Real Robot Integration Guide

This guide defines the structure to bridge **MoveIt 2** planning results to your **Physical Motors** (via ESP32/Arduino).

1. The Architecture

To move from Simulation (Ignition) to Reality, we swap out the "Simulation Layer" for a "Hardware Layer".

Simulation Flow (Current): `MoveIt` -> `FollowJointTrajectory` (Action) -> `ros2_control` (JTC) -> `Ignition Plugin` -> `Simulated Motors`

Physical Flow (Proposed): `MoveIt` -> `FollowJointTrajectory` (Action) -> **Python Driver Node** -> `USB Serial` -> **ESP32 Firmware** -> `Stepper Motors`

Why this way? It is the simplest to implement. You write a single Python script that acts as the "Manager". It accepts the trajectory from MoveIt, interpolates it, and sends commands to the ESP32.

---

2. The Python Driver Node (`real_robot_driver.py`)

This node does two things: 1. **Action Server**: Listens for the trajectory from MoveIt. 2. **Serial Writer**: Sends joint angles to the ESP32.

```
import rclpy
from rclpy.node import Node
from rclpy.action import ActionServer
from control_msgs.action import FollowJointTrajectory
from sensor_msgs.msg import JointState
import serial
import struct
import time

class RealRobotDriver(Node):
    def __init__(self):
        super().__init__('real_robot_driver')

        # 1. Setup Serial to ESP32
        # self.ser = serial.Serial('/dev/ttyUSB0', 115200, timeout=0.1)
        self.get_logger().info("Connected to ESP32")

        # 2. Action Server (Replaces ros2_control's JTC)
        self._action_server = ActionServer(
            self,
            FollowJointTrajectory,
            '/parol6_arm_controller/follow_joint_trajectory',
            self.execute_callback)

        # 3. Publisher (To update RViz)
        self.joint_pub = self.create_publisher(JointState, '/joint_states', 10)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        feedback_msg = FollowJointTrajectory.Feedback()

        # Iterate through trajectory points
        for point in goal_handle.request.trajectory.points:
            positions = point.positions # [L1, L2, L3, L4, L5, L6]
```

```python
            # A. Send to ESP32 (Format: <A,B,C,D,E,F>)
            cmd_str = f"<{','.join(map(str, positions))}>\n"
            # self.ser.write(cmd_str.encode())

            # B. Publish State for RViz
            msg = JointState()
            msg.header.stamp = self.get_clock().now().to_msg()
            msg.name = goal_handle.request.trajectory.joint_names
            msg.position = positions
            self.joint_pub.publish(msg)

            # Wait for next point (Simple open-loop timing)
            time.sleep(0.1)

        goal_handle.succeed()
        result = FollowJointTrajectory.Result()
        return result

def main(args=None):
    rclpy.init(args=args)
    driver = RealRobotDriver()
    rclpy.spin(driver)
```

---

3. The Firmware (ESP32/Arduino)

The logic on the microcontroller side to parse the protocol `<J1,J2,J3,J4,J5,J6>`.

```cpp
#include <AccelStepper.h>

// Define your steppers here...

void setup() {
  Serial.begin(115200);
}

void loop() {
  if (Serial.available() > 0) {
    String input = Serial.readStringUntil('\n');
    // Expected format: <0.5, 1.2, -0.4, ...>
    if (input.startsWith("<") && input.endsWith(">")) {
      input = input.substring(1, input.length() - 1);

      // Parse Floats (Naive implementation)
      float joints[6];
      int index = 0;
      int lastIndex = 0;
      for (int i = 0; i < input.length(); i++) {
        if (input.charAt(i) == ',') {
          joints[index++] = input.substring(lastIndex, i).toFloat();
          lastIndex = i + 1;
        }
      }
      joints[index] = input.substring(lastIndex).toFloat();

      // Move Motors
      // stepper1.moveTo(radToSteps(joints[0]));
      // ...
```

```
    }
  }
  // stepper1.run();
}
```

4. How to Run It

1. **Close Simulation**: Stop any running Ignition instances.
2. **Flash ESP32**: Upload the code.
3. **Launch Driver**: `python3 real_robot_driver.py`
4. **Launch MoveIt**: `ros2 launch parol6_moveit_config moveit.launch.py` (You might need a modified launch file that doesn't wait for Gazebo).

5. Digital Twin Setup (Simulation + Real Robot)

Yes! You can run the simulation and have the real robot copy it (Shadow Mode). This is often called a "Digital Twin".

The Data Flow: 1. **MoveIt** plans a path. 2. **Ignition Gazebo** executes it visually. 3. **Real Robot Driver** (Shadow Mode) subscribes to `/joint_states` from the simulation. 4. **Real Robot** moves to match the Simulation angles in real-time.

Modified Python Driver for Shadowing: Instead of an Action Server, standard subscriber logic:

```python
# Inside RealRobotDriver __init__:
self.sub = self.create_subscription(JointState, '/joint_states', self.shadow_callback, 10)

def shadow_callback(self, msg):
    # msg.position contains [L1...L6] from Simulation
    # Send directly to ESP32
    cmd = f"<{','.join(map(str, msg.position))}>\n"
    self.ser.write(cmd.encode())
```

Risk: If Simulation jumps instantly (teleport), Real Robot tries to jump instantly (dangerous). You need safety limits in firmware.

6. The "Standard" ros2_control Firmware Way

You asked: *"does ros_control firmware has something to do with that?"* **Answer: YES.**

The method in Section 1 (Python Driver) is a **bypass**. The "Official/Professional" ROS 2 way is:

Architecture: `MoveIt` -> `ros2_control` (Controller Manager) -> **Hardware Interface (C++)** -> `Micro-ROS Agent` -> `ESP32 Firmware`

1. **Hardware Interface**: You write a C++ class in ROS that inherits from `SystemInterface`.
2. **Communication**: This C++ class sends/receives data to the hardware.
3. **Firmware**: The ESP32 runs a **Micro-ROS Node** that exposes Services/Topics directly to the Host.

Why simpler (Python) vs. Standard (ros2_control)? * **Python Bridge**: Easier to understand, debug, and implement for a Thesis/Prototyping. * **ros2_control C++**: Harder to write (C++ plugins, pluginlib), but gives real-time guarantees and perfect synchronization with MoveIt.

Recommendation: Start with the **Python Bridge / Digital Twin** (Section 5). It achieves 90% of the value with 10% of the complexity.

7. The Conventional Way: Mode Switching (Best Practice)

You asked: *"if gazebo has a problem the real robot will too? what is the conventional way?"*

You are absolutely correct. Chaining them (MoveIt -> Gazebo -> Real) introduces a "single point of failure" (Gazebo).

The Conventional Industrial Approach is "Mode Switching". You do not run both at the same time. You act like a railroad switch.

Architecture: * **Mode A (Simulation)**: `MoveIt -> Action A -> Gazebo` * **Mode B (Real)**: `MoveIt -> Action B -> Real Robot`

How it works: 1. **Verify in Sim**: You launch **Mode A**. You plan, execute, and see the robot move in Gazebo. "Looks good." 2. **Switch**: You kill Gazebo. You launch **Mode B** (your Python Driver). 3. **Execute in Real**: You hit "Plan & Execute" in MoveIt again. This time, MoveIt talks **directly** to your Python Driver (Action Server).

