

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 29, 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
1.1	Problem Statement	1
1.2	High-Level Requirement List	2
2	Design	3
2.1	Design Procedure (Alternatives)	3
2.2	Perception Module	4
2.2.1	Object identifier	4
2.2.2	High-level Perception: Scene description	4
2.2.3	Low-level Perception: Fine-grained object location and orientation	4
2.3	Planning Module	6
2.3.1	High-level Planning: Structured Instructions	6
2.3.2	Low-level planning: Inverse Kinematics	7
2.4	Control Module	8
2.4.1	Peripheral Connection - Force Sensor	9
2.4.2	Peripheral Connection - Motor	9
2.4.3	Device Connection - Camera, UR3e and Server	10
2.4.4	Module Overall Workflow	11
2.5	Action Module	11
2.5.1	Design Procedure	11
2.5.2	Design Details	14
2.5.3	Workflow	15
3	Verification	16
3.1	Verification of PCB Circuit	16
3.1.1	Comparator Threshold Accuracy	16
3.1.2	LED Indicator Performance	16
3.2	Verification of Perception Module	17
3.2.1	RGB Camera	17
3.2.2	Force Sensor	17
3.2.3	Object Locator	17
3.3	Verification of Planning Module	18
3.3.1	Task Decomposition & Structured Plan Generation	18
3.3.2	Motion Planning	19
3.4	Verification of Action Module	19

3.4.1	Grasping Size Range	19
3.4.2	Grasping Mass Limit	20
3.4.3	Stop/Safe Operation	20
3.4.4	Closed-loop Control Response	20
4	Cost and Schedule	21
4.1	Cost Analysis	21
4.2	Schedule	22
5	Conclusion	24
5.1	Accomplishments	24
5.2	Uncertainties	24
5.3	Future Work	25
5.4	Ethical Considerations	25
5.4.1	IEEE and ACM Code of Ethics[23][24]	25
5.4.2	ISO/TS 15066[25]	25
	References	26
	Appendix A PCB Design Details	29
	Appendix B ROS Custom Package Code	31
B.1	FSR Sensor Package	31
B.2	Motor Control Package	32
	Appendix C Verifications for Planning Module	38

1 Introduction

1.1 Problem Statement

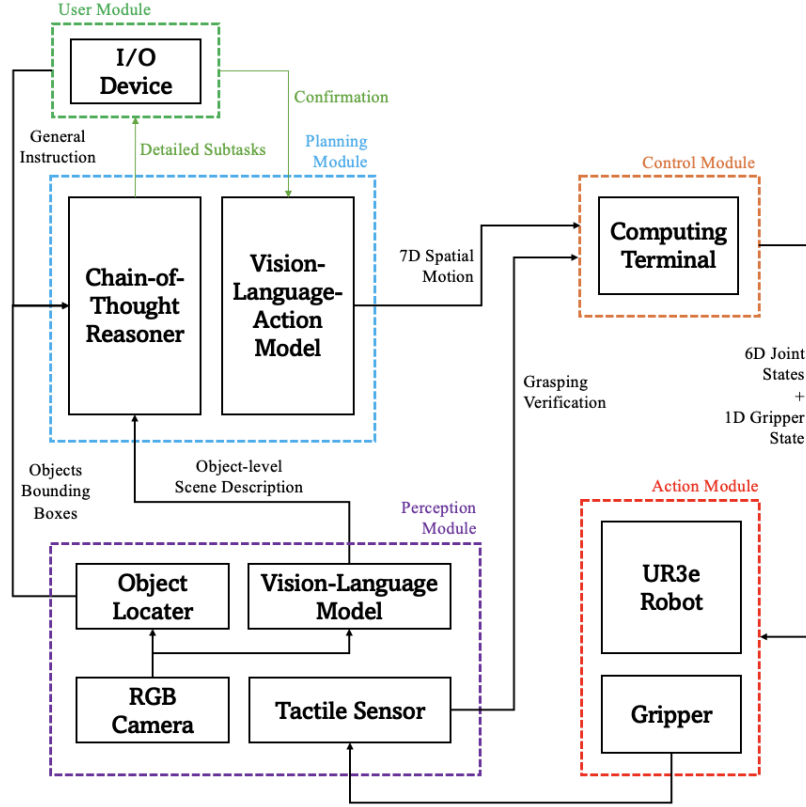


Figure 1: The overall block diagram.

The application of robotic arms to assist with daily tasks is becoming increasingly important, particularly for individuals with limited mobility, such as the elderly, children, and people with disabilities. Many basic 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 demonstrate robust, consistent performance across multiple steps.

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

back. 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 Design Procedure (Alternatives)

One of the most challenging designs we’ve faced in this project is to generate effective plans at various levels of abstraction. Initially, for this project, considering the hierarchical nature of the planning process, we referred to these compositional foundation models[1] which generate high-level plans with Large Language Model, conduct motion planning with video diffusion, and calculate the inverse kinematics with Vision Transformer in an end-to-end manner. However, considering the consumption on fine-tuning the diffusion model based on our settings, as well as the risk of failure, we turned to a more easygoing approach.

As an alternative to planning with hierarchy and finer granularity, the Vision-Language-Action (VLA) model integrates different levels of planning in an end-to-end manner. This is a training model capable of generating action directly from a given scene and instruction, showing considerable performance in work [2]. We have also referred to this enhanced model [3] with stronger performance. However, due to the moderately high training cost, data demands, and overall room to improve in accuracy to accomplish complicated tasks, we eventually gave up this approach.

