the ROS2 MoveIt was used. Using predefined kitting stations, bins, and AGV locations the linear rail of the floor robot was set to move to the specific locations. The waypoints method of the movement planning group interface class object was used to get the smooth motion of the UR5 floor robot. To avoid collisions during the movement of robots static objects were added to the movement planning scene. To process the order some services were used from the references given by the professor, and some were created to perform the processing of the order. To perform tasks related to moving, picking, and placing of trays and parts the services are written in C++ and the clients are in Python.

There is a number of try variable introduced to avoid an infinite loop while MoveIt is planning trajectories, for the experiments and implementations the chosen value is 10, i.e., it will try planning trajectories 10 times. After experiments, there are offsets introduced to trays and parts because the pose received from sensors gives the pose that is at the center of the object and not the surface and to avoid collision there is a need for offsets for each part and trays.

3.3.1 Pick and Place Tray

Upon receiving the order the corresponding tray ID and its placing AGV location is determined the floor robot goes to the kitting station which will be nearest to the bins from which we will pick parts for the order. Now after changing the gripper to the tray gripper, the floor robot moves to the kitting station and picks up the corresponding tray with the correct tray ID. If the trajectory generated from the MoveIt to pick the tray is valid the the floor robot approaches the tray enables the gripper and attaches the tray to the MoveIt Planning scene.

The floor robot moves to the AGV location and places the tray disables the gripper and detaches the Tray from the planning scene. After the tray has been successfully placed we move to picking the part else the robot goes and plans the trajectory to place the tray again. Then the AGV locks the tray using the services provided by Ariac.

3.3.2 Pick and place part

After placing the tray if a new order of higher priority is announced then will start preparing the second order, which is one of the agility challenges explained in the forthcoming sections. To pick and place parts the floor robot changes the gripper from a kitting station that is nearest to the bin where the parts that are going to be picked are kept. After changing the gripper the floor robot goes to the bin where the part is kept and this information is received from the sensors. The picking pose is fed to MoveIt with the pre-calculated offset, after the trajectory is planned the

gripper is enabled and the part is attached to the gripper, now there is a constant check of conditions where if the gripper is enabled and the gripper state is attached this information is checked from their topics if in any time from pick to the placing of part, the gripper disables or gets detached that means we have a faulty gripper and we append left the order and go again to pick a same part from the bins or where it fell off. If the part is not available pick the next part in order. This is part of the faulty gripper agility challenge which will be explained in the forthcoming sections.

During the placing of the part the part is placed to the corresponding AGV the floor robot places the part on a tray and the part is still attached to the gripper then the quality check is performed to see if the part is faulty or not, this faulty part check is part of the agility challenges and the details will be explained further. If the part is faulty then the part is trashed, and the replacement is searched if a replacement is not found then the robot moves and picks the next part in the order or else submits the order. If the part passes the quality check then the floor robot gripper is disabled and detaches the part from the planning scene.

3.4 Submit and Ship Order

The last step in the kitting task is to pass on the order parts or kits to the next station, which involves shipping and submitting the orders to the warehouse or whatever destination they need to be sent. The control system looks after this in the FulfillOrders method under the ARIAC interface. The following subsections get into the implementation details of this step.

3.4.1 Shipping Order

A method defined under *ShipOrderSubmission* class manages the task of shipping orders. In addition to this, AGV locations are also checked by subscribing to */ariac_msgs/msg/AGVStatus* It involves locking the tray on AGV and moving it, sending the call to */ariac/agv_lock_tray* and */ariac/move_agv* service to move the AGV to the specified station which is warehousing in our scenario. If successful, a tuple of order ID and AGV number is returned, which is required by the submit order method.

3.4.2 Submitting Order

To ensure that the trays are locked and the AGV is moved to the right location, a client to the */ariac_ msgs/srv/SubmitOrder* service is created. The method takes the order ID and AGV number as the argument and calls the service to request the response whether it is submitted successfully or not.

3.5 Detecting parts without advanced logical camera

In trivial approach to detecting parts in the environment involves the use of advanced logical cameras placed over each bin, which outputs the pose and color of the part. Although this works, it still affects the cost factor while computing the score for your control system performance. Changing the default sensor configuration to replace the advanced logical camera over the bins and tray station with a basic logical camera and RGB camera to cut the cost of sensors used. Pose can be retrieved using the basic logical camera and part type and color from RGB camera, which would require a computer vision pipeline as described below.

