ECE 445

SENIOR DESIGN LABORATORY

FINAL REPORT

Long-horizon Task Completion with Robotic Arms by Human Instructions

---

Team #25

BINGJUN GUO (bingjun3)

QI LONG (qilong2)

QINGRAN WU (qingran3)

YUXI CHEN (yuxi5)

(alphabetically)

Sponsor: Gaoang Wang, Liangjing Yang

TA: Tielong Cai, Tianci Tang

May 18, 2025

Abstract

This project developed a robotic system to assist individuals with limited mobility by understanding human instructions and autonomously performing long-horizon tasks like table cleaning. The system integrates Perception (Vision Language Models like Qwen-VL, Grounded SAM), Planning (VLMs for task/motion planning), Control (Raspberry Pi 5, ROS2), and Action (custom force-feedback gripper) modules. Key achievements include successful module integration, advanced AI implementation (90% object identification accuracy), and custom hardware development (force-sensing PCB, lead screw gripper). This work demonstrates a viable approach for creating intelligent robotic assistants capable of complex instruction interpretation and real-world interaction, advancing autonomous and helpful robotics.

Contents

1 Introduction

1.1 Problem Statement

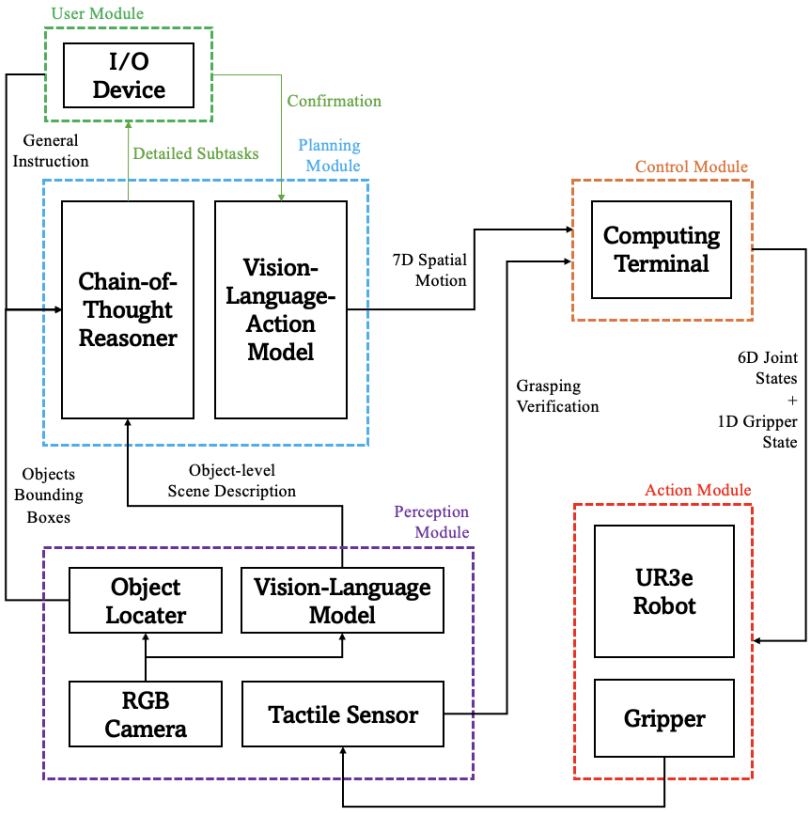

Figure 1: The overall block diagram.

The application of robotic arms for assisting with daily tasks is becoming increasingly important, particularly for individuals with limited mobility, such as the elderly, children, and people with disabilities. Many simple activities—like cleaning a table—can be difficult for these groups to perform independently. While robotic arms have the potential to improve quality of life and reduce the burden on caregivers, they must be able to understand human instructions, adapt to changing environments, and provide robust, consistent performance across multiple steps.

Existing robotic systems often struggle with breaking down and executing long-horizon tasks, especially in dynamic, real-world settings. Most are limited to predefined scenarios and cannot effectively respond to unforeseen changes or integrate real-time feedback. To

address these limitations, we propose a comprehensive framework that integrates perception, planning, and action for autonomous task completion.

In this project, we focus on developing a robotic system that assists people with limited mobility in performing everyday tasks, such as cleaning a table, by following human instructions and adapting to real-world conditions.

1.2 High-Level Requirement List

Perception Accuracy The system must achieve at least 90% accuracy in identifying and localizing target objects within its operating environment.

Planning Efficiency The robot must generate actionable multi-step operation plans within 5 seconds after receiving human instructions.

Execution Robustness The robotic arm must successfully complete at least 90% of attempted long-horizon tasks without collisions or critical errors under varying environmental conditions.

2 Design

2.1 Perception Module

2.1.1 Object identifier

The Perception Module is designed to identify objects of interest in the scene and locate them. Specifically, our design consists of a vision language model[1], [2], which extends the emerged power of large language models[3] to the modalities of both text and image, for scene description, and a so-called vision foundation model, Grounded SAM[4] for object locating.

2.1.2 High-level Perception: Scene description

Given a scene image as input, a large Vision Language Model (VLM)[2] is prompted through ModelScope[5] API to describe the scene and identify objects in the scene.

- Base Model: Qwen/Qwen2.5-VL-7B-Instruct[2] from ModelScope[5].

- Prompt (text input): to make the best practice of a VLM, the prompt is designed to be as simple as possible, but also to be able to ensure a formatted output for further processing. (Figure 2)
- Combined Input: image of the scene captured from the RGB camera.
- Output: a list of names of objects in the image.

> Please identify the objects in the image, output as a list in this format:
> <objects>[object 1, object 2, object 3, ...]</objects>

Figure 2: Scene Descriptor prompt.

2.1.3 Low-level Perception: Fine-grained object stability

Figure 3: Example output of Grounded SAM; input labels: person, plant, bin

Given a scene image and a list of object names in the scene, our Object Detector identifies them in the image, returns masks of the objects in the image, and the corresponding bounding boxes of them. In our design, we take advantage of the effective open-world vision representations learned by the Distillation with No Labels(DINO) model[6], which utilizes the Transformer[7] backbone and was trained in a self-supervised manner. Specifically, we deploy the *Grounded Segment-Anything (Grounded SAM)*[4] on our server that listens for a list of object names and an image sent from our control module in binary form, and returns the masks and boxes for each identified object. Compared with traditional

vision models such as You Only Look Once (YOLO)[8], the model we deployed is capable for open-set identification, that is, for any object in our daily life, rather than a small group of objects limited by the training process. The pipeline is established with python packages of *FastAPI*[9] and *Uvicorn*[10], which will be introduced in detail in following sections.

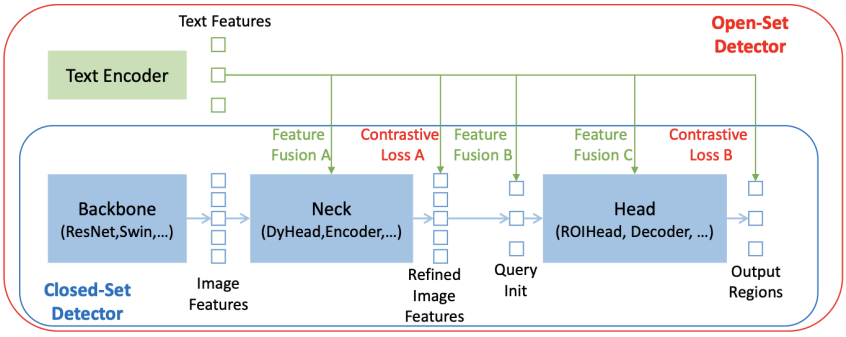

Figure 4: How Grounding DINO becomes open-set

2.2 Planning Module

INSTRUCTION: Considering the given scene, you need to select actions from a given list to generate a plan of steps to accomplish the task.