For the components in our specific visual pipeline, instead of mounting an RGB camera on the gripper, an RGB-D camera would be another promising choice, and we could have mounted the camera to a fixed platform, both of which were expected to lead to more accurate object locations. However, the fixed location of the camera also limits the robot arm’s ability to operate in a wider range, which conflicts with our goal to enable the robot to perform long-horizon tasks in a general manner. Meanwhile, RGBD cameras are normally in larger volumes and at much higher cost. Furthermore, after all, we as human beings do not rely on another eye that emits lasers to detect distance when we grasp things. Therefore, we eventually have the RGB camera on the gripper to infer the locations of objects in a more general and challenging manner.

The design of our action module has also undergone several iterations, which will be elaborated in detail in **Section 2.5**.

2.2 Perception Module

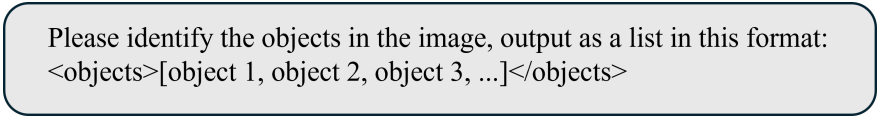
2.2.1 Object identifier

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[4], [5], which extends the emerged power of large language models[6] to the modalities of both text and image, for scene description, and a so-called vision foundation model, Grounded SAM[7] for object locating.

2.2.2 High-level Perception: Scene description

Given a scene image as input, a large Vision Language Model (VLM)[5] is prompted through ModelScope[8] API to describe the scene and identify objects in the scene.

- Base Model: Qwen/Qwen2.5-VL-7B-Instruct[5] from ModelScope[8].
- 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.

2.2.3 Low-level Perception: Fine-grained object location and orientation

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[9], which utilizes the Transformer[10] backbone and was trained in a self-supervised manner. Specifically, we deploy the *Grounded Segment-Anything (Grounded SAM)*[7] on our server that listens for a list of object names and an image sent from our control module in binary



Figure 3: Input labels: person, plant, bin

form, and returns the masks and boxes for each identified object. Compared with traditional vision models such as *You Only Look Once (YOLO)*[11], 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*[12] and *Uvicorn*[13], which will be introduced in detail in **Section 2.4**.

After having the contours marked (red in Figure 3), we first calculate the geometric center of the target pixel group with the built-in function of OpenCV[14], `cv2.moments()`. Then, image coordinates of the center is mapped to the world frame by:

$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_W & Y_W & Z_W & 1 \end{bmatrix}^\top$$

in which Z_c is the real-time distance from the camera to the surface of table. R and T are transformation matrices from the world frame to the camera frame, which are determined by joint states and the forward kinematics process implemented with screw axes[15]. X_W, Y_W, Z_W are coordinates of centers in the world frame, which are then sent to the low-level planning module.

In addition to the world locations of targets, orientations of targets are also required for reasonable grasps. For that, an ellipse fitting is applied to the marked contour. Representing an ellipse as $F(x, y) = ax^2 + bxy + cy^2 + dx + ey + f = 0$, the following optimization is performed:

$$\min \sum_{i=1}^N F(x_i, y_i)^2$$

in order to derive the ellipse parameters a to f . This is also assisted with an OpenCV built-in function, `cv2.fitEllipse()`. Shifted angle of the ellipse is considered as the orientation angle of the target and is set as the state of the last joint, θ_6 . Derivation of the rest angles will be demonstrated in the next section.

2.3 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 4: Planning VLM prompt.

2.3.1 High-level Planning: Structured Instructions

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[8].
- 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 4. Where the Task is user input, the action list is a combination between "Grasp" and objects as well as "Move to" and given locations.

2.3.2 Low-level planning: Inverse Kinematics

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 ECE470 lab5[16], we have derived a close-form solution with Denavit–Hartenberg parameters[17] of UR3e[18] as follows:

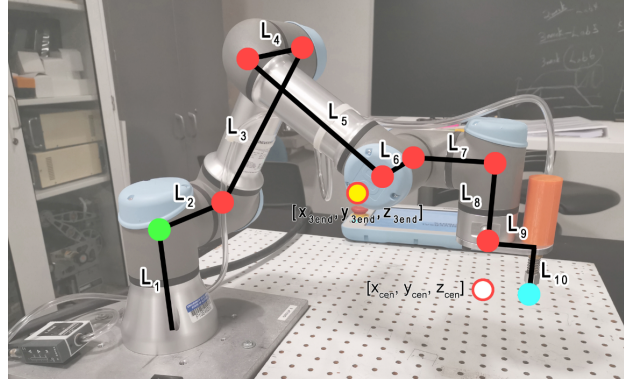


Figure 5: Inverse-kinematics solution illustration for the UR3e robot arm.[16]

- θ_1 : Base rotation angle

Denoting the length $l_2 - l_4 + l_6$ as l_p , the perpendicular offset from P_{cen} to P_{3end} , and then we have the following equations:

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

- x_{3end} , y_{3end} , and z_{3end} :

$$x_{3end} = x_{cen} + l_p \sin(\theta_1) - l_7 \cos(\theta_1)$$

$$y_{3end} = y_{cen} - l_p \cos(\theta_1) - l_7 \sin(\theta_1)$$

$$z_{3end} = z_{cen} + l_8 + l_{10}$$

- θ_2 and θ_3 :

$$l_3 \sin(-\theta_2) + l_1 - l_5 \sin(\theta_3 + \theta_2) = z_{3end}$$

$$l_3 \cos(\theta_2) + l_5 \cos(\theta_3 + \theta_2) = \sqrt{x_{3end}^2 + y_{3end}^2}$$

- θ_4 : Maintains horizontal end effector

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

- θ_5 : Keeps end effector vertical

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

2.4 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 [19] (Figure 6), which runs Ubuntu 24.04 LTS and the ROS2 (Robot Operating System 2) [20], 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.