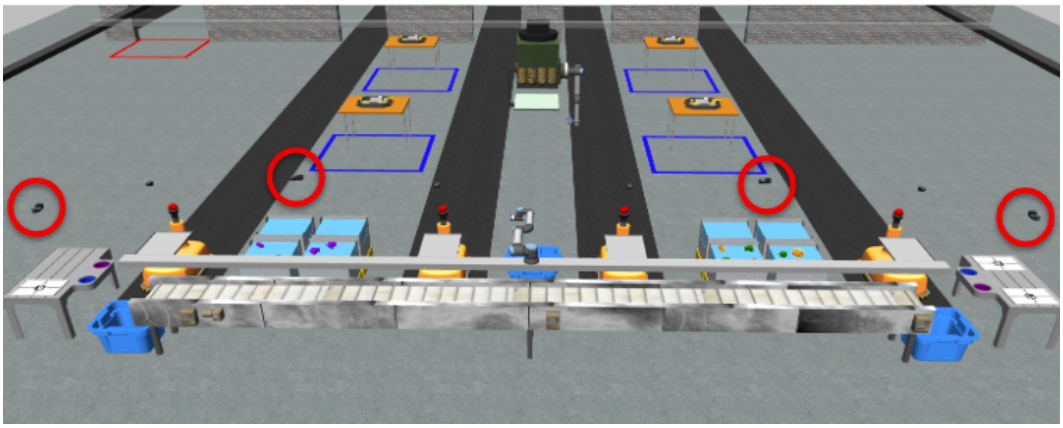

Figure 8: YOLO Sensor config

We used YOLOv8 model for color and type detection, we have 20 classes categorized by color, as shown below:

- Blue: Battery, Sensor, Pump, Regulator
- Red: Battery, Sensor, Pump, Regulator
- Purple: Battery, Sensor, Pump, Regulator
- Orange: Battery, Sensor, Pump, Regulator
- Green: Battery, Sensor, Pump, Regulator

3.5.1 Pipeline for Part Type and Color Detection

Our pipeline for part type and color detection involves several steps, as outlined below:

1. **Subscribing to RGB Camera Image:** We subscribe to the image topic from the RGB camera to capture real-time images of the environment.
2. **Object Detection using Custom Model:** We perform object detection on the RGB camera images using our custom YOLOv8 model. The detection

process identifies the part type and color of detected objects. We calculate the centroid of the detected part for localization.

3. **Subscribing to Basic Logical Camera (BLC):** Simultaneously, we subscribe to the Basic Logical Camera (BLC) to obtain the poses of the parts detected by the BLC camera. This provides us with the spatial information of the detected parts in the environment.

4. **Mapping Centroid to Gazebo Coordinates:** The centroid calculated from the RGB camera's detection results is mapped to Gazebo coordinates. This mapping allows us to establish a spatial reference frame for the detected objects. The mapping parameters between camera and basic logical camera output are mentioned below:

   - Right bins:

$$\begin{aligned} \mathrm{W}_{x\mathrm{min}}, \mathrm{W}_{x\mathrm{max}} &= -0.67, 0.43 \\ \mathrm{W}_{y\mathrm{min}}, \mathrm{W}_{y\mathrm{max}} &= -0.56, 0.54 \\ \mathrm{C}_{x_{\mathrm{min}}}, \mathrm{C}_{x_{\mathrm{max}}} &= 514, 142 \\ \mathrm{C}_{y_{\mathrm{min}}}, \mathrm{C}_{y_{\mathrm{max}}} &= 426, 62 \end{aligned}$$

   - Left bins:

$$\begin{aligned} \mathrm{W}_{x\mathrm{min}}, \mathrm{W}_{x\mathrm{max}} &= -0.59, 0.51 \\ \mathrm{W}_{y\mathrm{min}}, \mathrm{W}_{y\mathrm{max}} &= -0.56, 0.54 \\ \mathrm{C}_{x_{\mathrm{min}}}, \mathrm{C}_{x_{\mathrm{max}}} &= 490, 125 \\ \mathrm{C}_{y_{\mathrm{min}}}, \mathrm{C}_{y_{\mathrm{max}}} &= 425, 61 \end{aligned}$$