Why this is better: * **Safety**: If Gazebo crashes, the Real Robot doesn't care. It's not connected. * **Latency**: You don't have the delay of the simulation physics engine. * **Isolation**: Real hardware issues (serial noise) don't freeze your simulation, and vice versa.

To implement "Simultaneous" (Advanced): If you *really* want both moving at once without the dependency, you write an **Action Splitter**. * MoveIt sends to `/joint_trajectory_splitter`. * Splitter sends copy 1 to `/gazebo/joint_trajectory`. * Splitter sends copy 2 to `/real/joint_trajectory`. This is complex to synchronize (what if Real is slower than Sim?) and usually avoided unless necessary.

8. Mapping: The ROS-to-Hardware Math

You asked: *"how to map commands from ros to arduino/esp"*

The ROS world speaks **Radians**. Your Motor world speaks **Steps**. You need a translation layer.

A. Concept: The Conversion Factor

You need to calculate `STEPS_PER_RADIAN` for each joint. Equation: `Steps_Per_Rev * Microstepping * Gear_Ratio / (2 * PI)`

Example (NEMA 17 + 20:1 Planetary Gear): * Motor: 200 steps/rev (1.8 deg) * Microstepping: 16 (on driver) * Gear Ratio: 20 * Total Steps/Rev = 200 * 16 * 20 = 64,000 steps * **STEPS_PER_RADIAN** = 64,000 / 6.28318 ~= **10,185.9**

B. Implementing in Firmware (Arduino/ESP32)

Don't send big step numbers over Serial. Send radians (floats) and let the MCU convert.

```
// 1. Define Ratios (Calibrate these!)
float STEPS_PER_RAD_L1 = 10185.9;
float STEPS_PER_RAD_L2 = 8500.0;
// ...

// 2. The Loop Logic
if (received_new_target) {
    // ROS sends: 1.57 radians (90 degrees)
    float target_rad = parsed_joint_L1;

    // Convert
    long target_steps = target_rad * STEPS_PER_RAD_L1;

    // 3. Handle Direction (If motor spins wrong way)
    // if (INVERT_L1) target_steps = -target_steps;

    stepper1.moveTo(target_steps);
}
```

C. Handling "Home" (Zero Offset)

ROS assumes `0.0` is the robot standing straight up (or specific pose). Your motors assume `0` is "where I turned on". * **Solution**: You need a Limit Switch Homing Sequence in `setup()`. * Once switch is hit, call `stepper.setCurrentPosition(HOME_OFFSET_STEPS)`.

9. Velocity & Acceleration

You asked: *"does moveit send speed and acceleration info?"* **Answer: YES.**

The `FollowJointTrajectory` message contains a list of points. Each point has: 1. `positions` (Where to go) 2. `velocities` (How fast to pass through this point) 3. `accelerations` (How fast to speed up/slow down) 4. `time_from_start` (When to arrive exactly)

A. How to access it in Python

In your `real_robot_driver.py`:

```python
for point in goal_handle.request.trajectory.points:
    pos = point.positions       # List[float]
    vel = point.velocities      # List[float]
    acc = point.accelerations   # List[float]

    # Send all to ESP32: <P1,P2... | V1,V2... | A1,A2...>
    # OR just ignore V/A if your firmware is simple.
```

B. Do you need it?

- **Simple Way (Recommended)**: Ignore V/A. Just send Positions. Let `AccelStepper` in Arduino handle the acceleration between points.
  - *Pros*: Much simpler firmware.
  - *Cons*: Robot might stop/start briefly at each point (choppy) if points are far apart.
- **Advanced Way**: Send V/A to firmware.
  - *Pros*: Perfectly smooth motion.
  - *Cons*: Complex math on Arduino (Cubic Spline interpolation).

Tip: MoveIt generates points very close together (e.g., every 0.1s). So even if you ignore Velocity, "Position Control" usually looks pretty smooth because the updates are frequent.

10. Can I do math on PC? (PC-Side Interpolation)

You asked: *"can i make the intense calculations on my machine and send all info to arduino"* **Answer: YES. This is actually a very smart approach.**

The Concept ("Dumb Firmware"): Instead of sending 2 points spaced 1 second apart and asking Arduino to interpolate, the PC calculates **100 little intermediate points** and sends them continuously.

A. How to implement

1. **Python Node**:
   - Receives Trajectory (points every 0.1s).
   - Runs a high-speed loop (e.g., 50Hz or 100Hz).
   - Calculates exactly where the robot should be *right now*.
   - Sends stream: `<J1, J2...>` -> Serial -> `<J1, J2...>` -> Serial…
2. **Arduino**:
   - Does **ZERO** math.
   - Just parses `<J1...>` and calls `stepper.moveTo()`.

B. The Bottleneck: Serial Bandwidth

If you send updates too fast, Serial chokes. * **Math**: 6 floats + chars ~= 50 bytes per message. * **Baud Rate**: 115200 bits/sec ~= 11,500 bytes/sec. * **Max Rate**: 11,500 / 50 ~= **230 updates/second**. * **Conclusion**: You can safely send updates at **50Hz - 100Hz** effortlessly. This is extremely smooth for a robot arm.

Recommendation: Use **PC-Side Interpolation**. It keeps your Arduino code simple and robust (less crashing), while leveraging your i7/Ryzen CPU for the heavy lifting.

11. Thesis Level: The Industrial Standard

You asked: *"what is the conventional way (thesis level)"*

In professional robotics (KUKA, Universal Robots, ABB), the standard is **"Centralized Planning, Distributed Control"**.

The Architecture (Thesis Terminology)

1. **High-Level Controller (PC/ROS)**:
   - **Role**: "The Brain".
   - **Tasks**: Path Planning (A*), Inverse Kinematics, Collision Checking, Trajectory Interpolation.
   - **Output**: Constant stream of Setpoints (e.g., 100Hz or 1kHz) over a Fieldbus (EtherCAT, CANopen).
   - *This is exactly the "PC-Side Interpolation" approach.*
2. **Low-Level Controller (MCU/Drive)**:
   - **Role**: "The Muscle".
   - **Tasks**: Field Oriented Control (FOC), PID Position Loop, Encoder Reading.
   - **Input**: "Go to Angle X **NOW**".
   - **Output**: PWM currents to motor coils.

Why this is the standard?

1. **Dynamic Response**: If a camera sees a human, the PC stops sending updates *instantly*. If the MCU held the whole trajectory buffer (like a 3D printer), it might execute several seconds of motion before processing the "Stop" command.
2. **Compute Power**: Inverse Kinematics requires matrix inversion. An Arduino cannot do this fast enough.

For Your Thesis Report:

You should write: > *"The system utilizes a centralized control architecture where high-level trajectory generation and interpolation occur on the host computer running ROS 2. Setpoints are streamed at [Frequency] Hz to the distributed microcontroller nodes, which execute the low-level motor actuation loops. This mirrors standard industrial protocols like EtherCAT provided by KUKA/UR interfaces."*

12. Using MKS SERVO42C (Closed-loop / FOC)

You asked: *"i have FOC through mkservo42c drivers . does that work? what would the code look like"*
Answer: YES. It works beautifully, and it simplifies your job.