Figure 6: Raspberry Pi 5



Figure 7: FSR Sensor Installation on Gripper

2.4.1 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 7) 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 signal and provides visual feedback via LEDs. The details of the PCB design are included in Appendix A.

Sensor Signal Processing The digital signal is transmitted to the Raspberry Pi 5's GPIO port. A custom ROS2 package processes this signal with a 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 B.

2.4.2 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.

2.4.3 Device Connection - Camera, UR3e and Server

Camera Setup We also used a 4K Pi camera with field of view $H108^\circ \times V92^\circ$ 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 [21]. 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[12], 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[13], 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.

2.4.4 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 8 illustrates the overall workflow of the system.

2.5 Action Module

2.5.1 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.

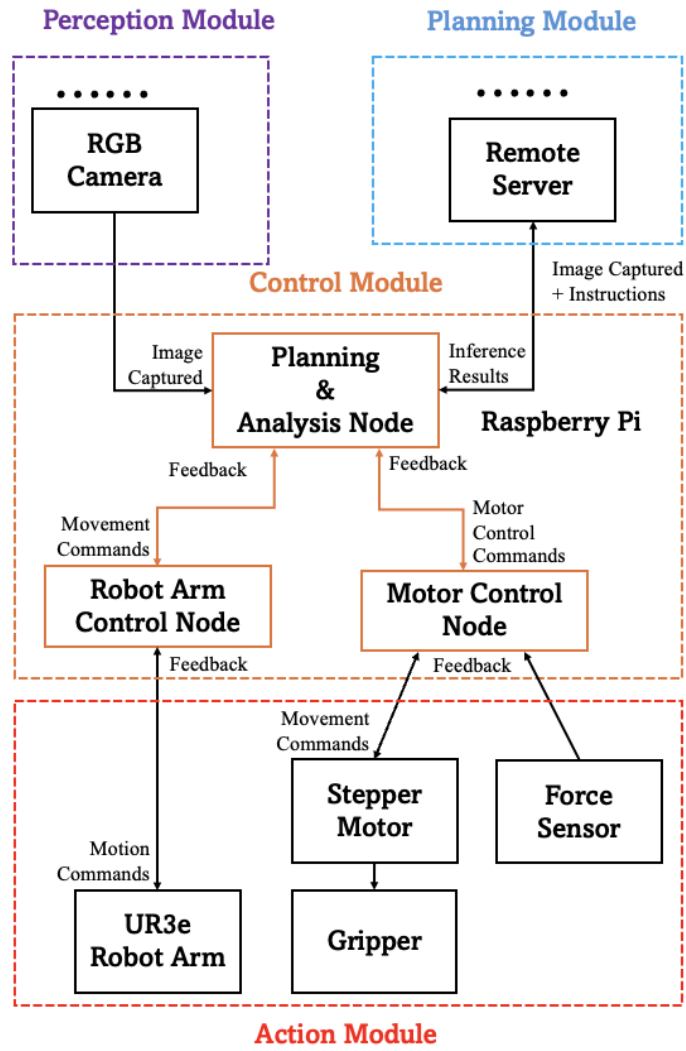


Figure 8: Control & Action Module Workflow

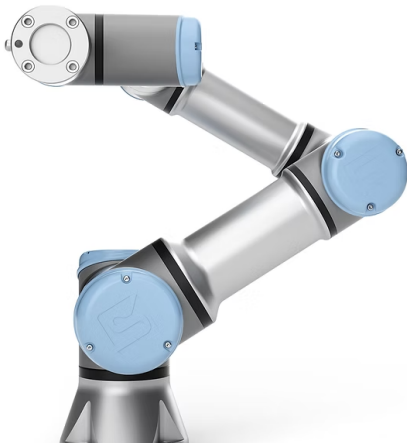


Figure 9: UR3e

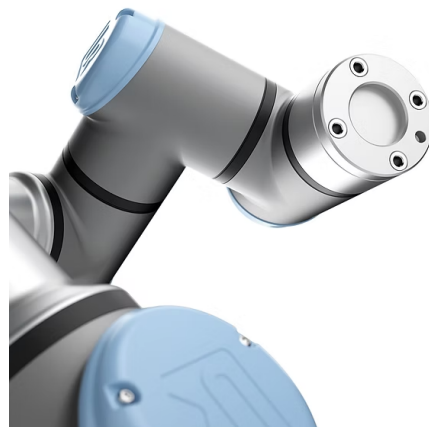


Figure 10: UR3e End Effector

Initially, we proposed a mechanism depicted in Figure 11. 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.

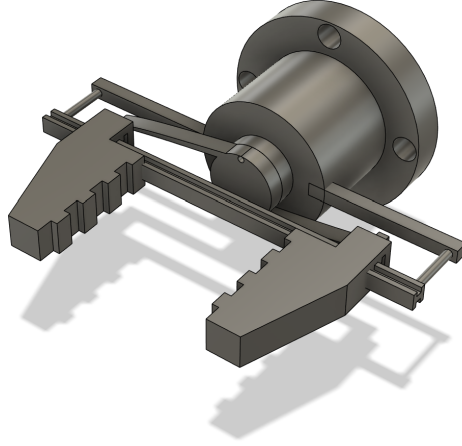


Figure 11: Gripper Version 1

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 12.