Task: {}

Action List: {}

Guideline:
1. You should condition your thinking based on the given scene (image), fully consider the physical constraints (e.g., relative positions of objects) to include necessary intermediate steps.
2. You should firstly decompose the Task into subtasks in a step-by-step way and then generate the plan.
3. Each step of your plan should include exactly one item from the Action List.
4. Each "grasp" action should be followed by a "move" action, since you can not grasp more than one objects at a time.
5. Finally, output your plan as a list in this format:
<plan>[Action 1, Action 2, Action 3, ...]</plan>

Figure 5: Planning VLM prompt.

2.2.1 Structured High-level Planning

The Planning Module is designed to generate a plan of robot arm actions to decompose the top-level user instruction and accomplish it. Briefly speaking, it uses a VLM to reason upon information from user (the instruction), from scene (the object locator output) and make a plan from a composed action list from object list (the scene descriptor output).

- Base Model: Qwen/Qwen2.5-VL-32B-Instruct from ModelScope<**empty citation**>
- Input: user instruction, object list from Scene Descriptor, and image of scene captured from the RGB camera and labeled by Object Locator.
- Output: a list of predefined action candidates as a plan.
- Prompt: since the prompt needs to guide the VLM to reason about the physical world from 2D photos, the intricate constraints on "Grasp" and "Move" actions, and the final output list format, it is designed to include a guideline for detailed instructions. Besides, inspired by the Chain-of-Thought practice of Large Language Models (LLMs), the prompt includes a "step-by-step way" thinking keyword 5. Where the Task is user input, the action list is a combination between "Grasp" and objects as well as "Move to" and given locations.

2.2.2 Low-level (kinematics) planning

Inverse kinematics is the process of determining the joint parameters (e.g., angles) of a robotic manipulator that achieve a desired position and orientation of its end effector. Mathematically, given the desired end effector location $\mathbf{T}_{\text{desired}}$ sent from the planning module, the inverse kinematics problem solves for the joint variables $\mathbf{q}$ such that:

$$\mathbf{T}(\mathbf{q}) = \mathbf{T}_{\text{desired}}$$

where $\mathbf{T}(\mathbf{q})$ is the forward kinematics function that maps joint variables to the Cartesian coordinates of the end effector. In lab5 of the course ECE470[11], we have derived a close form inverse kinematics process according to the Denavit–Hartenberg parameters[12] of UR3e[13] as follows:

- θ_1: Base rotation angle

  First we represent the length $l_2 - l_4 + l_6$ as l_p, the perpendicular offset from P_{cen} to

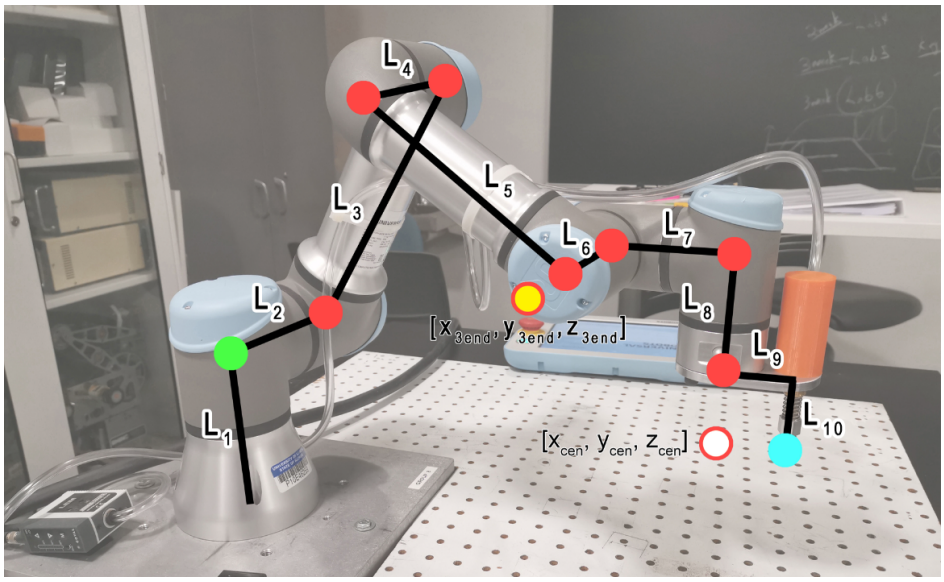

Figure 6: Inverse-kinematics solution illustration for the UR3e robot arm.[11]

P_{3end}. Then we have the following equations:

$$\theta_1 = \arctan 2(y_{cen}, x_{cen}) - \arcsin(\frac{l_p}{\sqrt{x_{cen}^2 + y_{cen}^2}})$$

in which $\arctan 2$ is the inverse tangent function, which returns the angle in radians between the positive x-axis and the point (x, y).

- x_{3end}, y_{3end}, and z_{3end}: From the system of equations above, we have the following equations:

$$\begin{aligned} x_{3end} &= x_{cen} + l_p \sin(\theta_1) - l_7 \cos(\theta_1) \\ y_{3end} &= y_{cen} - l_p \cos(\theta_1) - l_7 \sin(\theta_1) \\ z_{3end} &= z_{cen} + l_8 + l_{10} \end{aligned}$$

- θ_2 and θ_3: Using law of cosines and sines, we have the following equations:

$$\begin{aligned} l_3 \sin(-\theta_2) + l_1 - l_5 \sin(\theta_3 + \theta_2) &= z_{3end} \\ l_3 \cos(\theta_2) + l_5 \cos(\theta_3 + \theta_2) &= \sqrt{x_{3end}^2 + y_{3end}^2} \end{aligned}$$

- θ_4: Maintains horizontal end effector

$$\theta_4 = -(\theta_2 + \theta_3)$$

- θ_5: Keeps end effector vertical

$$\theta_5 = -\frac{\pi}{2}$$

- θ_6: Provides yaw orientation

$$\theta_6 - \theta_1 + \theta_{yaw} = \frac{\pi}{2}$$

2.3 Control Module

The Control Module is the central hub for managing communication, synchronization, and information exchange between all other modules in our system. This module is built around a Raspberry Pi 5 [14] (Figure 7), which runs Ubuntu 24.04 LTS and the ROS2 (Robot Operating System 2) [15], enabling communication between sensors, cameras, motors, and the model server in real-time.

We chose the Raspberry Pi 5 as the central processing unit due to its enhanced computational power and interface support. Featuring a faster processor and improved connectivity options. It efficiently manages real-time data to and from the UR3e robot arm and the model server. It also ensures seamless integration with peripheral devices through interfaces like Universal Asynchronous Receiver-Transmitter (UART) connection with motor, General-purpose Input/Output (GPIO) connection with sensor, and Camera Serial Interface (CSI) connection for camera.

2.3.1 Peripheral Connection - Force Sensor

Force Sensor Placement To detect whether an object is properly grasped, a FSR402 force sensor is positioned on the inner surfaces of the gripper finger, where it will make direct contact with the object.(Figure 8) The sensor outputs an analog signal, which is proportional to the amount of force applied to the object.