   Equations are defined for mapping in x: (1) and for y: (2):

$$\mathrm{M}_x = \left\lfloor \frac{\mathrm{W}_x - \mathrm{W}_{x\mathrm{min}}}{\mathrm{W}_{x\mathrm{max}} - \mathrm{W}_{x\mathrm{min}}} \cdot (\mathrm{C}_{x_{\mathrm{max}}} - \mathrm{C}_{x_{\mathrm{min}}}) + \mathrm{C}_{x_{\mathrm{min}}} \right\rfloor \tag{1}$$

$$\mathrm{M}_y = \left\lfloor \frac{\mathrm{W}_y - \mathrm{W}_{y\mathrm{min}}}{\mathrm{W}_{y\mathrm{max}} - \mathrm{W}_{y\mathrm{min}}} \cdot (\mathrm{C}_{y_{\mathrm{max}}} - \mathrm{C}_{y_{\mathrm{min}}}) + \mathrm{C}_{y_{\mathrm{min}}} \right\rfloor \tag{2}$$

5. **Finding Nearest Pose from BLC:** Using the mapped Gazebo coordinates, we find the nearest pose from the Basic Logical Camera (BLC) poses. This step helps in associating the detected parts with their accurate spatial positions.

6. **Publishing to Advanced Logical Camera Topic:** Finally, we publish the detected part information, including part type, color, and accurate spatial pose,

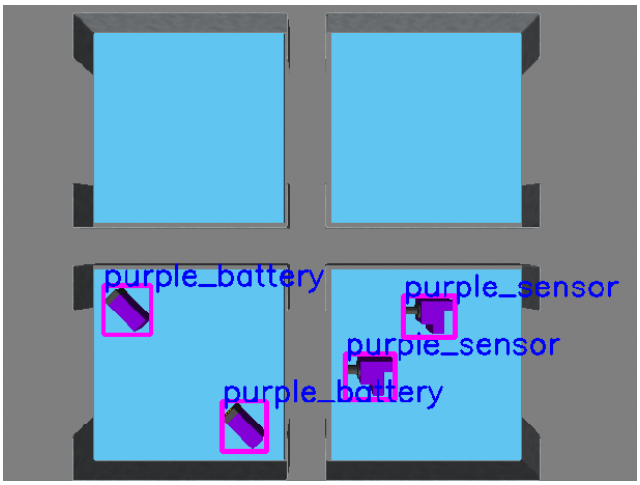

Figure 9: Detected Parts for Right Bin

to the Advanced Logical Camera topic. This published data is used further for picking and placing tasks.

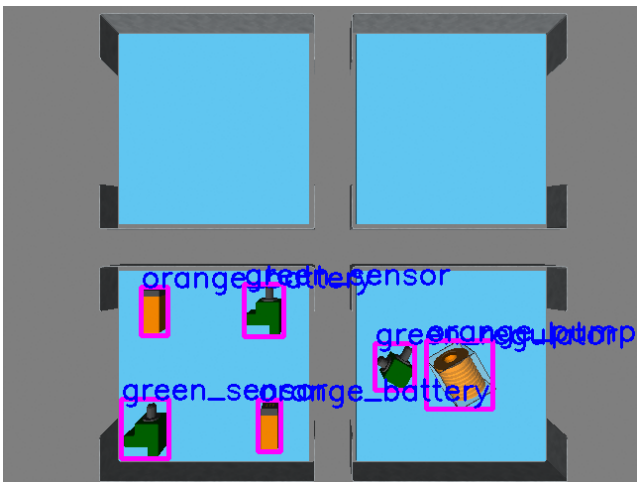

Figure 10: Detected Parts for Left Bin

3.5.2 Pipeline for ARUCO Marker Detection

Our pipeline for ARUCO marker detection for the tray also involves several steps, as outlined below:

1. **Subscribing to Camera Image:** We subscribe to the camera image topic to capture images of the tray containing ARUCO markers.

2. **ARUCO Marker Detection:** Using cv2 ARUCO marker detection function, we identify and locate ARUCO markers within the tray's image. Each ARUCO marker contains a tray id. The Aruko marker dictionary for our case

was (cv2.aruco.DICT_4X4_250)

3. **Subscribing to Basic Logical Camera (BLC):** Simultaneously, we subscribe to the Basic Logical Camera (BLC) to obtain the poses of the parts detected by the BLC camera. This provides us with the spatial information of the detected trays in the environment.