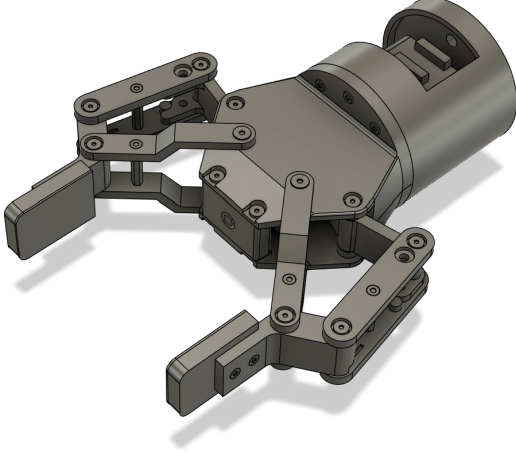


Figure 12: Gripper Version 2

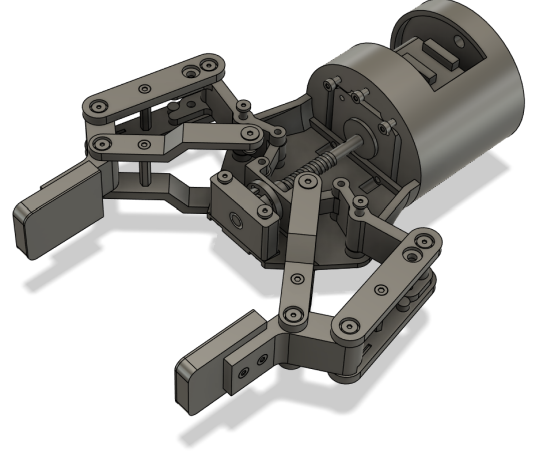


Figure 13: Gripper Internal View

2.5.2 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 Nm and lead screw pitch (lead) of 2 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 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 mecha-

nism. Therefore, we conservatively set the maximum grasping weight to 0.25 kg for safe and reliable operation.

2.5.3 Workflow

The workflow of the Action Module is shown in Figure 8 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

This test evaluates the LM393 comparator's ability to be tuned to specific grasp forces and its stability.

Procedure: The FSR402 sensor was placed on the input node of the comparator. The potentiometer R2 was varied across its full range. We observed and recorded the voltage shift in the toggle point for a defined change in R2. R2 was cycled back and forth over its range multiple times, and the threshold voltage was measured to assess repeatability. Known weights (500 g and 1 kg) were applied to the FSR402 sensor, and the corresponding R2 value at which the comparator toggled was recorded.

Quantitative Results: A $1\text{ k}\Omega$ change in R2 produced an average 0.32 V shift in the toggle point, demonstrating the system's resolution. The sensor-to-threshold correlation showed that when a 500 g weight was applied to the FSR, the comparator toggled at $R2 \approx 4.5\text{ k}\Omega$. Increasing the weight to 1 kg required R2 to be approximately $3.0\text{ k}\Omega$, confirming the expected inverse relationship. 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

This test verifies the correct operation and visibility of the LED indicators.

Procedure: VCC and GND pins were connected to the PCB. The status of LED1 was observed. The input to the comparator was manipulated to achieve an output "1", and the status of LED2 was observed. Visibility of both LEDs under normal lab lighting conditions was assessed.

Quantitative Results: LED1 turns on as long as the VCC and GND pins are correctly connected. LED2 turns on when the comparator output is "1". Both LEDs were observed to switch cleanly with the comparator output and were 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

This device is evaluated based on communication with the Control Module.

Procedure: Connect the RGB camera to the Raspberry Pi 5 via a CSI cable[22], and time the process of displaying an image of the scene on the computer screen.

Quantitative Results: The whole process is completed in less than 0.1 seconds.

3.2.2 Force Sensor

This device is evaluated based on communication with the Control Module.

Procedure: Put the FSR Sensor on the gripper, connect it to the Raspberry Pi 5, check the signal on PCB when pressing the sensor, verify that the gripper will stop properly when gripping an object.

Quantitative Results: When pressing the sensor, the red signal on the PCB is activated stably. When the gripper is being actuated to grip an object, it will stop to avoid danger with 100% success.

3.2.3 Object Locator

Our Object Locator is evaluated on a common-objects test dataset.

Procedure: Collect open source common object images, label the objects in the image manually, record the number of objects in each category, run and time the Object Locator model on all the images, calculate the accuracy based on its match with labels.

Quantitative Result: Figure 14 and Figure 18 15 show the effectiveness of this model, reaching an accuracy of 90% on the identification of 860 objects of 24 common classes. The detection process for a single image with resolution 640×480 is tested to be within 0.5 seconds on RTX 3090.

Uncertainty: When applying to real scenarios, some objects may be misclassified due to the limitation of the pretraining model. For example, during the final demo, the box of poker cards is classified as a cigarette pack, which is not correct according to user's instruction. However, such uncertainty is trivial since the Object Locator is guided by Scene Descriptor, between which the name of the object is guaranteed to align. In addition, such

uncertainty can be further relieved by adding features to the ambiguous objects (putting an ace on the top of a poker card box).