A. How MKS SERVO42C works

These are **"Smart Drivers"**. * **Internal**: They have their own processor, encoder, and FOC algorithm built-in. * **External**: To your ESP32, they lock exactly like a standard "Dumb" stepper driver (A4988/TMC2209).

B. The Code

The code **DOES NOT CHANGE**. You still use the `DIR` and `STEP` (Pulse) interface. * The ESP32 sends a pulse. * The MKS 42C receives the pulse and uses its FOC algorithm to move the motor exactly one step, applying exactly the right torque.

Your Firmware remains:

```
// Works on MKS SERVO42C just like a regular stepper
accelStepper.moveTo(target_steps);
accelStepper.run();
```

C. Critical Setup for These Drivers

1. **Calibration**: You MUST run the MKS calibration menu (using the buttons/OLED on the driver) before connecting to the robot. This teaches the driver the motor's magnet properties.
2. **Microstepping**: Set the microstepping on the Driver's OLED screen (e.g., 16 or 32). Ensure your ROS code (Section 8) uses the **SAME** number for the `STEPS_PER_RADIAN` calculation.
3. **Speed**: These drivers can handle very high RPM. Ensure your `AccelStepper` max speed is set high enough in firmware (`setMaxSpeed`).

Why this is great for Thesis: You get "Industrial FOC Performance" (Silence, No lost steps, Efficiency) without writing a single line of FOC code yourself. It is a "Black Box" solution.

13. Feedback: Do I need to read the Encoders?

You asked: *"is the feedback from the mkservo important in trajectory planning in ros?"*

A. For Planning (MoveIt Generation)

NO. MoveIt assumes the robot is "perfect". It generates a plan based on where it *thinks* the robot is effectively instantaneously. It does not use live torque/error feedback to generate the geometric path.

B. For Execution (ros2_control / Driver)

YES, but MKS handles it. * **Safety**: If the robot hits a wall, the MKS driver will detect "Stall" (position error) and stop the motor to prevent damage. * **ROS Visualization**: ROS needs to know where the robot IS to update the 3D model in RViz.

Types of Feedback Implementation:

1. **"Open Loop" Reporting (Standard/Easiest)**:
   - Your ESP32 counts how many steps it *sent*.
   - It assumes `Position = Steps_Sent / Ratio`.
   - It publishes this to `/joint_states`.
   - *Why allows this?* Because MKS 42C guarantees that if it *can* move, it *will* move. If it fails, it errors out. So "Steps Sent" is extremely accurate (99.9%).
2. **True Closed Loop (Advanced)**:
   - You wire the MKS `TX/RX` serial port or `Enc` pins back to the ESP32.
   - You ask the driver: "Where are you really?"
   - *Verdict*: Overkill for this project. The MKS driver is trusted to do its job.

Recommendation: Use **Type 1**. Trust the MKS driver. report `Steps_Sent` back to ROS as the current state.

14. Academic Context: When is Feedback Critical?

You asked: *"is it academically recommended to at least get feedback from time to time?"* **Answer: YES. For two specific reasons.**

A. "The Start State" (t=0)

MoveIt cannot plan a path if it doesn't know where the robot is **RIGHT NOW**. * **Scenario**: Robot is at 90 degrees. You turn it off. You move it by hand to 0 degrees. You turn it on. * **No Feedback**: ROS thinks it's still at 90. It plans a path through a table. **CRASH.** * **With Feedback**: Enc sends "0". ROS sees "0". Plans safe path. * **Thesis Point**: *"Feedback is essential for validating the initial state `q(t=0)` before any trajectory generation begins."*

B. Trajectory Execution Monitoring (Replanning)

This is a standard diagram in robotics textbooks. 1. **Monitor**: You continuously compare `Actual` vs `Planned`. 2. **Threshold**: If `|Error| > 0.1 rad` (maybe a heavy object pulled the arm down), you trigger an **ABORT**. 3. **Re-Plan**: MoveIt calculates a *new* correction path from the deviation point.

Implementation for Thesis

Even if you use "Open Loop" stepping (steps sent), you simulate this standard by: 1. **Homing at Startup**: This is the "Gold Standard" feedback. It resets the world to Truth. 2. **Stall Detection**: If MKS Servo42c detects a stall, it should toggle a pin. ESP32 reads this pin -> Sends "ABORT" to ROS.

15. Concrete Implementation Steps

You asked: *"how can this be implemented in our setup"*

Part A: Homing (The "Start State" Fix)

1. Hardware: * Install a mechanical Limit Switch on each joint (or at least J1, J2, J3). * Wire them to ESP32 Pins (e.g., `D13`, `D12`...). Use Pull-up resistors (Internal `INPUT_PULLUP`).

2. Firmware (Arduino/ESP32): Add this to `setup()`:

```
void setup() {
  Serial.begin(115200);
  pinMode(LIMIT_SWITCH_1, INPUT_PULLUP);

  // 1. Homing Routine
  Serial.println("STATUS: HOMING...");

  // Move until switch hit
  while(digitalRead(LIMIT_SWITCH_1) == HIGH) {
    stepper1.runSpeed(); // Move slowly backwards
  }

  // 2. Set Zero
  stepper1.setCurrentPosition(0);
  Serial.println("STATUS: READY");
}
```

3. Python Driver Update: In your `__init__`, wait for that "READY" signal before letting MoveIt start.

```
# Wait for ESP32 to finish homing
while True:
    line = self.ser.readline().decode()
    if "READY" in line:
        self.get_logger().info("Robot Homed & Ready!")
        break
```

Part B: Stall/Error Monitoring (The "Execution" Fix)

1. Hardware: * The MKS SERVO42C has an **Alarm (ALM)** pin (sometimes called `Di`). * Wire this to an ESP32 Input Pin.

2. Firmware: In your main `loop()`:

```
void loop() {
  if (digitalRead(ALM_PIN_1) == LOW) { // Assuming Active LOW alarm
    Serial.println("ERROR: STALL_J1");
    // Stop everything
    while(1);
  }
  // ... rest of code
}
```

3. Python Driver: In your execution loop:

```python
# Check for errors while moving
if self.ser.in_waiting:
    msg = self.ser.readline().decode()
    if "ERROR" in msg:
        self.get_logger().error("Robot Stalled! Aborting.")
        goal_handle.abort()
        return FollowJointTrajectory.Result()
```

16. Deep Dive: Alarms & Deviation Monitoring

You asked: *"what does the alarm pin do? and how to monitor deviations"*

A. What does the Alarm Pin do?

On the MKS SERVO42C, the `ALM` pin goes active (Low/High) when the **Driver gives up**. It triggers if: 1. **Position Deviation**: The motor is physically forced > X degrees away from where it should be (default is usually ~2 degrees or 1 full step cycle). 2. **Stall**: The magnet cannot push the load anymore. 3. **Overheat**: Driver is too hot.

It is a "Catastrophic Failure" signal. It means the driver has lost control of the position.

B. How to Monitor Deviations (Tracking Error)

There are two places to do this.

Level 1: Hardware Monitoring (Internal to Driver)

- **How**: The MKS 42C is *always* calculating `Error = Target - Actual` internally at 20kHz.
- **Action**: It applies more current to fix the error.
- **Failure**: If it can't fix it, it fires the **Alarm Pin**.
- **Verdict**: This is usually **Enough for a Thesis**. You rely on the hardware to police itself.