4. **Mapping ARUCO Marker Poses:** The same approach is used to for mapping the detected Aruko markers.

5. **Publishing ARUCO Marker Data:** Finally, we publish the detected ARUCO markers' IDs, poses to ALC topic.

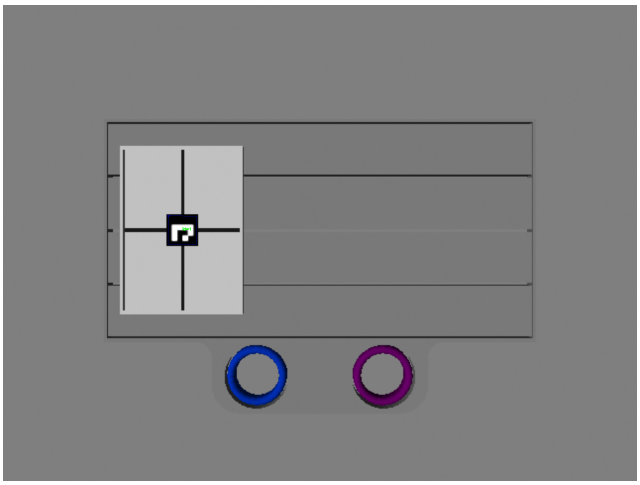

Figure 11: Detected Trays for Kts1

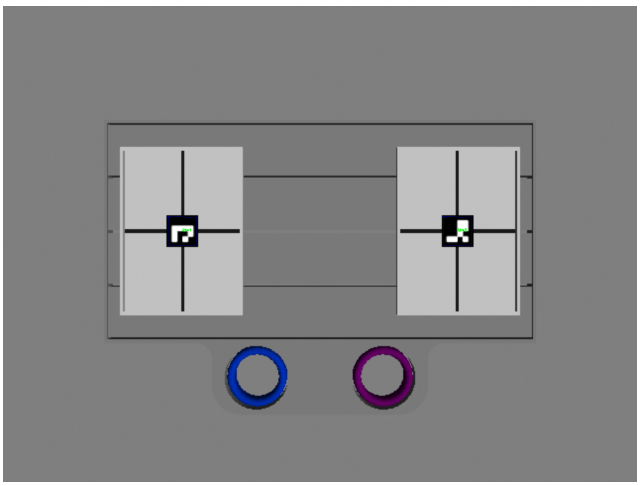

Figure 12: Detected Trays for Kts2

4 Agility Challenges

4.1 High Priority Order

This challenge tests CCS's agility to handle high-priority orders compared to regular priority orders. The priority of an order is decided by the *priority* field in */ariac/Orders*. The announced orders are stored in a double-ended queue with the necessary info i.e., ID and priority flag. Later, the orders are accessed to efficiently manage the order processing for the kitting task.

The update order function is responsible for checking the high-priority order, consequently pausing the regular one. These requirements are thoughtfully positioned so that the current order can be adequately continued at a later time without interfering with the system's general functionality. Once the flag is set to true, the control is switched to that order pausing the current one, and is stored in pending order. By following the sequence of actions, it is ensured that high-priority challenge is managed without any issue.