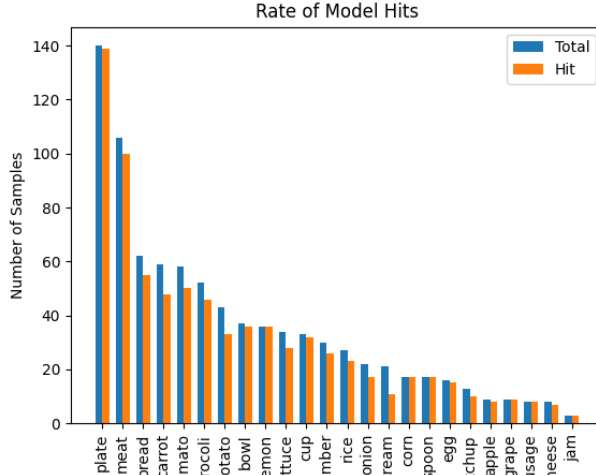


Figure 14: Grouding DINO quantitative evaluation result

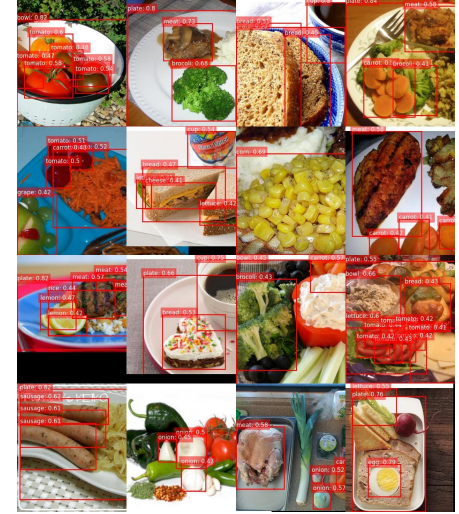


Figure 15: Grouding DINO annotation examples

3.3 Verification of Planning Module

3.3.1 Task Decomposition & Structured Plan Generation

Task decomposition and structured plan generation is evaluated with case studies.

Procedure: Collect images of real scenes containing different objects from our laboratory, input different user instructions, ask our Planner to generate subtasks and structured plans, analyze the correctness and feasibility of these plans.

Quantitative Results: The case study (Figure 19, Appendix C) shows its effectiveness in decomposing high-level tasks into structured plans. The other 20 test scenes contain combinations of objects, such as a sponge, a poker card box, a marker pen, a bottle, a box, and a tissue pack. 14 of them are generated correctly (70% accuracy), which is valid.

Uncertainty: All remaining invalid cases have the wrong target location to move to. Such uncertainty is relieved by asking the user to check whether the generated plan satisfies. If not, the planner will generate again. From our experiment, two-round generation results in over 90% valid plans.

3.3.2 Motion Planning

Procedure: In order to verify our implemented kinematics functions, for each sample target location in Cartesian coordinates, we first map the location into the six-dimensional joint space with inverse kinematics. Then, the joint states are mapped back to the Cartesian space with forward kinematics and compared with the original input. If the input remains the same after being mapped backward and forward, combining with field tests to be verified by intuition, both forward and inverse kinematics are verified.[16]

Quantitative Results: We tested on three sample targets: $(0.257, -0.350, 0.067)$, $(0.400, -0.128, 0.152)$, and $(0.136, -0.274, 0.021)$ and had corresponding RMS errors as:

$$\epsilon_1 = 2.831 \times 10^{-16}, \epsilon_2 = 2.096 \times 10^{-16}, \epsilon_3 = 6.497 \times 10^{-16}$$

Uncertainty: For angle of the last joint which adapts the target’s orientation, the regularization mechanism we implemented restricted the absolute gripper orientation in the world frame. However, the base angle varies, and the relative gripper orientation is determined by both the base angle and the absolute gripper orientation. Due to the limited length of the csi cable that connects to the camera, a more complicated regularization is in need to restrict both absolute and relative gripper orientation.

3.4 Verification of Action Module

3.4.1 Grasping Size Range

Procedure: The gripper’s operational size range is evaluated by testing its ability to grasp objects of varying widths: 1 cm, 3 cm, and 5 cm. For each object size, the gripper performs a series of 10 grasp grasp-and-lift trials. It’s ability to securely hold each object without slipping or dropping was observed and recored.

Quantitative Results: The gripper achieved a 90% success rate in grasping and holding 1 cm, 3 cm, and 5 cm width objects. Throughout all tests, no objects are dropped.

Uncertainty: Variability in the shape and texture of the object can affect the stability of the grasp. Irregular objects such as umbrellas may fail to trigger the force sensor, causing the gripper to be unable to stop grasping. To reduce uncertainty, larger force sensors or flexible materials will be used on the gripper.

3.4.2 Grasping Mass Limit

Procedure: The gripper's load capacity is evaluated by testing its ability to securely lift and transport objects weighing 0.1 kg, 0.2 kg, and 0.25 kg. For each weight, the gripper performed 10 lift-and-hold trials. The ability to maintain a secure grip without slippage or failure was observed and recorded.

Quantitative Results: The gripper achieved a 100% success rate in securely lifting and holding objects up to 0.25 kg. No slippage or dropping occurred in any of the trials.

Uncertainty: Some objects, such as small packs of tissue paper, may undergo significant shape deformation when grasped. In these cases, the sensitivity of the force sensor needs to be adjusted to properly control the grasping force and ensure an appropriate grip without damaging the object.

3.4.3 Stop/Safe Operation

Procedure: Test the emergency stop function using the robot control panel during grasping and releasing operations. While the gripper is performing tasks, press the emergency stop button to immediately halt robot motion. Observe the gripper's ability to maintain a secure hold or safely cease operation. Repeat this process 10 times for each operation.

Quantitative Results: In all 20 trials (10 grasping, 10 releasing), pressing the emergency stop button halts all motion immediately. The gripper maintains a secure hold on the object during grasp stops and safely ceases operation during release stops. No object drops or mechanical faults occurs, demonstrating reliable emergency stop and safe handling capability.

3.4.4 Closed-loop Control Response

Procedure: Test the closed-loop control by applying pressure to the force sensor both manually and during object grasping. When pressing the force sensor or when the gripper grasps an object, verify that the stepper motor stops automatically to prevent further motion. Repeat this test 10 times for each scenario.

Quantitative Results: In all 20 trials (10 manual presses, 10 object grasps), the stepper motor stops immediately after the force sensor is triggered. The system consistently prevents over-compression, ensuring protection for both the gripper and the object without failure.