Level 2: Software Monitoring (ROS Level) If you want to see a graph of "Planned vs. Actual" in ROS (like `rqt_plot`): 1. **Requirement**: You MUST wire the MKS `TX` pin to the ESP32 `RX` and read the *real* encoder angle. 2. **Logic**: * ROS sends `Target(t)`. * ROS reads `Actual(t)`. * ROS calculates `Error(t)`. 3. **Visualization**: * Launch `rqt_plot`. * Plot `/joint_states/position[0]` (Actual) vs `/parol6_arm_controller/follow_joint_trajectory/goal...` (Planned).

Recommendation: Stick to **Level 1 (Alarm Pin)**. Reading 6 serial streams from 6 motors to get "Level 2" data is very complex (timing issues, slow serial) and often unnecessary if the Driver is smart.

17. Replanning: What happens after an Error?

You asked: *"what about replanning"*

Replanning is the automatic attempt to find a *new* path when the current one fails.

The Architecture

1. **The Trigger**: The "Stall" (Section 15/16) stops the robot and returns `ABORTED` to MoveIt.
2. **The Reaction**: MoveIt sees `ABORTED`.
3. **The Logic**:
    - **Default**: MoveIt gives up.
    - **With Replanning Enabled**: MoveIt looks at the *current* state (where it stopped) and tries to plan a path to the *original* goal again.

How to Enable It (MoveIt)

- **In RViz**: Check the box **"Replanning"** in the MotionPlanning panel.
- **In Code (Python)**: `python group.allow_replanning(True) group.set_planning_time(5.0) # Give it time to think`

The Danger (Thesis Critical Thinking)

Replanning is dangerous without a **Camera** (Octomap). * **Scenario**: Robot hits a "Invisible Wall" (Not in URDF). * **Stall**: Robot stops. * **Replan**: MoveIt doesn't know *why* it stopped. It thinks "Maybe the path was just weird". It plans a slightly different path *right back into the wall.* * **Loop**: Bang -> Replan -> Bang -> Replan…

Thesis Recommendation

For your project, implement **"Stop and Notify"** instead of "Auto-Replan". > *"Due to the lack of full-body tactile sensing, automatic replanning is disabled to prevent repeated collisions. The system halts on deviation and requires operator intervention."*

18. Visual Feedback: Should I watch the End Effector?

You asked: *"i have kinect v2 camera. is it recommended that i get the end effector feedback from it?"*

The Answer: NO (for Control Loop), YES (for Calibration).

1. Why NOT for Control (Real-time Feedback): * **Latency**: Kinect gives data at 30Hz with ~50-100ms delay. Your motors need updates at 1000Hz+. By the time you "see" the error, the robot has already moved past it. * **Accuracy**: Kinect v2 noise is ~2mm-10mm depending on distance. Your MKS Steppers are accurate to ~0.05mm. The camera is *less* accurate than the robot! * **Occlusion**: The robot arm will often block the camera's view of the hand.

2. Where it IS Recommended: * **Target Finding**: Use Kinect to find the *Apple*. Tell MoveIt "Go to Apple". Then trust the Encoders to get there. * **Verification (ArUco)**: Put an **ArUco Marker** on the hand. Move to a point. Check Camera. Record the difference (Static Error). Use this to calibrate your URDF (Link Lengths), but do not use it in the live movement loop.

Thesis Verdict: > *"Visual feedback is utilized for Object Detection and Static Pose Verification, but excluded from the dynamic control loop due to latency constraints and occlusion risks."*

19. The Complete Thesis Pipeline: "Visual Seam Tracking"

You asked: *"how the series of points from the camera should become a smooth path… what is responsible for each task?"*

This is the **Architecture Diagram** for your Thesis.

Phase 1: Perception (The "Eyes")

- **Responsible**: Custom ROS Node (`vision_processor.py`).
- **Input**: `/kinect2/qhd/points` (PointCloud2) + `/kinect2/qhd/image_color`.
- **Logic**:
  1. **YOLOv8**: Detects Object (Bounding Box).
  2. **YOLO-Seg / OpenCV**: Isolates the "Seam" or "Curve".
  3. **Deprojection**: Converts 2D Pixels (u,v) -> 3D Points (X,Y,Z) using PCL or Depth Map.
- **Output**: `geometry_msgs/PoseArray`. A naive list of noisy 3D points.

Phase 2: Path Generation (The "Smoother")

- **Responsible**: Same Node or new `path_generator.py`.
- **The Problem**: Kinect data is noisy. The points will zigzag (jitter). You cannot weld like that.
- **The Solution**: **B-Spline Interpolation**.
- **Logic**:
  1. Take the list of noisy 3D points.
  2. Fit a **B-Spline Curve** (using `scipy.interpolate`).
  3. Resample the curve at fixed intervals (e.g., every 1mm).
- **Output**: `nav_msgs/Path`. A perfectly smooth curve.

Phase 3: Motion Planning (The "Brain")

- **Responsible**: MoveIt 2 (`move_group`).
- **Logic**: **Cartesian Path Planning**.
  - You don't just say "Go here".
  - You call `compute_cartesian_path(waypoints)`.
  - MoveIt solves Inverse Kinematics (IK) for *every* point on that curve to ensure the arm follows the line exactly.
- **Output**: `trajectory_msgs/JointTrajectory` (Time-stamped joint angles).

Phase 4: Execution (The "Muscle")

- **Responsible**: Your `real_robot_driver.py` + MKS Drivers.
- **Logic**: PC-Side Interpolation (from Section 10).
  - Takes the trajectory.
  - Streams setpoints to ESP32.
  - Stops if `ALM` pin triggers.

Summary Table for Thesis Report

Task	Node Name (Proposed)	Algorithm / Library	Input	Output
Object Detection	`vision_node`	YOLOv8 / Ultralytics	RGB Image	ROI (Region of Interest)
Seam Extraction	`vision_node`	PCL / Depth Deprojection	Depth Map	`PoseArray` (Noisy Points)
Path Smoothing	`path_planner`	**B-Spline Interpolation**	`PoseArray`	`nav_msgs/Path` (Smooth)
Trajectory Gen	`move_group`	**Compute Cartesian Path**	`nav_msgs/Path`	`JointTrajectory`
Trajectory Gen	`move_group`	**Compute Cartesian Path**	`nav_msgs/Path`	`JointTrajectory`
Control	`driver_node`	Serial Streaming	`JointTrajectory`	Motor Steps (RS232/TTL)

20. Deep Dive: Planning a Path (Not a Point)

You asked: *"the moveit operation will change because the target is not a point but a path, how is that achieved?"*

The difference: PTP vs. LIN

1. **Standard (`go()`)**: You act like a specialized Taxi. "Take me to the Airport". The driver (MoveIt) chooses the route. It might swing the arm wildly (PTP Motion) as long as it gets there safe.
2. **Seam Tracking (`compute_cartesian_path()`)**: You act like a Train. "Follow this exact track". You must pass through every point in order (LIN Motion).