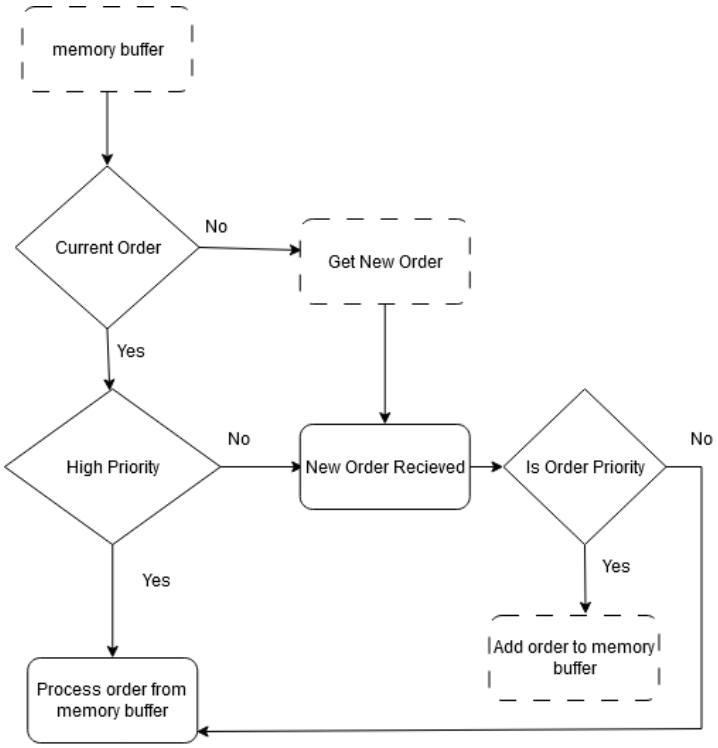

Figure 13: High Priority order pipeline

4.2 Faulty Part Challenge

In the kitting task, parts that are not appropriate are referred to as faulty parts. Such parts if submitted in the order, lead to no score for that specific quadrant. The

quality of the part is checked when it is placed in the tray present on AGV, by using the quality control sensor on it.

Our approach to resolving this involves continuously checking the quality control sensor on AGV by creating a client-to-service */ariac/perform_quality_check* and sending asynchronous requests to it. This sequence of actions is carried out by the control system when placing a part on the tray, where it is not detached until the part quality is checked. If any of the quadrants is flagged with the faulty part, the floor robot moves to the nearest trash bin depending on the AGV location to discard the part. Now, part of the same type and color is checked in the bin, if available it is used to complete the order. In case, that part isn't available then it moves to the next part for completing the order.

4.2.1 Insufficient parts challenge

A situation may arise while processing orders where the environment doesn't contain enough parts to complete the order, this is referred to as insufficient parts challenge. It may arise either in Faulty part or gripper challenge, and the control system should opt for the part replacement as mentioned in the faulty part or should submit the incomplete order. This issue is managed by using the sensor data from the bins to make sure the orders are fulfilled efficiently.

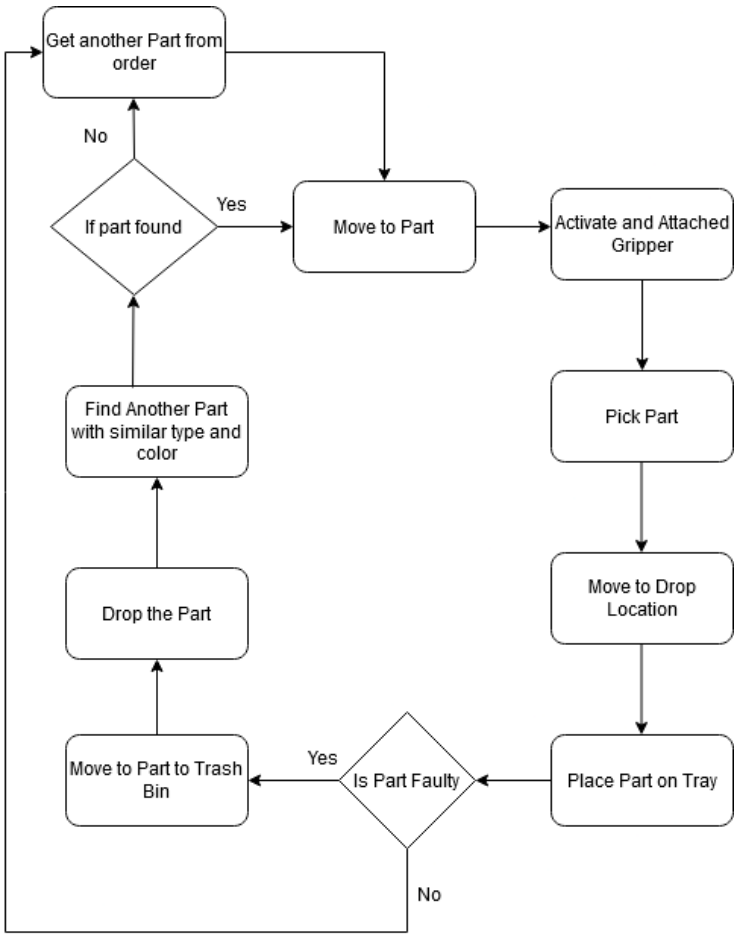

Figure 14: Faulty part pipeline

4.3 Faulty Gripper Challenge

During the kitting task, when the floor robot is being used to pick parts from the bin and place them on the tray present on AGV, a situation might occur where the part falls due to the gripper problem; known as faulty gripper challenge. To handle this, CCS should continuously check whether the part is attached to the gripper or not, and if it falls then its replacement should be used for completing the order.