PCB Design The analog signal will be processed by a simple custom PCB, which outputs a digital signal to indicate whether the object has been properly grasped. The board converts the analog voltage from an FSR402 force sensor into a clean digital signals and

Figure 7: Raspberry Pi 5

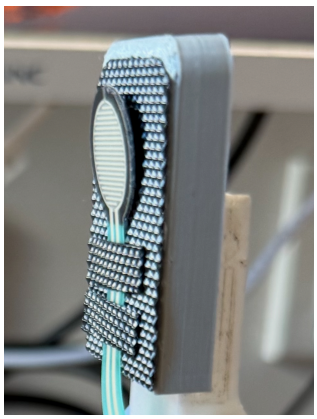

Figure 8: FSR Sensor Installation on Gripper

provides visual feedback via LEDs. The circuit schematics and the PCB layout are shown in Figure 9 and Figure 10. The 3D view of the PCB board is shown in Figure 11: The key components are:

- **FSR Sensor Input:** The FSR402 sensor is wired as a pull-up voltage divider with R1 ($10\,k\Omega$). As force increases, the FSR resistance drops and the divider node voltage rises.
- **Decoupling and Filtering Capacitor (C1, C2):** A $100\,nF$ capacitor C1 close to U1's VCC and GND stabilizes the power supply. Another $100\,nF$ capacitor C2 at the sensor input node suppresses high-frequency noise before the comparator.
- **Threshold Comparator (U1 – LM393):**
  - *Inverting Input* (pin 2) receives the filtered FSR voltage.
  - *Non-inverting Input* (pin 3) is tied to a threshold network with adjustable potentiometer R2 ($10\,k\Omega$).
  - When the FSR voltage exceeds the threshold (set by R2), the comparator output toggles through the OUT/A pin. (pin 1)
- **LED Indicators:**
  - LED1 with R3 ($1\,k\Omega$) illuminates when the circuit is working properly.
  - LED2 (red) with R4 ($1\,k\Omega$) illuminates when the comparator output is high (pressure detected).

In summary, a pull-up resistor feeds the sensor voltage into the LM393 comparator, whose other input is set by a potentiometer. When the sensor voltage exceeds the threshold, the comparator output toggles.

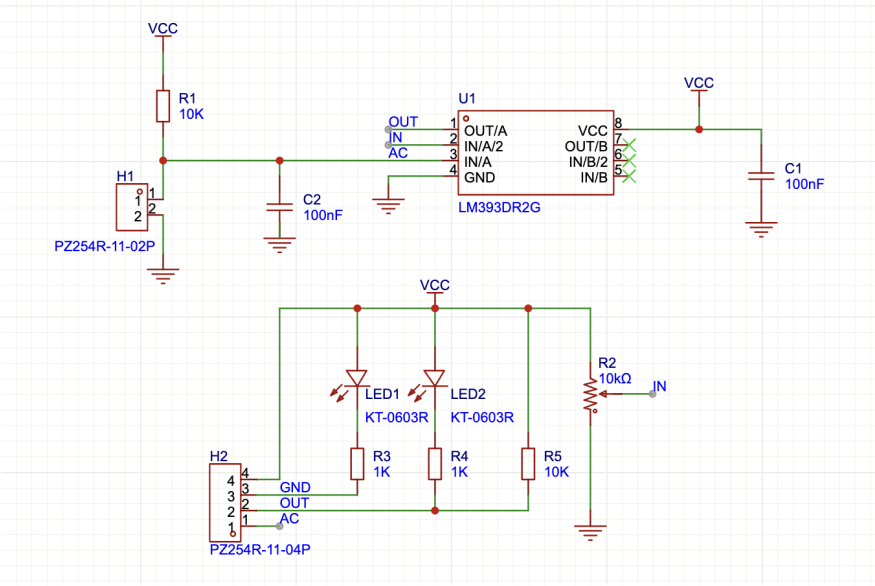

Figure 9: PCB Board Circuit Schematics

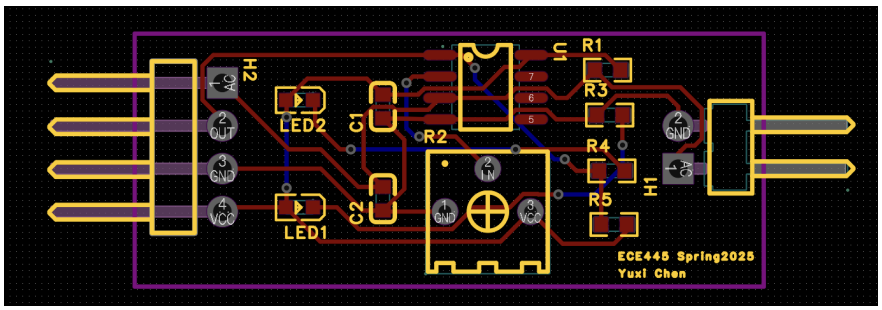

Figure 10: PCB Board Layout

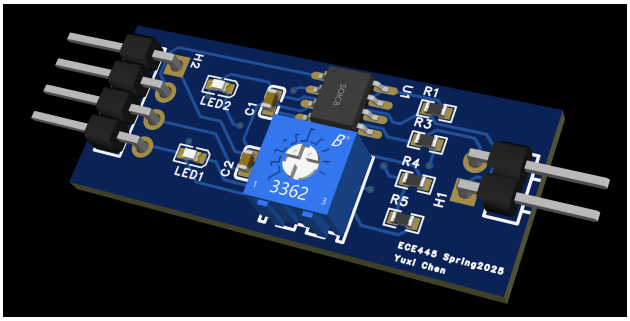

Figure 11: PCB Board 3D View

Sensor Signal Processing The digital signal is transmitted to the Raspberry Pi 5's GPIO port. A custom ROS2 package processes this signal with debounce to prevent false triggers. The package's node publishes the sensor data to a ROS topic, allowing other modules to access it via the ROS2 interface. The main portion of the code for this package is provided in Appendix A.

2.3.2 Peripheral Connection - Motor

We control the gripper using a stepper motor and a dedicated control board. The stepper motor is powered independently via a separate power adapter. The control board interfaces with the Raspberry Pi 5 through a UART (serial) connection.

The control board accepts various commands via the UART interface, including enabling and disabling of the motor, absolute and relative position moves, speed-based jog commands, emergency stop, etc.

A custom ROS 2 package is developed to manage motor control. When the node initializes, it establishes a serial connection to the motor controller and sets the motor's initial position to zero. The */grasp_object* service commands the motor to jog clockwise at a fixed speed until a force sensor detects contact. Once triggered, the force sensor node publishes a message to the corresponding ROS topic and stops the motor. The */release_object* service moves the motor counter-clockwise back to position zero, resetting the gripper to its initial open state.

To protect the hardware, the node enforces a global movement range limit. If the motor attempts to move beyond this predefined range, it immediately stops the motor regardless of ongoing commands, preventing potential damage to the gripper. The main portion of the code for this package is provided in Appendix A.

2.3.3 Device Connection - Camera, UR3e and Server

Camera Setup We also used a 4K Pi camera with field of view $\mathrm{H}108^\circ \times \mathrm{V}92^\circ$ as the primary visual input. It connects directly to the Raspberry Pi 5's CSI port, which natively supports the camera without the need for additional drivers.