How to Implement (Python Code)

You don't use `set_pose_target`. You use `compute_cartesian_path`.

```python
# 1. Define the Waypoints (from your B-Spline)
waypoints = []

# Ideally, these come from your 'nav_msgs/Path' topic
p1 = geometry_msgs.msg.Pose()
p1.position.x = 0.4
p1.position.y = 0.1
p1.position.z = 0.4
```

```
waypoints.append(p1)

p2 = copy.deepcopy(p1)
p2.position.x = 0.5 # Move 10cm forward
waypoints.append(p2)
# ... add 50 more points ...

# 2. Ask MoveIt to solve IK for the whole chain
# eef_step: Resolution (1cm). jump_threshold: 0.0 (Disable jump check)
(plan, fraction) = move_group.compute_cartesian_path(
                                   waypoints,   # waypoints to follow
                                   0.01,        # eef_step
                                   0.0)         # jump_threshold

# 3. Check Success
if fraction < 1.0:
    print(f"Warning: Only computed {fraction*100}% of the path!")
else:
    # 4. Execute
    move_group.execute(plan, wait=True)
```

What happens inside `compute_cartesian_path`?

1. **Interpolation**: It draws a straight line between Waypoint 1 and Waypoint 2.
2. **Slicing**: It chops that line into tiny steps (defined by `eef_step`, e.g., 1cm).
3. **IK Solving**: It solves Inverse Kinematics for *every single slice*.
4. **Stitching**: It combines all those joint solutions into one big `JointTrajectory`.
5. **Stitching**: It combines all those joint solutions into one big `JointTrajectory`.
6. **Output**: This trajectory is exactly what your `real_robot_driver` (Section 2) already knows how to play!

21. Going Up the Stack: Vision & Path Implementation

You asked: *"how would compute cartesian path be implemented, scipy, spline, perception..."*

Here is the **Python Code Logic** for the upstream phases.

Phase 2: Path Smoothing (Scipy B-Spline)

You have a list of noisy dots from the camera (`raw_points`). You need a smooth line for MoveIt.

```
import numpy as np
from scipy.interpolate import splprep, splev

def generate_smooth_path(raw_points, resolution=0.01):
    # raw_points = [[x1,y1,z1], [x2,y2,z2]...] (Noisy)

    # 1. Transpose to [X_list, Y_list, Z_list]
    points_array = np.array(raw_points).T

    # 2. Fit B-Spline (s=smoothing factor)
    # The larger 's', the smoother (less jitter), but further from original points.
    tck, u = splprep(points_array, s=0.001)

    # 3. Generate new smooth points
    # Create 100 evenly spaced points along the curve
    u_new = np.linspace(0, 1, num=100)
    x_new, y_new, z_new = splev(u_new, tck)

    # 4. Convert to ROS Poses for MoveIt
```

```
    waypoints = []
    for i in range(len(x_new)):
        p = Pose()
        p.position.x = x_new[i]
        p.position.y = y_new[i]
        p.position.z = z_new[i]
        # Orientation: Hardcode 'Down' (Tool pointing at table)
        p.orientation.w = 1.0
        waypoints.append(p)

    return waypoints
```

Phase 1: Perception (YOLO + PCL)

How to get `raw_points` from an image.

```
# Pseudo-code logic for 'vision_processor.py'
import cv2
import numpy as np
from ultralytics import YOLO

def process_vision(rgb_image, depth_image, camera_intrinsics):
    # 1. Detect Seam/Object Mask
    model = YOLO("yolov8-seg.pt")
    results = model(rgb_image)
    mask = results[0].masks.data[0].numpy() # Binary mask of the seam

    # 2. Extract Pixels on the Seam
    # Get all (u,v) coordinates where mask == 1
    seam_pixels = np.argwhere(mask > 0)

    raw_Points_3d = []

    # 3. Deproject to 3D
    fx = camera_intrinsics['fx']
    cx = camera_intrinsics['cx']
    # ...

    for (v, u) in seam_pixels:
        d = depth_image[v, u] # Distance in mm
        if d == 0: continue # Invalid depth

        # Pinhole Camera Math
        z = d / 1000.0
        x = (u - cx) * z / fx
        y = (v - cy) * z / fy

        raw_Points_3d.append([x, y, z])

    return raw_Points_3d
```

The Full Pipeline Execution

```
def main():
    # 1. Perception
    raw_points = process_vision(img, depth, K)

    # 2. Smoothing
    smooth_waypoints = generate_smooth_path(raw_points)
```

```python
# 3. Planning (MoveIt)
(plan, _) = move_group.compute_cartesian_path(smooth_waypoints, 0.01, 0.0)

# 4. Execution
move_group.execute(plan)
```

22. Final Node & Topic Architecture (The "ROS Graph")

You asked: *"list the nodes, what each node will be responsibe for, and what topics they should publish to"*

This is your **System Architecture** for the Thesis.

Node 1: `kinect2_bridge` (Driver)

- **Type**: Third-party Package (iai_kinect2).
- **Responsibility**: Talks to USB hardware, rectifies images, generates PointCloud.
- **Publishes**:
  - `/kinect2/qhd/image_color` (`sensor_msgs/Image`): RGB data for YOLO.
  - `/kinect2/qhd/points` (`sensor_msgs/PointCloud2`): Depth data for Deprojection.
- **Subscribes**: None.

Node 2: `vision_processor_node` (Custom)

- **Type**: Python Node (You write this).
- **Responsibility**: Runs YOLO segmentation, connects pixels to depth, outputs raw 3D points.
- **Subscribes**:
  - `/kinect2/qhd/image_color`
  - `/kinect2/qhd/points`
- **Publishes**:
  - `/vision/raw_seam_points` (`geometry_msgs/PoseArray`): The raw, noisy list of 3D points on the seam.
  - `/vision/debug_mask` (`sensor_msgs/Image`): Visual overlay for you to see what it detected.

Node 3: `seam_path_planner_node` (Custom)

- **Type**: Python Node (You write this).
- **Responsibility**: B-Spline Smoothing, Waypoint Generation, Calling MoveIt.
- **Subscribes**:
  - `/vision/raw_seam_points`
- **Action Client**:
  - Connects to `move_group` via `MoveGroupInterface`.
- **Publishes**:
  - `/planning/smoothed_path` (`nav_msgs/Path`): For visualization in RViz (Red line).

Node 4: `move_group` (MoveIt 2)

- **Type**: Standard MoveIt Node.
- **Responsibility**: Inverse Kinematics (IK), Collision Checking, Trajectory Stitching.
- **Action Server**: Receives requests from `seam_path_planner_node`.
- **Action Client**: Sends trajectories to `real_robot_driver`.
- **Publishes**:
  - `/display_planned_path` (Visuals).

Node 5: `real_robot_driver_node` (Custom)

- **Type**: Python Node (Section 1/2 of this guide).
- **Responsibility**: Interpolates trajectory to 100Hz, streams Serial commands to ESP32.
- **Action Server**: `follow_joint_trajectory`.
- **Subscribes**: None (Receive Action Goals).

- **Publishes**:
  - `/joint_states` (`sensor_msgs/JointState`): Reports "Steps Sent" back to ROS for RViz updates.