4 Cost and Schedule

4.1 Cost Analysis

Cost Item	Unit Cost (USD)	Quantity	Total Cost (USD)
<i>Hardware Components</i>			
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
<i>Labor</i>			
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 Re-view: Guiding Long-Planning with VLM	Literature Review: Hierarchical Planning Foundation Model	Literature Re-view: Optimal force sensor placement for gripper design	Investigate common gripper design for robot arm
3/24	Literature Re-view: VLA models, including OpenVLA	Literature Re-view: 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 with Advanced AI:** 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. State-of-the-art AI models (Qwen2.5-VL for planning, Grounded SAM for perception) was integrated.
- **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.

As a whole, the system accomplishes the goal to perform long-horizon tasks given simple human instructions. During our [final demo](#), the instruction "Put everything into the box" was given to the system. First, the robot arm rotated around to capture images of two scenes to perform object identification and detection. Then, the planner reasoned about the scene and the instructions to generate a plan and asked for user's approval. After approved, each step of the plan was further planned into joint trajectories and executed by the robot arm, until the tasks were done. While planning, it captured images and dynamically adjusted its target location.

5.2 Uncertainties

While our system demonstrates promising results, several uncertainties remain:

First, the system's robustness in unfamiliar or cluttered environments is not fully validated. Our current tests were conducted in relatively controlled setups, and performance may degrade when encountering occlusions, poor lighting, or novel object types not present in the training data.

Second, while the object detection module achieves high accuracy on test datasets, real-world variability (e.g., reflective surfaces, overlapping items) occasionally causes false detections or missed objects, which could disrupt the task flow.

Third, the Action Module assumes reliable gripper operation and precise motor control. However, occasional mechanical slips or delays in ROS2 message passing were observed during testing, which could affect execution consistency in longer tasks.

More details of uncertainties in individual modules are covered in the Design section.

5.3 Future Work

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.

5.4 Ethical Considerations

5.4.1 IEEE and ACM Code of Ethics[23][24]

The project could be misused for unsafe or malicious tasks, such as unauthorized modifications or weaponization. Responding to both the IEEE Code’s concern [23] about physical abuse and ACM Code’s[24] 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 that ensure that users are aware of the robot’s limitations, responding to the ACM Code’s requirement to foster public awareness of our technology.

5.4.2 ISO/TS 15066[25]

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. Ajay, S. Han, Y. Du, *et al.*, “Compositional foundation models for hierarchical planning,” *arXiv preprint arXiv:2309.08587*, 2023.
- [2] M. Kim, K. Pertsch, S. Karamcheti, *et al.*, “Openvla: An open-source vision-language-action model,” *arXiv preprint arXiv:2406.09246*, 2024.
- [3] M. Zawalski, W. Chen, K. Pertsch, O. Mees, C. Finn, and S. Levine, “Robotic control via embodied chain-of-thought reasoning,” *arXiv preprint arXiv:2407.08693*, 2024.
- [4] 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>.
- [5] S. Bai, K. Chen, X. Liu, *et al.*, “Qwen2.5-vl technical report,” *arXiv:2502.13923*, 2025.
- [6] 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>.
- [7] T. Ren, S. Liu, A. Zeng, *et al.*, *Grounded sam: Assembling open-world models for diverse visual tasks*, 2024. arXiv: 2401.14159 [cs.CV].
- [8] ModelScope, *ModelScope: One-stop, open-source platform for foundation models*, version 1.12.0, accessed 05/18/2025, 2022. [Online]. Available: <https://modelscope.cn/>.
- [9] 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>.
- [10] 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>.
- [11] 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>.

- [12] 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/>.
- [13] T. C. et al., *Uvicorn: A lightning-fast asgi server implementation, using uvloop and http-tools*, version 0.29.0, accessed 05/18/2025, 2018. [Online]. Available: <https://www.uvicorn.org/>.
- [14] G. Bradski, "The opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.
- [15] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. [Online]. Available: http://hades.mech.northwestern.edu/index.php/Modern_Robotics.
- [16] 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.
- [17] 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>.
- [18] 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.
- [19] 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.
- [20] 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>.
- [21] Universal Robots, *Universal robots ros2 driver*, https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver, Accessed 05/18/2025, 2025.
- [22] Manufacturer Name, *Raspberry pi camera cable*, Online product page; accessed 18 May 2025, n.d. [Online]. Available: <https://www.raspberrypi.com/products/camera-cable/>.
- [23] IEEE, *Ieee code of ethics*, <https://www.ieee.org/about/corporate/governance/p7-8.html>, Approved June 2024; accessed 05/18/2025, 2024.
- [24] 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.

- [25] 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 PCB Design Details

The PCB design is crucial for ensuring the optimal performance of the Gripper Controller system. Careful consideration was applied to the layout, component selection, and thermal management to minimize noise and enhance reliability. Figure 16 and Figure 17 show the circuit schematics and the PCB layout. The 3D view of the PCB board is shown in Figure 18. The key components of the PCB are:

- **FSR Sensor Input:** The FSR402 sensor is wired as a pull-up voltage divider with R1 ($10\text{ k}\Omega$). 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 ($10\text{ k}\Omega$).
 - 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 ($1\text{ k}\Omega$) illuminates when the circuit is working properly.
 - LED2 (red) with R4 ($1\text{ k}\Omega$) 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.