UR3e Robot Arm Connection We connected the Raspberry Pi 5 to the UR3e robot via Ethernet by following the official guidelines [16]. By setting up the recommended ROS 2 packages on the Raspberry Pi, we can establish a reliable communication channel between the Pi and the UR3e robot arm. This setup allows us to send motion commands

and receive feedback directly from the robot's controller in real time. As a result, we can easily control the robot's movements, monitor its state, and integrate it into the overall system.

Remote Server Connection We created straightforward APIs on the server to enable seamless communication between the ROS2 framework and the model server. Specifically, the backend uses FastAPI[9], a modern Python web framework focused on speed. FastAPI's automatic data validation and async capabilities made it particularly suitable for handling real-time image uploads and predictions efficiently. The server runs on Uvicorn[10], a lightning-fast ASGI server, which ensures low-latency, concurrent processing of incoming requests. Through these APIs, the ROS2 system can upload images directly to the model server and receive inference results in real time, allowing for immediate feedback and smooth integration with the robotics pipeline.

2.3.4 Module Overall Workflow

The overall system workflow integrates multiple hardware and software modules to enable seamless, real-time robotic operation. At the center of this architecture is the Raspberry Pi 5, which orchestrates data flow and command execution among sensors, actuators, the UR3e robotic arm, and the model server.

The process starts with the camera capturing images of the robot's workspace. These images are processed locally and then transmitted to the model server via the ROS2 framework using a FastAPI-based API. The model server performs image analysis and returns inference results in real time.

Based on the inference results, the Raspberry Pi 5 sends movement commands to the UR3e robotic arm. When the arm reaches the target position, the system calls the *grasp_object* service to activate the gripper. The gripper closes until the FSR402 force sensor detects contact with the object, signaling a successful grasp. The Raspberry Pi 5 then issues further movement instructions to the robot arm. Upon reaching the final target location, the system calls the *release_object* service to open the gripper and release the object.

This integration of the modules ensures synchronized operation and robust error handling, resulting in a reliable robotic system. Figure 12 illustrates the overall workflow of the system.

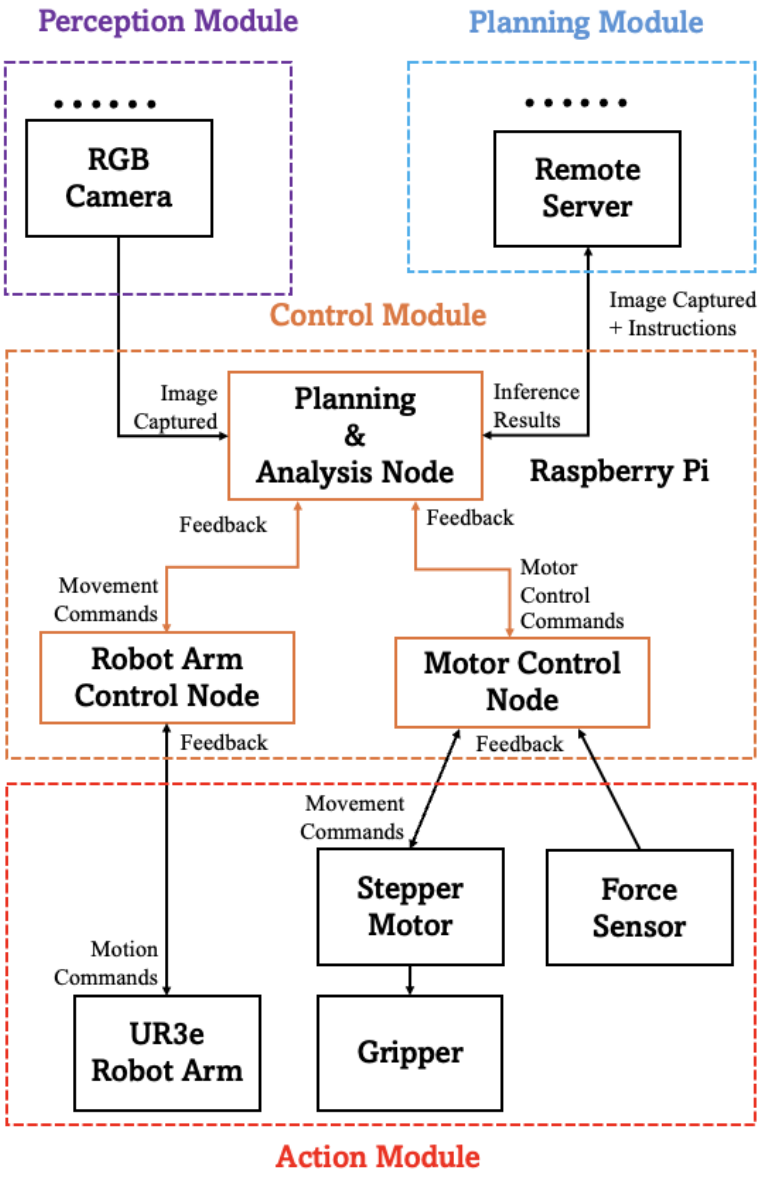

Figure 12: Control & Action Module Workflow

2.4 Action Module

2.4.1 Design Procedure

In this project, a custom gripper needed to be designed and attached to the end-effector. The default end-effector provided was a suction cup, which was limited to picking objects with flat upper surfaces. This limitation became problematic when attempting to handle rounded objects, such as apples or bottles, or items with soft surfaces, like kitchen sponges.

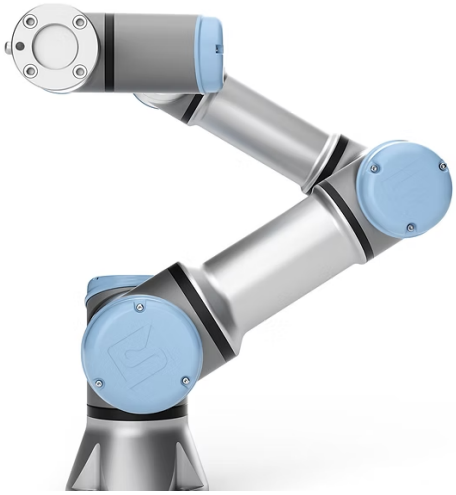

Figure 13: UR3e

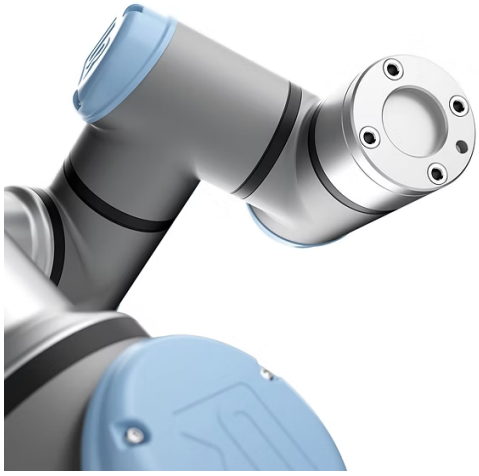

Figure 14: UR3e End Effector

Initially, we proposed a mechanism depicted in Figure 15. This gripper utilized a servo motor to actuate two linkages connected directly to two parallel "fingers". These fingers were constrained by sliders, limiting their motion to one direction (grasping or releasing). Although simple and straightforward, this design presented a significant shortcoming: the mechanism was unable to supply adequate torque, as it translated the servo motor's torque directly into linkage force without amplification.