Data Flow Summary

`Camera` -> [**Node 1**] -> `(Image/Cloud)` -> [**Node 2**] -> `(PoseArray)` -> [**Node 3**] -> `(Compute Path Call)` -> [**Node 4**] -> `(FollowJointTrajectory Goal)` -> [**Node 5**] -> `(Serial)` -> **ESP32**

23. Master Execution Plan: Running the System

You asked: *"write me a detailed technical plan to run the whole project"*

This is your **"Runbook"**. It defines the exact sequence to bring the 5-node architecture to life.

Phase A: Hardware Startup (T-Minus 1 Minute)

1. **Power Up**: Turn on the 24V PSU for the Steppers.
2. **Connect USBs**:
   - ESP32 (Motors) -> `/dev/ttyUSB0`
   - Kinect v2 -> USB 3.0 Port
3. **Homing (Critical)**:
   - Press the "Reset" button on ESP32.
   - Watch the robot physically move to find home limits.
   - **Verify**: Robot is stationary and holding torque.

Phase B: The Driver Layer (The Foundation)

Launch the hardware drivers first. They usually don't depend on anyone. **Command 1**:

```
ros2 launch parol6_bringup hardware_drivers.launch.py
```

- **What this launches**:
  1. `real_robot_driver_node` (Connects to ESP32).
  2. `kinect2_bridge` (Starts pumping points).
  3. `robot_state_publisher` (Loads URDF).
- **Verify**: `ros2 topic list` shows `/joint_states` and `/kinect2/qhd/points`.

Phase C: The Brain Layer (MoveIt)

Launch the motion planning server. It needs `/joint_states` from Phase B. **Command 2**:

```
ros2 launch parol6_moveit_config moveit.launch.py use_sim_time:=false
```

- **What this launches**:
  1. `move_group` (The big solver).
  2. `rviz2` (Visualization).
- **Verify**: You see the robot in RViz matching reality. No red errors in console.

Phase D: The Application Layer (The Logic)

Launch your custom "Seam Tracking" logic. **Command 3**:

```
ros2 run parol6_vision seam_tracking_app.py
```

- **What this launches**:
  1. `vision_processor` (YOLO).
  2. `seam_path_planner`.
  3. `main_workflow` (The orchestrator).
- **Verify**:
  1. Hold an object in front of the Kinect.
  2. See the "Red Line" (Path) appear in RViz.
  3. Robot moves to trace the line.

Phase E: Emergency Stop (Safety)

- **Software**: Ctrl+C in terminal 3 (Application).
- **Hardware**: Big Red Button cuts 24V power (Stepper Torque off).

24. The Implementation Checklist (What to Build Now)

You asked: *"what steps i should do? what do i need to modify or add to the current setup?"*

Here is your **Gap Analysis**. This is the difference between your current *Simulation-Only* setup and the *Thesis Target*.

Phase 1: Dependencies (The Dockerfile)

Status: Missing Vision Libraries. **Action**: Update your `Dockerfile` to include: * `libfreenect2-dev` (Kinect Driver) * `python3-scipy` (For B-Spline) * `python3-pip -> ultralytics` (For YOLOv8) * `ros-humble-libfreenect2` (ROS Bridge)

Phase 2: Hardware & Firmware

Status: Hardware is unconnected. Firmware is blank. **Action**: 1. **Wiring**: * Wire **3 Limit Switches** to ESP32 (e.g., Pins 12, 13, 14). * Wire **MKS Alarm Pin** to ESP32 (e.g., Pin 27). 2. **Code**: * Write the Arduino Sketch (from Section 1 & 15). * Implement `STEPS_PER_RAD` math (Section 8). * Implement Homing Logic (Section 15). * **Verify**: Open Arduino Serial Monitor. Type `<0,0,0,0,0,0>`. Motors should hold.

Phase 3: The Driver Node (`real_robot_driver.py`)

Status: Does not exist. **Action**: 1. Create file: `PAROL6/parol6/real_robot_driver.py`. 2. Copy code from **Section 2**. 3. Add **Homing Wait** logic (Section 15). 4. Add **PC-Side Interpolation** (Section 10). 5. **Verify**: Run `ros2 run parol6 real_robot_driver`. It should say "Waiting for Arduino…".

Phase 4: The Vision Package (`parol6_vision`)

Status: Does not exist. **Action**: 1. Create new package: `ros2 pkg create --build-type ament_python parol6_vision`. 2. Create `vision_processor.py`: * Subscribe to `/kinect2/qhd/points`. * Implement YOLO + Deprojection (Section 21). 3. Create `seam_path_planner.py`: * Implement B-Spline (Section 21). * Implement `compute_cartesian_path` (Section 20).

Phase 5: The Launch Files

Status: Missing "Bringup". **Action**: 1. Create `PAROL6/launch/real_robot_bringup.launch.py`. 2. It should launch: * `real_robot_driver` * `robot_state_publisher` * `kinect2_bridge` * `rviz2`

Summary of New Files Needed

1. `Dockerfile` (Edit)
2. `firmware.ino` (New)
3. `real_robot_driver.py` (New)
4. `vision_processor.py` (New)
5. `seam_path_planner.py` (New)
6. `real_robot_bringup.launch.py` (New)

25. Do I need to modify `ros2_controllers.yaml`?

You asked: *"should something be modified in ros2_controller? or any other file?"*

A. `ros2_controllers.yaml`

- **Answer**: **NO.** You can ignore this file.

- **Reason**: This file configures the `ros2_control` C++ plugin (for Simulation). Since you are using a **Python Driver** (Section 2), you are *bypassing* the standard `ros2_control` system entirely. Your Python script *IS* the controller.

B. `moveit_controllers.yaml`

- **Answer**: **NO**, but check the name.
- **Reason**: MoveIt expects to talk to an Action Server named: > `/parol6_arm_controller/follow_joint_trajector`

  (You can see this in `parol6_moveit_config/config/moveit_controllers.yaml` under `action_ns`).

  Crucial Implementation Detail: When you write your `real_robot_driver.py`, you **MUST** name your Action Server exactly this:

  ```
  # In real_robot_driver.py
  self._action_server = ActionServer(
      self,
      FollowJointTrajectory,
      '/parol6_arm_controller/follow_joint_trajectory', # MATCH THIS EXACTLY
      self.execute_callback)
  ```

  If you match the name, MoveIt won't know the difference between the Real Robot and the Simulation. It just works.

C. `PAROL6.urdf`

- **Answer**: **NO.**
- The `<ros2_control>` tags in the URDF are for Gazebo/SystemInterface. If you don't launch the `ros2_control_node`, these tags are simply ignored. MoveIt just reads the `<joint>` limits and collision meshes, which are perfect.

D. The Launch File (The Real Change)

- **Answer**: **YES (Create New).**
- You cannot use `ignition.launch.py`. It starts Gazebo.
- You must create `real_robot_bringup.launch.py` that starts:
  1. `robot_state_publisher` (for TF).
  2. `rviz2`.
  3. `real_robot_driver` (Your new node).
  4. **NOT** Gazebo.
  5. **NOT** `ros2_control_node`.

Appendix: Full Code Implementation

You asked: *"can you write the codes that should be in each file inside the file"*

Here are the **Complete Files**. You can copy-paste these directly.

Appendix A: `real_robot_driver.py`

Location: `PAROL6/parol6/real_robot_driver.py` **Description**: The Action Server that allows MoveIt to control the real robot.