The control system examines the floor robot gripper state, by subscribing to the */ariac_msgs/msg/VaccumGripperState* topic. In between pick and place, if at any time the *attached* or *enabled* field is set to false, the dropped part recovery process comes to action. Besides this, the pick service also requests the AGV number and bin side in addition to part info, which is used during the part recovery depending on the case where it fell.

- **Dropped during picking:** If the part is dropped in the bin itself from it was picked, the floor robot plans to recover the fallen part.

- **Dropped in transit:** Part is picked but during the motion to specified AGV, it is dropped in between. In this case, the control system commands the floor robot to look for the part of the same type and color in the bin.
- **Dropped in bin:** Another case can be when the floor robot is moving towards the AGV to place part, but it falls on some other bin in transit. During this, the CCS uses the *bin side* argument to move towards the bin where the part is dropped and picks it up.
- **Dropped on AGV:** To handle this edge case, 4 advanced logical cameras are placed over each of the AGVs to check for the dropped part. If the part is dropped on an AGV which is required for some other order, the floor robot replans motion to pick that part from AGV by using the AGV number argument.

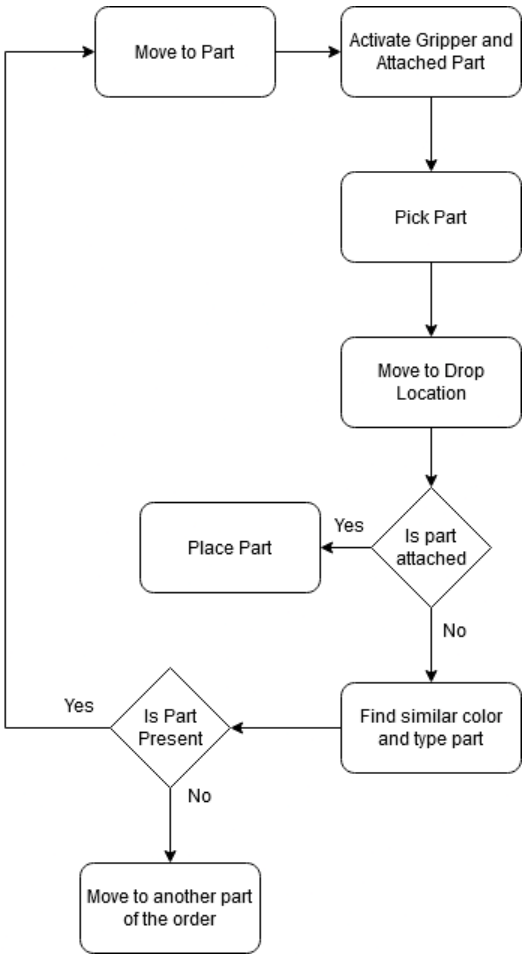

Figure 15: Faulty gripper pipeline

5 Problems encountered

- Using MoveIt's setPoseTarget() method for cartesian planning posed a challenge due to the generated trajectory not being smooth. To address this, the group implemented waypoints for smoother trajectory planning when moving the linear rail of the floor robot to various stations and locations.
- The YOLOv8 detection model performed poorly on less data, hence upon training it for more data that has more perturbations and possible angles the model performed better to give 90 percent accuracy.
- When spotting Aruco markers on trays, the Aruco seemed too small in the RGB camera image, resulting in no detections. To fix this, the image was zoomed in by 4 times, allowing successful detection. Scaling adjustments were made for accurate mapping.
- Another challenge we encountered during the integration of different nodes with Moveit functionality was ensuring that each of them could operate asynchronously without being hindered by other nodes or service calls. This was addressed by executing nodes and callback groups in separate threads to create a multi-threaded system.

6 Resources

- **ARIAC documentation:** *https://pages.nist.gov/ARIAC_docs/en/latest/*
- **Moveit lecture:** *https://github.com/zeidk/enpm663_spring2024/tree/lecture9*
- **MoveIt documentation:** *https://moveit.picknik.ai/main*
- **ROS2 (galactic) documentation:** *https://docs.ros.org/en/galactic/*
- **YOLOv8:** *https://github.com/ultralytics/ultralytics*