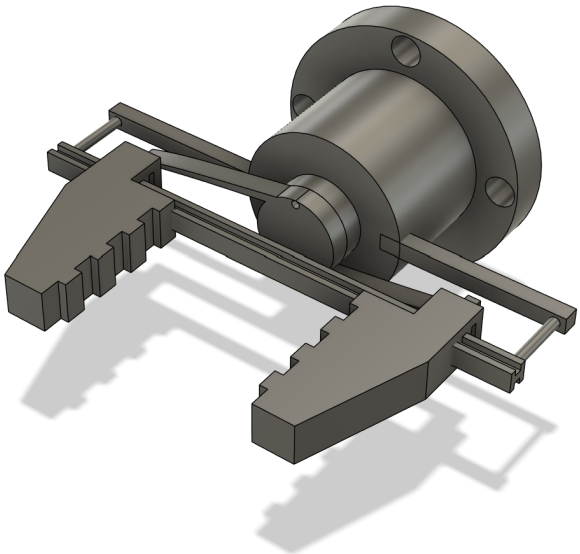

Figure 15: Gripper Version 1

Given a servo torque $\tau = 0.016\,\text{Nm}$, finger length $l = 0.05\,\text{m}$, and friction factor $\mu = 0.5$,

the grasping force F can be calculated as:

$$F = \frac{\tau}{l \times \mu} = \frac{0.016}{0.05 \times 0.5} = 0.64\,\text{N}$$

The force is too small to grasp a kitchen sponge. To address this torque limitation, two potential solutions were considered:

1. Implementing a torque-amplifier, such as a gear set, to enhance the effective torque transmitted to the fingers.
2. Redesigning the gripper mechanism entirely.

Due to the constraints imposed by the limited precision of the 3D printers available in our laboratory—a valuable lesson learned from ME371—we opted against adding complexity via gear systems. Instead, we pursued a more robust and straightforward alternative: a lead screw-based mechanism as shown in Figure 16.

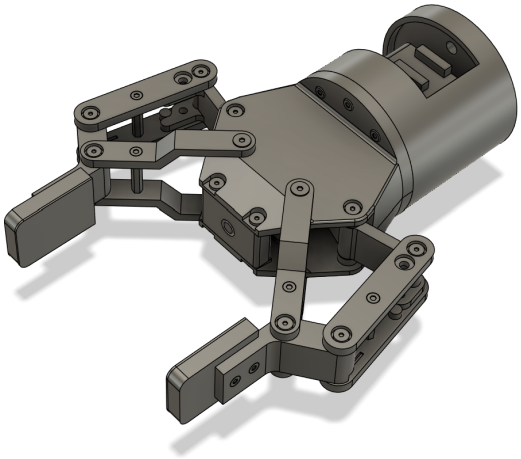

Figure 16: Gripper

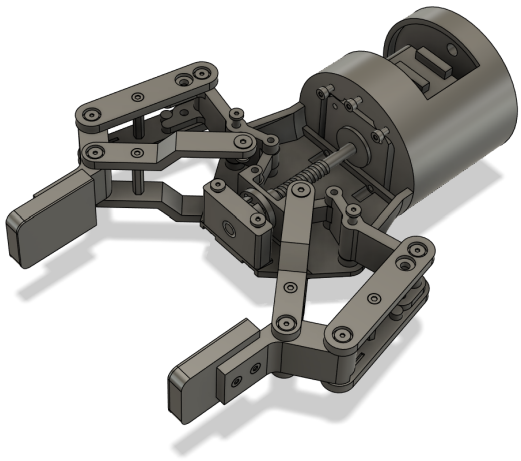

Figure 17: Gripper Internal View

2.4.2 Design Details

The lead screw mechanism is central to our revised gripper design, converting the rotational motion of the stepper motor into precise linear displacement. This design provides a mechanical advantage, significantly amplifying the applied torque.

The stepper motor selected has a torque of $0.43\,\text{Nm}$ and lead screw pitch (lead) of $2\,\text{mm}$.

Considering an efficiency factor of approximately 20%, the effective linear force provided by the lead screw is calculated as:

$$F_{\text{axial}} = \frac{2\pi\tau \times \eta}{\text{Lead}} = \frac{2\pi \times 0.43 \times 0.2}{0.002} \approx 270.18\,\text{N}$$

$$F_f = \mu \times F_{\text{axial}} = 0.5 \times 270.18 \approx 135.09\,\text{N}$$

This effective force of approximately $135.09\,\text{N}$, accounting for efficiency losses, should in theory be sufficient for robust gripping in typical application scenarios. However, in practice, the maximum reliable gripping force is typically lower due to additional mechanical losses, compliance in the parts, and imperfect force transmission throughout the mechanism. Therefore, we conservatively set the maximum grasping weight to $0.25\,\text{kg}$ for safe and reliable operation.

2.4.3 Workflow

The workflow of the Action Module is shown in Figrue 12 and described as follows:

First, the **Robot Arm Control Node** transmits motion commands to the UR3e Robot Arm to execute the trajectories generated by the Planning Module. Once the UR3e Robot Arm reaches the target position, it provides feedback to the Control Module, indicating successful positioning.

Subsequently, the **Motor Control Node** issues movement commands to the stepper motor. The stepper motor actuates the gripper mechanism to either grasp or release objects as required. During the grasping process, the force sensor continuously monitors the applied gripping force. When the sensor detects that a sufficiently large force has been achieved—indicating a successful grasp—it sends feedback to the Motor Control Node to halt the stepper motor, thereby preventing excessive force application.

For object release, the Motor Control Node instructs the stepper motor to move the gripper to a predefined "safe" open position. Once this position is reached, the stepper motor notifies the Motor Control Node to stop further actuation.

This closed-loop workflow ensures accurate and safe execution of both grasping and releasing actions, with continuous feedback at each step to enhance reliability and protect both the hardware and the objects being manipulated.

3 Verification

3.1 Verification of PCB Circuit

To ensure reliable operation of the PCB circuit, we performed several verification tests on the LM393 comparator, the FSR402 sensor, and the LEDs.

3.1.1 Comparator Threshold Accuracy

We placed the FSR402 sensor on the input node and varied the potentiometer R2 across its full range to test the comparator's toggle point:

- **Resolution:** A $1\ k\Omega$ change in R2 produced an average 0.32 V shift in the toggle point.
- **Repeatability:** Cycling R2 back and forth over its range multiple times yielded a maximum error of $50mV$ in the measured threshold, demonstrating stable comparator performance despite mechanical variation in R2.
- **Sensor-to-Threshold Correlation:** When a 500 g weight was applied to the FSR, the comparator toggled at R2 about $4.5\ k\Omega$. Increasing weight to 1 kg required R2 about $3.0\ k\Omega$, confirming the expected inverse relationship between sensor resistance and threshold setting.

These results verify that by adjusting R2, the LM393 comparator can be tuned to detect specific grasp forces, and the system's resolution is sufficient for our application.

3.1.2 LED Indicator Performance

We measured LED currents and illumination thresholds to verify visual feedback:

- **LED1** turns as long as the VCC and GND pins are correctly connected.
- **LED2** turns on when comparator output "1".

Both LEDs switch cleanly with the comparator output and are clearly visible under normal lab lighting.

Overall, these tests validate that the LM393 comparator, the FSR sensor, and the LED operate within the design specifications.

3.2 Verification of Perception Module

3.2.1 RGB Camera

The RGB camera is successfully connected to the Raspberry Pi 5 via a csi cable[17]. The pictures are taken with the Linux library `libcamera`. The collected scene images can be transmitted to the Perception Module via ssh connection and python packages fastapi[9] and Uvicorn[10]. The whole process is verified to be able to complete in 0.1 second.