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.action import ActionServer, CancelResponse, GoalResponse
from control_msgs.action import FollowJointTrajectory
from sensor_msgs.msg import JointState
```

```python
from trajectory_msgs.msg import JointTrajectoryPoint
import serial
import time
import math
import threading

class RealRobotDriver(Node):
    def __init__(self):
        super().__init__('real_robot_driver')

        # 1. Serial Connection
        # Adjust port as needed (/dev/ttyUSB0 or /dev/ttyACM0)
        try:
            self.ser = serial.Serial('/dev/ttyUSB0', 115200, timeout=0.1)
            time.sleep(2) # Wait for Arduino reset
            self.get_logger().info("Connected to ESP32 via Serial.")
        except Exception as e:
            self.get_logger().error(f"Failed to connect to Serial: {e}")
            # We continue for testing, but in production this should exit
            self.ser = None

        # 2. Homing Wait
        self.wait_for_homing()

        # 3. Action Server
        self._action_server = ActionServer(
            self,
            FollowJointTrajectory,
            '/parol6_arm_controller/follow_joint_trajectory',
            execute_callback=self.execute_callback,
            goal_callback=self.goal_callback,
            cancel_callback=self.cancel_callback)

        # 4. Joint State Publisher (Feedback)
        self.joint_pub = self.create_publisher(JointState, '/joint_states', 10)
        self.timer = self.create_timer(0.05, self.publish_fake_feedback) # 20Hz

        self.current_joints = [0.0] * 6
        self.joint_names = ['joint_L1', 'joint_L2', 'joint_L3', 'joint_L4', 'joint_L5', 'joint_L6']

    def wait_for_homing(self):
        if not self.ser: return
        self.get_logger().info("Waiting for Robot Homing...")
        while True:
            if self.ser.in_waiting:
                line = self.ser.readline().decode().strip()
                if "READY" in line:
                    self.get_logger().info("Robot Homing Complete!")
                    break
            time.sleep(0.1)

    def goal_callback(self, goal_request):
        self.get_logger().info('Received Goal Request')
        return GoalResponse.ACCEPT

    def cancel_callback(self, goal_handle):
        self.get_logger().info('Received Cancel Request')
        return CancelResponse.ACCEPT
```

```
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing Trajectory...')

        # Get the trajectory
        traj = goal_handle.request.trajectory
        points = traj.points

        # Simple Execution Loop (PC-Side Interpolation would go here)
        # For now, we send waypoints directly.
        for point in points:
            if goal_handle.is_cancel_requested:
                goal_handle.canceled()
                self.get_logger().info('Goal Canceled')
                return FollowJointTrajectory.Result()

            # 1. Update Internal State (for feedback)
            self.current_joints = list(point.positions)

            # 2. Format Command: <J1,J2,J3,J4,J5,J6>
            # Convert Radians to Strings
            cmd_str = f"<{','.join([f'{p:.4f}' for p in self.current_joints])}>\n"

            # 3. Send to ESP32
            if self.ser:
                self.ser.write(cmd_str.encode())

                # Check for ALARM/STALL
                if self.ser.in_waiting:
                    resp = self.ser.readline().decode()
                    if "ERROR" in resp:
                        self.get_logger().fatal("ROBOT STALL DETECTED!")
                        goal_handle.abort()
                        return FollowJointTrajectory.Result()

            # 4. Wait for time_from_start (Basic timing)
            # In a real interpolation loop, you'd calculate exact sleep
            time.sleep(0.05) # Simulation of execution time

        goal_handle.succeed()
        result = FollowJointTrajectory.Result()
        return result

    def publish_fake_feedback(self):
        # Reports where we *think* we are (Steps Sent)
        msg = JointState()
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.name = self.joint_names
        msg.position = self.current_joints
        self.joint_pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = RealRobotDriver()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

Appendix B: `firmware.ino`

Location: Arduino/ESP32 Project **Description**: Firmware for ESP32 + MKS SERVO42C (Step/Dir Mode).

```
#include <AccelStepper.h>

// --- Pin Definitions (Adjust for your ESP32 Shield) ---
// Joint 1
#define STEP_1 12
#define DIR_1  14
#define ALM_1  27  // Alarm Input
#define LIM_1  13  // Limit Switch

// ... Repeat for 6 joints ...

// --- Constants ---
// Pre-calculated Steps Per Radian (Example values)
float STEPS_PER_RAD[] = {10185.9, 8500.0, 8500.0, 4000.0, 2000.0, 2000.0};

AccelStepper stepper1(AccelStepper::DRIVER, STEP_1, DIR_1);
// ... Define all 6 steppers ...

const byte numChars = 64;
char receivedChars[numChars];
boolean newData = false;

void setup() {
  Serial.begin(115200);

  // 1. Setup Pins
  pinMode(ALM_1, INPUT_PULLUP);
  pinMode(LIM_1, INPUT_PULLUP);

  // 2. Setup Steppers
  stepper1.setMaxSpeed(20000);
  stepper1.setAcceleration(10000);

  // 3. Homing Routine
  homeRobot();
}

void loop() {
  // 1. Check Safety
  checkAlarms();

  // 2. Read Serial Command
  recvWithStartEndMarkers();
  if (newData) {
      parseData();
      newData = false;
  }

  // 3. Move Motors
  stepper1.run();
  // ... run all ...
}
```

```
void homeRobot() {
  Serial.println("STATUS: HOMING...");

  // Simple Homing: Move Backwards until switch hit
  while(digitalRead(LIM_1) == HIGH) {
    stepper1.setSpeed(-500);
    stepper1.runSpeed();
  }
  stepper1.setCurrentPosition(0);

  Serial.println("STATUS: READY");
}

void checkAlarms() {
  if(digitalRead(ALM_1) == LOW) { // Assuming Active LOW
    Serial.println("ERROR: ALARM_J1");
    while(1); // Halt
  }
}

// --- Serial Parsing Boilerplate ---
void recvWithStartEndMarkers() {
    static boolean recvInProgress = false;
    static byte ndx = 0;
    char startMarker = '<';
    char endMarker = '>';
    char rc;

    while (Serial.available() > 0 && newData == false) {
        rc = Serial.read();
        if (recvInProgress == true) {
            if (rc != endMarker) {
                receivedChars[ndx] = rc;
                ndx++;
                if (ndx >= numChars) ndx = numChars - 1;
            } else {
                receivedChars[ndx] = '\0'; // terminate string
                recvInProgress = false;
                newData = true;
            }
        } else if (rc == startMarker) {
            recvInProgress = true;
        }
    }
}

void parseData() {
    char * strtokIndx;

    strtokIndx = strtok(receivedChars, ",");
    float j1_rad = atof(strtokIndx);

    // Parse others...

    // Move
    stepper1.moveTo(j1_rad * STEPS_PER_RAD[0]);
}
```

Appendix C: vision_processor.py

Location: parol6_vision/vision_processor.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image, PointCloud2
from geometry_msgs.msg import PoseArray, Pose
from cv_bridge import CvBridge
import cv2
import numpy as np
import struct

# Requires: pip install ultralytics
# from ultralytics import YOLO

class VisionProcessor(Node):
    def __init__(self):
        super().__init__('vision_processor')
        self.bridge = CvBridge()