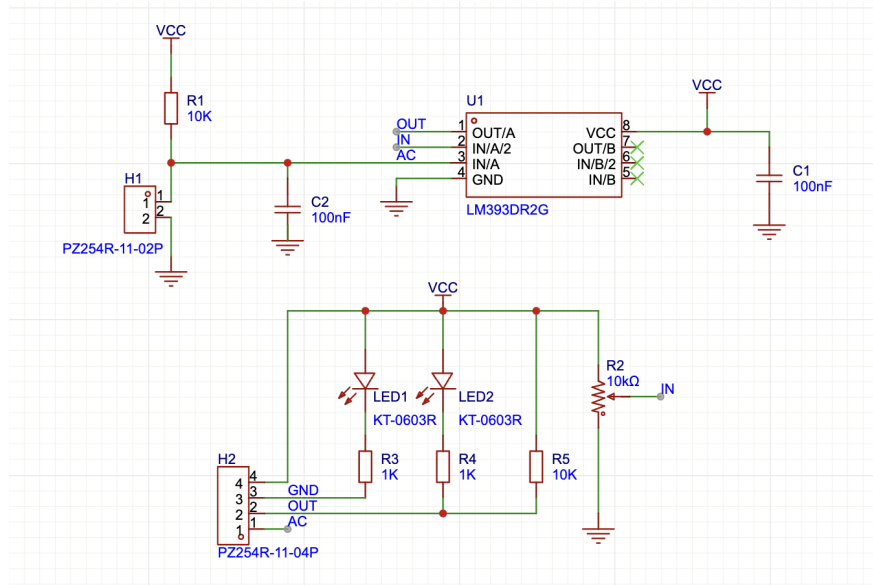


Figure 16: PCB Board Circuit Schematics

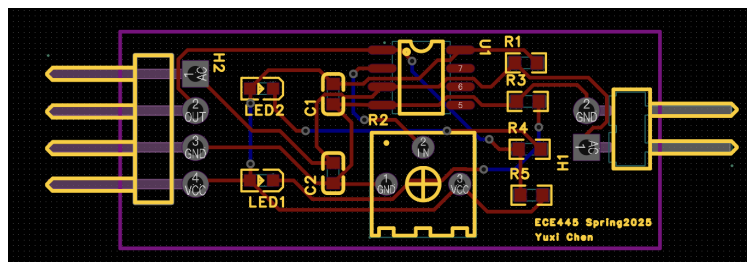


Figure 17: PCB Board Layout

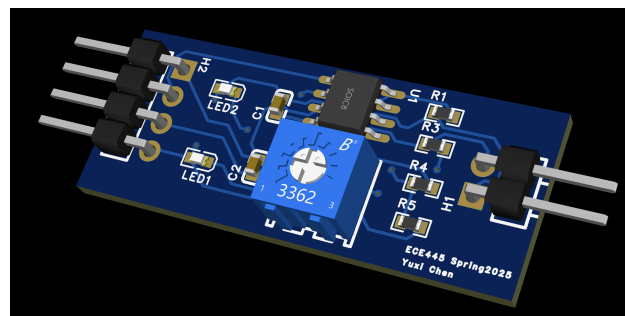


Figure 18: PCB Board 3D View

Appendix B ROS Custom Package Code

B.1 FSR Sensor Package

```
1 import rclpy
2 from rclpy.node import Node
3 from std_msgs.msg import Bool
4 from gpiozero import DigitalInputDevice
5
6 class FSRPublisher(Node):
7     def __init__(self):
8         super().__init__('fsr_publisher')
9
10        # Declare a parameter 'gpio_pin' with default value 17
11        self.declare_parameter('gpio_pin', 17)
12        gpio_pin =
13        ↪ self.get_parameter('gpio_pin').get_parameter_value()
14
15        # Initialize GPIO pin 17 for FSR input with debounce
16        self.fsr = DigitalInputDevice(gpio_pin, bounce_time=0.1)
17
18        # Publisher for FSR status
19        self.publisher_ = self.create_publisher(Bool,
20        ↪ 'fsr_pressed', 10)
21
22        # Internal state to track current pressure status
23        self.is_pressed = not self.fsr.value
24
25        # Callbacks to update the internal state
26        self.fsr.when_activated = self._on_released
27        self.fsr.when_deactivated = self._on_pressed
28
29        # Timer to publish the current status at 10 Hz
30        self.timer = self.create_timer(0.1, self._publish_status)
31
32        self.get_logger().info('FSR sensor node started.')
33
34    def _on_pressed(self):
```

```

33         self.is_pressed = True
34         self.get_logger().info('Pressure detected (pressed)')
35
36     def _on_released(self):
37         self.is_pressed = False
38         self.get_logger().info('Pressure released')
39
40     def _publish_status(self):
41         msg = Bool()
42         msg.data = self.is_pressed
43         self.publisher_.publish(msg)
44
45 def main(args=None):
46     rclpy.init(args=args)
47     node = FSRPublisher()
48     try:
49         rclpy.spin(node)
50     except KeyboardInterrupt:
51         pass
52     node.destroy_node()
53     rclpy.shutdown()