3.2.2 Force Sensor

The FSR Sensor is successfully installed on gripper and can adjust the robot arms' motor based on signals from it. It can assist gripper's action when gripping an object accordingly as expected.

3.2.3 Object Locator

Our Object Locator is evaluated on a common-objects test dataset. The results 18 19 show the effectiveness of this model, reaching an accuracy of 90% on the identification of 860 objects of 24 common classes. For a single image image of DINO timing pending

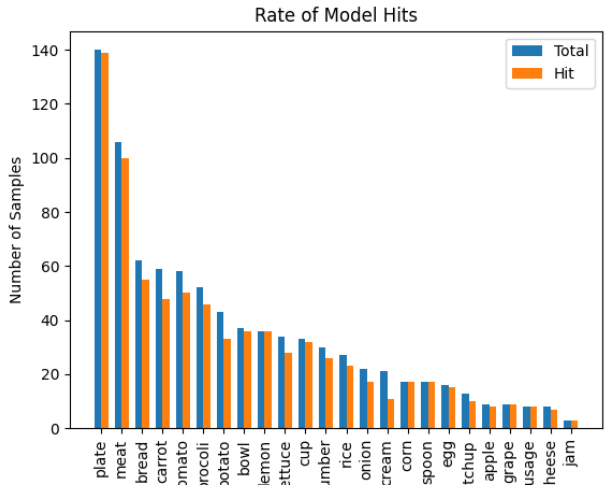

Figure 18: Grouding DINO evaluation result

Figure 19: Grouding DINO annotation example

3.3 Verification of Planning Module

3.3.1 Task Decomposition

The Planning Module can correctly decompose high-level user instructions into subtasks that are ordered in valid and feasible logic. A case study 20 shows its effectiveness, where the task is decomposed correctly.

3.3.2 Structured Plan generation

The Planning Module can generate plans that are consistent with the current scene, as described by the Perception Module. The case study 20 shows its effectiveness, where the plan is generated feasibly and consistent to the user instruction and current scene.

Figure 20: A planning module case study.

<User Instruction> Give me two fruits after giving me a coke.
<Output> To accomplish the task of giving two fruits after giving a coke, we need to follow the sequence of actions carefully.
Here's the step-by-step decomposition of the task and the corresponding plan:
Task Decomposition:
1. **Pick up the coke**: The user needs a coke first.
2. **Move the coke to the user**: Deliver the coke to the
user.
3. **Pick up the banana**: After delivering the coke, pick up one fruit (banana).
4. **Move the banana to the user**: Deliver the banana to
the user.
5. **Pick up the apple**: Pick up the second fruit (apple).
6. **Move the apple to the user**: Deliver the apple to the user.
Plan Generation:
Based on the action list and the guidelines, the plan will be:
```xml
<plan>['grasp coke', 'move to user', 'grasp banana', 'move
to user', 'grasp apple', 'move to user']</plan>
```
Explanation of Each Step:
1. **'grasp coke'**: Pick up the coke can from the table.
2. **'move to user'**: Deliver the coke to the user.
3. **'grasp banana'**: Pick up the banana from the table.
4. **'move to user'**: Deliver the banana to the user.
5. **'grasp apple'**: Pick up the apple from the table.
6. **'move to user'**: Deliver the apple to the user.This plan ensures that the task is completed in the correct order, adhering to
the constraints and guidelines provided.

3.3.3 Motion Planning

The planner can successfully produce physically executable motion plans for the robot. The output action tuples can be further transformed into coordinates of movement captured by the camera. This requirement was initially verified in the Gazebo simulation environment and then in the reality.

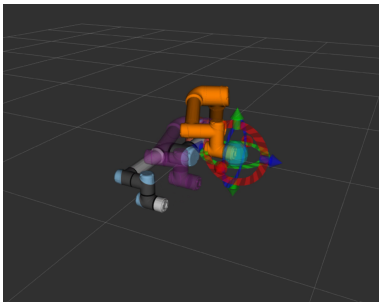

Figure 21: Motion Planning in Simulation.

3.3.4 Plan Display (previously user interface)

Before robot's execution, the planning module will present planned subtasks to the user and incorporate confirmation before execution. As shown by the case 20, the generated plan is displayed on the screen and if the generation is not in the given format of action tuples, it will regenerate the plan.

3.3.5 Contribution to the System

The Planning Module can effectively interface with upstream (perception) and downstream(control)modules, which is shown in the final demo or demo video.

3.4 Verification of Action Module

3.4.1 Grasping Size Range

To verify the size range, the gripper was used to grasp objects with widths of 1 cm, 3 cm, and 5 cm. In each test, the gripper was able to reliably grasp and hold objects of these sizes without slipping or dropping. This confirms that the gripper meets the expected operational range for object width.

3.4.2 Grasping Mass Limit

To assess the load capacity, the gripper was tested on objects weighing 0.1 kg, 0.2 kg, and 0.25 kg. The gripper securely lifted and transported each weight without any slippage or failure. These results confirm that the gripper can safely handle objects up to 0.25 kg as intended.

3.4.3 Stop/Safe Operation

For safety verification, the emergency stop function was tested using the robot control panel. During both grasp and release operations, pressing the emergency button immediately paused the robot and stopped all motion. In each case, the gripper maintained a secure hold on the object or safely ceased operation, demonstrating reliable stop and safe handling capability.

3.4.4 Closed-loop Control Response

The closed-loop control feature was verified by interacting directly with the force sensor. When the force sensor was pressed, or when the gripper successfully grasped an object, the stepper motor stopped automatically as designed. This confirms that the closed-loop feedback system is responsive and prevents over-compression, protecting both the gripper and the object.

4 Cost and Schedule

4.1 Cost Analysis

Cost Item	Unit Cost (USD)	Quantity	Total Cost (USD)
Hardware Components			
Raspberry Pi 5	$80.00	1	$80.00
Force Sensors (FSR402)	$8.50	1	$8.50
UR3e Robotic Arm	$0.00	1	$0.00
Stepper Motor	$7.72	1	$7.72
Controller	$7.83	1	$7.83
Power Supply	$4.38	1	$4.38
Screws and Nuts	$3.47	1	$3.47
Anti-Slip Silicone Tape	$1.09	1	$1.09
Torsion Spring	$1.82	1	$1.82
Bearing	$0.22	10	$2.20
PCB Board	$25.00	1	$25.00
		Hardware Subtotal	$142.01
Labor			
Engineering Labor	$15.00/hr	200	$3,000.00
		Total Project Cost	**$3142.01**

Table 1: Detailed Cost Breakdown

4.2 Schedule

Table 2: Project Timeline and Team Responsibilities

Week	Qi Long	Bingjun Guo	Yuxi Chen	Qingran Wu
Phase 1: Research & Investigation				
3/17	Literature Review: Guiding Long-Planning with VLM	Literature Review: Hierarchical Planning Foundation Model	Literature Review: Optimal force sensor placement for gripper design	Investigate common gripper design for robot arm
3/24	Literature Review: VLA models, including OpenVLA	Literature Review: Improved VLA strategies, including ECoT	Investigate PCB design for sensor integration	Build the initial CAD model of the gripper
3/31	Set up server environments and experiment on OpenVLA model	Set up server environments and validate inverse kinematic method	Collaborate with Qingran sensor placement and gripper design	3D-print, assemble, test the gripper, and collaborate with Yuxi on sensor placement
Phase 2: Design				
4/7	**Team Collaboration: Design Document Composition**			
	Perception Module	Planning Module	Planning Module	Action Module
Phase 3: Implementation				