        # Subs
        self.create_subscription(Image, '/kinect2/qhd/image_color', self.rgb_cb, 10)

        # Pubs
        self.seam_pub = self.create_publisher(PoseArray, '/vision/raw_seam_points', 10)

        # self.model = YOLO("yolov8-seg.pt") # Load once

    def rgb_cb(self, msg):
        img = self.bridge.imgmsg_to_cv2(msg, "bgr8")

        # 1. Run YOLO (Dummy logic for now)
        # results = self.model(img)
        # mask = results[0].masks...

        # For testing: Let's pretend we found a red line
        # This is a color-threshold fallback if YOLO isn't ready
        hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, (0, 100, 100), (10, 255, 255))

        # 2. Extend to 3D (Simulated Deprojection)
        points_3d = []
        pixels = np.argwhere(mask > 0)

        # Downsample (take every 100th point to be fast)
        for i in range(0, len(pixels), 100):
            v, u = pixels[i]
            # In real code, lookup depth at (v,u)
            # z = depth_img[v,u]
            # x = (u - cx) * z / fx ...

            p = Pose()
            p.position.x = 0.5 # Dummy
            p.position.y = (u - 300) * 0.001
            p.position.z = 0.2
            points_3d.append(p)
```

```
        # 3. Publish
        out = PoseArray()
        out.header = msg.header
        out.poses = points_3d
        self.seam_pub.publish(out)

def main():
    rclpy.init()
    rclpy.spin(VisionProcessor())
```

Appendix D: seam_path_planner.py

Location: parol6_vision/seam_path_planner.py

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseArray, Pose
from nav_msgs.msg import Path
from moveit_msgs.action import MoveGroup
from rclpy.action import ActionClient
import numpy as np
from scipy.interpolate import splprep, splev

class SeamPathPlanner(Node):
    def __init__(self):
        super().__init__('seam_path_planner')
        self.create_subscription(PoseArray, '/vision/raw_seam_points', self.points_cb, 10)
        self.path_pub = self.create_publisher(Path, '/planning/smoothed_path', 10)

        # MoveIt Client (Conceptual)
        # self.move_group = MoveGroupInterface(...)

    def points_cb(self, msg):
        if len(msg.poses) < 4: return # Need points for spline

        # 1. Extract X,Y,Z
        x = [p.position.x for p in msg.poses]
        y = [p.position.y for p in msg.poses]
        z = [p.position.z for p in msg.poses]

        # 2. B-Spline
        try:
            tck, u = splprep([x, y, z], s=0.01)
            u_new = np.linspace(0, 1, 50) # 50 smooth points
            new_points = splev(u_new, tck)

            # 3. Publish Smooth Path
            path = Path()
            path.header = msg.header
            for i in range(50):
                p = Pose()
                p.position.x = new_points[0][i]
                p.position.y = new_points[1][i]
                p.position.z = new_points[2][i]
                p.orientation.w = 1.0 # Down

                state = PoseStamped()
                state.pose = p
```

```python
                state.header = msg.header
                path.poses.append(state)

            self.path_pub.publish(path)
            self.get_logger().info("Generated Smooth Path")

        except Exception as e:
            self.get_logger().warn(f"Spline Failed: {e}")

def main():
    rclpy.init()
    rclpy.spin(SeamPathPlanner())
```

Appendix E: `real_robot_bringup.launch.py`

Location: `PAROL6/launch/real_robot_bringup.launch.py`

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

def generate_launch_description():
    pkg_parol6 = get_package_share_directory('parol6')
    urdf_file = os.path.join(pkg_parol6, 'urdf', 'PAROL6.urdf')

    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        # 1. Robot State Publisher (TF)
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': robot_desc}],
            arguments=[urdf_file]
        ),

        # 2. Real Robot Driver (Your Node)
        Node(
            package='parol6',
            executable='real_robot_driver.py', # Make sure to install this
            name='real_robot_driver',
            output='screen'
        ),

        # 3. RViz2
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output='screen',
            # Add config file if you have one
        ),

        # 4. Kinect (Optional - if installed)
```

```
    # Node(package='kinect2_bridge', executable='kinect2_bridge'...)
])
```

Appendix F: Prompts for Future Expansion

You asked: *"can you add descriptive prompts for each section in case i wanted to expand on it in another conversation"*

Use these prompts to "Reload Context" when you start a new chat session for a specific task.

1. To Expand the Python Driver (`real_robot_driver.py`)

Prompt: "I am building a custom ROS 2 Action Server in Python for a 6-DOF robot arm (PAROL6). **Context:** * **Role**: It acts as a bridge between MoveIt 2 (`FollowJointTrajectory` action) and an ESP32 microcontroller. * **Communication**: It uses USB Serial (115200 baud) to send position commands <J1,J2,J3,J4,J5,J6> at 20-50Hz. * **Current State**: I have a basic skeleton that connects and accepts goals. **Task:** Please write the python code to implement [Interpolation / Error Handling / Feedback Publishing]. specifically, I need to read the 'ALARM' message from the ESP32 and cancel the MoveIt goal immediately if a stall is detected."

2. To Expand the Firmware (`firmware.ino`)

Prompt: "I am writing Arduino/C++ firmware for an ESP32 controlling 6 Stepper Motors. **Context:** * **Hardware**: 6x MKS SERVO42C (Closed Loop drivers) using STEP/DIR interface. * **Library**: `AccelStepper`. * **Protocol**: It parses serial strings like <0.5, 1.2, ...> (Radians) and converts them to Steps. **Task:** Please output the C++ code to implement [Homing Sequence / Alarm Monitoring]. I have Limit Switches on Pins 13,14,15 (Input Pullup). When the `home()` command is received, move each motor slowly backwards until the switch triggers, then set position to 0."

3. To Expand the Vision System (`vision_processor.py`)

Prompt: "I am developing a Computer Vision node for a Robotic Welding application using ROS 2 and Kinetic v2. **Context:** * **Input**: `/kinect2/qhd/image_color` (RGB) and `/kinect2/qhd/points` (PointCloud2). * **Goal**: Detect a 'Seam' (line) to follow. **Task:** Please write a Python ROS 2 node that uses **YOLOv8** to detect a target object, gets its Bounding Box, and then extracts the 3D coordinates (X,Y,Z) of the center of that box using the PointCloud data. Publish the result as a `geometry_msgs/PoseStamped`."

4. To Expand Path Planning (`seam_path_planner.py`)

Prompt: "I need to generate a smooth welding path for a 6-DOF robot arm using MoveIt 2. **Context:** * **Input**: A noisy list of 3D points `geometry_msgs/PoseArray` derived from a camera. * **Requirement**: The robot must move continuously through these points (Cartesian Path), not stop-and-go. **Task:** Please write the Python code using `scipy.interpolate` to fit a **B-Spline** through these noisy points and generate 50 evenly spaced Waypoints. Then, show how to send these waypoints to `move_group.compute_cartesian_path()`."

5. To Expand Maintenance/Deployment

Prompt: "I have a working ROS 2 robotic system (Parol6) running in a Docker container `parol6-ultimate`. **Task:** I need to export this work for my Thesis defense. Please guide me on how to: 1. Commit the current container state to a new Image. 2. Save that Image to a `.tar` file for USB transfer. 3. Write a `deployment.sh` script that installs Docker and runs the image on a fresh Ubuntu laptop without requiring internet access."