```

B.2 Motor Control Package

```

1  import rclpy
2  from rclpy.node import Node
3  from std_msgs.msg import Bool
4  from std_srvs.srv import Trigger, SetBool
5  from serial import Serial
6  from stepper.device import Device
7  from stepper.stepper_core.parameters import DeviceParams
8  from stepper.stepper_core.configs import Address
9  import re
10 import time
11
12 class GripperController(Node):
13     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 =
22         ↪ Device(DeviceParams(serial_connection=serial_conn,
23         ↪ address=Address(address)))
24     self.device.parse_cmd('ENA')
25
26     # Sensor state and parameters
27     self.fsr_triggered = False
28     self.is_grasping = False
29     self.is_releasing = False
30     self.is_out_of_bounds = False
31
32     # ROS interfaces
33     self.fsr_subscriber = self.create_subscription(Bool,
34         ↪ '/fsr_pressed', self.fsr_callback, 10)
35     self.grasp_service = self.create_service(SetBool,
36         ↪ '/grasp_object', self.handle_grasp)
37     self.release_service = self.create_service(SetBool,
38         ↪ '/release_object', self.handle_release)
39     self.end_lock_service = self.create_service(Trigger,
40         ↪ '/end_lock', self.handle_end_lock)
41     self.grasp_release_check_timer = self.create_timer(0.1,
42         ↪ self.check_grasp_release_status)
43     self.get_logger().info('GripperController node has
44         ↪ started.')
45
46     def fsr_callback(self, msg):
47         self.fsr_triggered = msg.data
48         # self.get_logger().info(f'FSR state updated:
49         ↪ {self.fsr_triggered}') # Uncomment for debugging

```

```

42 def handle_grasp(self, request, response):
43     if self.is_out_of_bounds:
44         if not request.data:
45             response.success = False
46             response.message = "Gripper in Emergency Stop. Call
47                 ↳ /grasp_object with data=true to force."
48             return response
49         else:
50             self.device.parse_cmd('ENA')
51             self.get_logger().info('Grasping object...')
52             self.fsr_triggered = False
53             self.is_grasping = True
54             self.device.parse_cmd('CLR')
55             self.device.parse_cmd('JOG 300')
56             response.success = True
57             response.message = 'Grasp command sent. Waiting for FSR
58                 ↳ trigger...'
59             return response
60
61 def handle_release(self, request, response):
62     if self.is_out_of_bounds:
63         if not request.data:
64             response.success = False
65             response.message = "Gripper in Emergency Stop. Call
66                 ↳ /release_object with data=true to force."
67             return response
68         else:
69             self.device.parse_cmd('ENA')
70             self.get_logger().info('Releasing object...')
71             self.device.parse_cmd('ENA')
72             self.device.parse_cmd('CLR') # Clear stall condition
73             self.device.parse_cmd(f'MOV {self.release_target_pos}')
74             self.is_releasing = True
75             response.success = True
76             response.message = 'Release command sent. Waiting for
77                 ↳ completion...'
78             return response

```

```

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

```

```

105         f'Position {current_pos} out of bounds
           ↳ ({self.min_pos} - {self.max_pos}). Stopping
           ↳ motor.'
106     )
107     self.device.parse_cmd('STP')
108     self.device.parse_cmd('DIS')
109     self.is_releasing = False
110     self.is_out_of_bounds = True
111     elif abs(current_pos - self.release_target_pos) < 250:
112         self.get_logger().info('Gripper fully opened.')
113         self.is_releasing = False
114
115     def get_position(self) -> int:
116         s = str(self.device.parse_cmd('POS'))
117         try:
118             m = re.search(r'POS\s*=\s*(-?\d+)', s)
119         except Exception as e:
120             self.get_logger().error(f"Error parsing position: {e}")
121             raise ValueError("Could not parse position from device
           ↳ response.")
122         if not m:
123             raise ValueError("Could not parse position from device
           ↳ response.")
124         return int(m.group(1))
125
126     def _cleanup(self):
127         if rclpy.ok(): # Only log if rclpy is still active
128             self.get_logger().info('Shutting down .....')
129         try:
130             self.device.parse_cmd('DIS')
131         except Exception:
132             pass
133
134     def destroy_node(self):
135         self._cleanup()
136         super().destroy_node()
137

```



```
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()
```

Appendix C Verifications for Planning Module

The images and code snippet illustrate a robotics system's planning capabilities. We see a real-world scenario where a robot needs to understand a user's request, break it down into actionable steps (task decomposition), generate a sequence of actions (plan generation), and then potentially execute those actions in a simulated environment. The example shows explicitly a robot arm tasked with delivering a Coke and then two fruits to a user, highlighting how abstract commands are translated into concrete robotic movements like 'grasp' and 'move to user'. This process enables robots to perform complex tasks in dynamic environments.



Figure 19: A Planning Module Case Study

```
1 <User Instruction> Give me two fruits after giving me a coke.
2 <Output> To accomplish the task of giving two fruits after
   ↳ giving a coke, we need to follow the sequence of actions
   ↳ carefully.
3 Here's the step-by-step decomposition of the task and the
   ↳ corresponding plan:
4 ### Task Decomposition:
5 1. **Pick up the coke**: The user needs a coke first.
6 2. **Move the coke to the user**: Deliver the coke to the
7   user.
8 3. **Pick up the banana**: After delivering the coke, pick up
   ↳ one fruit (banana).
9 4. **Move the banana to the user**: Deliver the banana to
10 the user.
```

```
11 5. **Pick up the apple**: Pick up the second fruit (apple).
12 6. **Move the apple to the user**: Deliver the apple to the
    ↪ user.
13 ### Plan Generation:
14 Based on the action list and the guidelines, the plan will be:
15 ```xml
16 <plan>['grasp coke', 'move to user', 'grasp banana', 'move
17 to user', 'grasp apple', 'move to user']</plan>
18 ```
19 ### Explanation of Each Step:
20 1. **'grasp coke'**: Pick up the coke can from the table.
21 2. **'move to user'**: Deliver the coke to the user.
22 3. **'grasp banana'**: Pick up the banana from the table.
23 4. **'move to user'**: Deliver the banana to the user.
24 5. **'grasp apple'**: Pick up the apple from the table.
25 6. **'move to user'**: Deliver the apple to the user. This plan
    ↪ ensures that the task is completed in the correct order,
    ↪ adhering to
26 the constraints and guidelines provided.
```