Continued on the next page

Table 2 – Continued from previous page

Week	Qi Long	Bingjun Guo	Yuxi Chen	Qingran Wu
4/14	Implement ECoT pipeline and experiment on ECoT-VLA model	Experiment the inverse kinematic method in lab environment	Create preliminary PCB layout	Improve the design and continue to assemble and test the gripper
4/21	Implement Grounding DINO calling code and verify on test set	Implement the motion driver (3d input) for UR3e with python package for ROS2	Write ROS nodes for force sensors	Complete the assembly and test the functionality
4/28	Test Object Locator and ECoT-VLA model	Review on Grounded SAM; set up the vision pipeline with fastapi and uvicorn	Execute end-to-end system testing on ROS and finalize PCB design	Figure out the connection between the gripper and the robot arm
Phase 4: Testing & Integration				
5/5	**Integration Testing**			
	Software workflow testing with Bingjun	Software workflow testing with Qi and Yuxi	Hardware workflow testing with Qingran and Bingjun	Hardware workflow testing with Yuxi
5/12	**Full System Implementation**			
5/19	**System Testing and Debugging**			

5 Conclusion

This project developed a robotic system to understand human instructions and perform long-horizon tasks, aiding individuals with limited mobility. It integrated perception, planning, control, and action modules for autonomous operation.

5.1 Accomplishments

The project achieved several key milestones:

- **Integrated System**: A multi-module robotic system (Perception, Planning, Control, Action with custom gripper) was successfully designed and developed, orchestrated by a Raspberry Pi 5 with ROS2.
- **Advanced AI**: State-of-the-art AI (VLMs like Qwen/Qwen2.5-VL for scene description/task planning, Grounded SAM for object localization) was implemented, with the Object Locator achieving 90% accuracy on a test dataset.
- **Custom Hardware**: Key custom hardware, including a PCB for FSR402 force sensor signal processing and a lead screw-based gripper, was designed and integrated.
- **Modular Verification**: Individual modules were verified, demonstrating accurate object identification (within 5 seconds), effective task decomposition, feasible plan generation, and successful inter-module communication.

5.2 Uncertainties and Future Developments

Despite successes, certain uncertainties and areas for future work remain. Fully achieving the 90% success rate for long-horizon tasks across diverse, dynamic environments requires further extensive testing, as detailed Action Module and integrated system performance metrics were not fully elaborated. Scaling to more complex tasks or nuanced human instructions, and managing the inherent limitations of VLMs, presents future challenges. Enhancing the naturalness and depth of human-robot interaction, particularly in handling ambiguity and implicit intent, requires further development.

5.3 Ethical Considerations

These are a couple of critical ethical considerations that go throughout our project:

5.3.1 IEEE and ACM Code of Ethics[18][19]

The project could be misused for unsafe or malicious tasks such as unauthorized modifications or weaponization. Responding to both the IEEE Code's concern [18] about physical abuse and ACM Code's[19] valuing on the public good, we strictly restrict our robot's capability to conduct physical harm through e.g. limiting the maximum operation speed or rejecting malicious language instructions. Another risk is that users might overestimate the robot's ability, leading to dangerous reliance on automation. We will provide clear user guidelines and training that ensure the maintenance of users' awareness of limitations of the robot, responding to the ACM Code's requirement to foster public awareness of our technology.

5.3.2 ISO/TS 15066[20]

As the project involves a robot interacting with human instructions, this standard provides guidelines on safe human-robot interaction, ensuring safe speeds, forces, and workspace conditions. We will also ensure that the instructions are given from a safe distance with respect to the robot's workspace.

References

[1] A. Radford, J. W. Kim, C. Hallacy, *et al.*, "Learning transferable visual models from natural language supervision," in *Proceedings of the 38th International Conference on Machine Learning (ICML)*, CLIP: a foundational vision–language model; accessed 05/18/2025, 2021, pp. 8748–8763. [Online]. Available: https://openai.com/research/clip.

[2] S. Bai, K. Chen, X. Liu, *et al.*, "Qwen2.5-vl technical report," *arXiv preprint arXiv:2502.13923*, 2025.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, and et al., "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020, GPT-3, a 175 B-parameter large language model; accessed 05/18/2025. [Online]. Available: https://arxiv.org/abs/2005.14165.

[4] T. Ren, S. Liu, A. Zeng, *et al.*, *Grounded sam: Assembling open-world models for diverse visual tasks*, 2024. arXiv: 2401.14159 [cs.CV].

[5] M. Contributors, *ModelScope: One-stop, open-source platform for foundation models*, version 1.12.0, Online; accessed 05/18/2025, 2022. [Online]. Available: https://modelscope.cn/.

[6] M. Caron, H. Touvron, I. Misra, *et al.*, "Emerging properties in self-supervised vision transformers," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, DINO: Self-Distillation with No Labels; accessed 18May2025, 2021, pp. 9650–9660. [Online]. Available: https://arxiv.org/abs/2104.14294.

[7] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, 2017, pp. 6000–6010. [Online]. Available: https://arxiv.org/abs/1706.03762.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788. [Online]. Available: https://arxiv.org/abs/1506.02640.

[9] S. Ramírez, *FastAPI: Fast and performant web framework for building APIs with Python 3.7+*, version 0.110.0, Online; accessed 05/18/2025, 2019. [Online]. Available: https://fastapi.tiangolo.com/.

[10] T. Christie and contributors, *Uvicorn: A lightning-fast asgi server implementation, using uvloop and httptools*, version 0.29.0, Online; accessed 05/18/2025, 2018. [Online]. Available: https://www.uvicorn.org/.

[11] ECE 470 Course Staff, University of Illinois Urbana-Champaign, *Lab 5: Robot Arm Motion Planning*, https://courses.engr.illinois.edu/ece470/sp2025/labs/lab5/, Course laboratory handout, accessed 05/18/2025, 2025.

[12] J. Denavit and R. S. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," *Journal of Applied Mechanics*, vol. 22, no. 2, pp. 215–221, 1955. [Online]. Available: https://doi.org/10.1115/1.4010995.

[13] Universal Robots A/S, *Dh parameters for calculations of kinematics and dynamics*, https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/, Online article, accessed 05/18/2025, n.d.

[14] Raspberry Pi Ltd., *Raspberry Pi 5 — technical specifications*, https://www.raspberrypi.com/products/raspberry-pi-5/, Datasheet and product page, accessed 05/18/2025, 2023.

[15] S. Macenski, G. Biggs, R. Lange, *et al.*, "ROS 2: Design, architecture, and uses," in *IEEE International Conference on Robots and Automation (ICRA) Workshop on Robots Operating in the Real World*, Accessed 05/18/2025, Philadelphia, PA, 2022. [Online]. Available: https://design.ros2.org.

[16] Universal Robots, *Universal robots ros2 driver*, https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver, Accessed 05/18/2025, 2025.

[17] Manufacturer Name, *Raspberry pi camera cable*, Online product page; accessed 18 May 2025, n.d. [Online]. Available: https://www.raspberrypi.com/products/camera-cable/.

[18] IEEE, *Ieee code of ethics*, https://www.ieee.org/about/corporate/governance/p7-8.html, Approved June 2024; accessed 05/18/2025, 2024.

[19] A. for Computing Machinery, *Acm code of ethics and professional conduct*, https://www.acm.org/code-of-ethics, Adopted 06/2018; accessed 05/18/2025, 2018.

[20] ISO. "Robots and robotic devices — collaborative robots." (2016), [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:ts:15066:ed-1:v1:en.

Appendix A ROS Custom Package Code

A.1 FSR Sensor Package

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Bool
from gpiozero import DigitalInputDevice

class FSRPublisher(Node):
    def __init__(self):
        super().__init__('fsr_publisher')

        # Declare a parameter 'gpio_pin' with default value 17
        self.declare_parameter('gpio_pin', 17)
        gpio_pin =
        ↪  self.get_parameter('gpio_pin').get_parameter_value()

        # Initialize GPIO pin 17 for FSR input with debounce
        self.fsr = DigitalInputDevice(gpio_pin, bounce_time=0.1)

        # Publisher for FSR status
        self.publisher_ = self.create_publisher(Bool,
        ↪  'fsr_pressed', 10)

        # Internal state to track current pressure status
        self.is_pressed = not self.fsr.value

        # Callbacks to update the internal state
        self.fsr.when_activated = self._on_released
        self.fsr.when_deactivated = self._on_pressed

        # Timer to publish the current status at 10 Hz
        self.timer = self.create_timer(0.1, self._publish_status)

        self.get_logger().info('FSR sensor node started.')

    def _on_pressed(self):
```

```
            self.is_pressed = True
            self.get_logger().info('Pressure detected (pressed)')

    def _on_released(self):
            self.is_pressed = False
            self.get_logger().info('Pressure released')

    def _publish_status(self):
            msg = Bool()
            msg.data = self.is_pressed
            self.publisher_.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = FSRPublisher()
    try:
            rclpy.spin(node)
    except KeyboardInterrupt:
            pass
    node.destroy_node()
    rclpy.shutdown()
```

A.2 Motor Control Package

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import Bool
from std_srvs.srv import Trigger, SetBool
from serial import Serial
from stepper.device import Device
from stepper.stepper_core.parameters import DeviceParams
from stepper.stepper_core.configs import Address
import re
import time

class GripperController(Node):
    def __init__(self):
```

```
14          super().__init__('gripper_controller')
15
16          # Define motor port and address, global bounds (omitted)
17          self.release_target_pos = 0
18
19          # Connect to motor
20          serial_conn = Serial(port, 115200, timeout=0.1)
21          self.device =
            ↪  Device(DeviceParams(serial_connection=serial_conn,
            ↪  address=Address(address)))
22          self.device.parse_cmd('ENA')
23
24          # Sensor state and parameters
25          self.fsr_triggered = False
26          self.is_grasping = False
27          self.is_releasing = False
28          self.is_out_of_bounds = False
29
30          # ROS interfaces
31          self.fsr_subscriber = self.create_subscription(Bool,
            ↪  '/fsr_pressed', self.fsr_callback, 10)
32          self.grasp_service = self.create_service(SetBool,
            ↪  '/grasp_object', self.handle_grasp)
33          self.release_service = self.create_service(SetBool,
            ↪  '/release_object', self.handle_release)
34          self.end_lock_service = self.create_service(Trigger,
            ↪  '/end_lock', self.handle_end_lock)
35          self.grasp_release_check_timer = self.create_timer(0.1,
            ↪  self.check_grasp_release_status)
36          self.get_logger().info('GripperController node has
            ↪  started.')
37
38      def fsr_callback(self, msg):
39          self.fsr_triggered = msg.data
40          # self.get_logger().info(f'FSR state updated:
            ↪  {self.fsr_triggered}') # Uncomment for debugging
41
```

```
    def handle_grasp(self, request, response):
        if self.is_out_of_bounds:
            if not request.data:
                response.success = False
                response.message = "Gripper in Emergency Stop. Call
                ↪  /grasp_object with data=true to force."
                return response
            else:
                self.device.parse_cmd('ENA')
        self.get_logger().info('Grasping object...')
        self.fsr_triggered = False
        self.is_grasping = True
        self.device.parse_cmd('CLR')
        self.device.parse_cmd('JOG 300')
        response.success = True
        response.message = 'Grasp command sent. Waiting for FSR
        ↪  trigger...'
        return response

    def handle_release(self, request, response):
        if self.is_out_of_bounds:
            if not request.data:
                response.success = False
                response.message = "Gripper in Emergency Stop. Call
                ↪  /release_object with data=true to force."
                return response
            else:
                self.device.parse_cmd('ENA')
        self.get_logger().info('Releasing object...')
        self.device.parse_cmd('ENA')
        self.device.parse_cmd('CLR') # Clear stall condition
        self.device.parse_cmd(f'MOV {self.release_target_pos}')
        self.is_releasing = True
        response.success = True
        response.message = 'Release command sent. Waiting for
        ↪  completion...'
        return response
```

```
75
76      def handle_end_lock(self, request, response):
77          self.get_logger().info('Removing out-of-bounds
            ↪  constraints...')
78          self.device.parse_cmd('ENA')
79          self.device.parse_cmd('CLR') # Clear stall condition
80          self.is_out_of_bounds = False
81          self.is_grasping = False
82          self.is_releasing = False
83          response.success = True
84          response.message = 'Successfully cleared out-of-bounds
            ↪  constraints.'
85          return response
86
87      def check_grasp_release_status(self):
88          current_pos = self.get_position()
89          if self.is_grasping:
90              if self.fsr_triggered:
91                  self.get_logger().info('Object detected by FSR |
                    ↪  stopping motor.')
92                  self.device.parse_cmd('STP')
93                  self.is_grasping = False
94              elif current_pos > self.max_pos:
95                  self.get_logger().warning(
96                      f'Position {current_pos} out of bounds
                        ↪  ({self.min_pos} - {self.max_pos}). Stopping
                        ↪  motor.'
97                  )
98                  self.device.parse_cmd('STP')
99                  self.device.parse_cmd('DIS')
100                 self.is_grasping = False
101                 self.is_out_of_bounds = True
102         elif self.is_releasing:
103             if current_pos < self.min_pos:
104                 self.get_logger().warning(
```

```
                    f'Position {current_pos} out of bounds ({self.min_pos} - {self.max_pos}). Stopping motor.'
                )
                self.device.parse_cmd('STP')
                self.device.parse_cmd('DIS')
                self.is_releasing = False
                self.is_out_of_bounds = True
            elif abs(current_pos - self.release_target_pos) < 250:
                self.get_logger().info('Gripper fully opened.')
                self.is_releasing = False

    def get_position(self) -> int:
        s = str(self.device.parse_cmd('POS'))
        try:
            m = re.search(r'POS\s*=\s*(-?\d+)', s)
        except Exception as e:
            self.get_logger().error(f"Error parsing position: {e}")
            raise ValueError("Could not parse position from device response.")
        if not m:
            raise ValueError("Could not parse position from device response.")
        return int(m.group(1))

    def _cleanup(self):
        if rclpy.ok():  # Only log if rclpy is still active
            self.get_logger().info('Shutting down ......')
        try:
            self.device.parse_cmd('DIS')
        except Exception:
            pass

    def destroy_node(self):
        self._cleanup()
        super().destroy_node()

```

```
138
139 def main(args=None):
140     rclpy.init(args=args)
141     node = GripperController()
142     try:
143         rclpy.spin(node)
144     except KeyboardInterrupt:
145         pass
146     finally:
147         node.destroy_node()
```