From Language to Action: A Vision-Language and Kinematics-Guided Framework for 6-DOF Robotic Arm Manipulation

ESE-498 Capstone Design Project Final Report

Submitted to the Department of Electrical and Systems Engineering

May 6, 2025

Client:
Electrical & Systems Engineering Department
Professor Dennis Mell

Course Advisor:
Professor Jason Trobaugh[1] - jasont@wustl.edu

Design Advisor:
Professor Dennis Mell[1] - dmell@wustl.edu

Engineer:
Zihao Yu[2] - yu.zihao@wustl.edu

Washington University in St. Louis
McKelvey School of Engineering

[1]Professor, Department of Electrical and Systems Engineering
[2]Candidate for B.S. in Computer Engineering and B.S. in Electrical Engineering (Minor in Robotics)

From Language to Action: A Vision-Language and Kinematics-Guided Framework for 6-DOF Robotic Arm Manipulation

Engineer: Zihao Yu[1] — yu.zihao@wustl.edu

Advisor: Professor Jason Trobaugh[2], Professor Dennis Mell[2]

jasont@wustl.edu, dmell@wustl.edu

Abstract — Mentored by Professor Jason Trobaugh and Professor Dennis Mell, this capstone project developed an intelligent manipulation system for a 6-DOF robotic arm, bridging classical robotics with modern vision-language reasoning. Building on foundations in Robotics Kinematics and Artificial Intelligence, the system enables the robot to interpret natural language instructions, perceive its environment through camera input, and execute accurate, kinematically guided pick-and-place tasks. A Vision-Language Model (Qwen2.5-VL) was integrated to empower natural conversation between the user and the robot arm, parse user instructions, and predict object bounding boxes. A 2D hand-eye calibration procedure was implemented to bridge image-space detections with robot-frame actions, enabling seamless communication between perception and motion. To enhance manipulation of non-uniform objects, a novel method combining EfficientSAM segmentation with PCA-based frame extraction was developed, allowing the gripper to align with object geometry for stable and adaptive grasping. Deployed on the myCobot 280 Pi platform, the system achieved a final placement uncertainty of 0.11 cm using a feedback-based correction loop. The platform remains cost-friendly, modular, and scalable, making it suitable for future integration into ESE-3050 laboratory exercises or demonstration modules in ESE-446. Future work will extend the system into full 3D manipulation by incorporating RGB-D sensing and 3D hand-eye calibration to enable 6-DOF grasp planning and complex scene understanding.

[1]Candidate for B.S. in Computer Engineering and B.S. in Electrical Engineering (Minor in Robotics)
[2]Professors, Department of Electrical and Systems Engineering, Washington University in St. Louis

Contents

1 Introduction

Artificial intelligence (AI) has become deeply embedded in modern society, reshaping everything from daily communication to advanced autonomous systems. The proliferation of intelligent agents, such as large language models (LLMs) like ChatGPT and Gemini, alongside rapid advancements in autonomous vehicles and robotics, underscores AI's transformative potential. Integrating these sophisticated AI technologies directly with robotic systems—especially robotic arms capable of intricate manipulation—remains a significant frontier for research and education, presenting unique challenges and opportunities.

Washington University in St. Louis (WashU) offers robust foundational courses in system modeling, robotics kinematics, and AI, including Control Systems (ESE-441), Robotics: Dynamics and Control (ESE-446), and Natural Language Processing (CSE-527), equipping students with essential theoretical and practical knowledge. Meanwhile, WashU provides valuable hands-on experience through courses such as Introduction to Engineering Design (ESE-205) and Control Systems Design Laboratory (ESE-4480). These offerings collectively enable students to build advanced, intelligent robotic systems by integrating classical methods with emerging AI techniques.

Building on this educational foundation, our capstone project develops a modular robotic manipulation framework that combines state-of-the-art vision-language models with classical robotics kinematics. This intelligent platform bridges language and robotic actions. People can interact with the robot through plain language, and the robot will perform corresponding actions. We successfully implemented this platform on the myCobot 280 Pi, a compact and cost-friendly 6-degree-of-freedom (6-DOF) robotic arm with an onboard Raspberry Pi and a $6 Adafruit Ultra Tiny Camera setup for visual sensing. The overarching goal is to create a scalable and intuitive robotic manipulation system that can be operated through natural language interactions, thereby lowering barriers for non-specialist users and enhancing educational accessibility. This project is a pioneering idea for integrating robotics systems with emerging AI technology and can serve as a meaningful demo for ESE-205, ESE-446, and ESE-4480 in the future.

Our project specifically aims to achieve the following objectives:

1. **Prototype 0:** Develop a foundational hardware and software integration enabling basic robotic manipulation tasks in a 2D environment with intuitive natural language instruction parsing.
2. **Objective 1:** Empower the robot assistant to interpret free-form language, respond naturally, and execute robot actions selectively, enhancing intuitiveness and accessibility.
3. **Objective 2:** Develop a vision-based grasping pipeline using an image segmentation model and PCA to accurately estimate object-centric frames, enabling stable manipulation of irregularly shaped objects.

Specifically, our system makes several novel contributions across hardware, software, and intelligent perception:

First, we designed a **compact and cost-effective hardware platform**, featuring the myCobot 280 Pi robotic arm mounted on a 3D-printed base, equipped with a $6 Adafruit Ultra Tiny camera and a parallel-jaw gripper. This setup enables easy deployment and experimentation in constrained environments like classrooms or small labs.

On the software side, we developed a **modular control framework** built in Python 3 using ROS 2. This architecture facilitates real-time coordination between vision, planning, and actuation modules, and is easily extensible for future research or course integration.

For visual grounding, we implemented a **2D hand-eye calibration method** using an affine transformation to accurately align camera pixels with robot-frame coordinates. This provides a lightweight yet reliable mapping for precise manipulation.

We also enabled **vision-language reasoning** by integrating Qwen2.5-VL-Instruct through Aliyun APIs. This allows the robot to interpret both visual scenes and natural language queries, returning executable commands (e.g., vlm_move(), vlm_vqa()) and contextual dialogue.

We also introduced **a robust grasping pipeline**, combining EfficientSAM for segmentation and PCA-based frame estimation to align grasps with object geometry. This supports the manipulation of irregularly shaped objects. A FastAPI-based voice interface further enhances user accessibility, enabling hands-free, voice-driven control without requiring any onboard audio hardware.

Finally, we developed a **voice interaction module** that enables hands-free control through spoken language. A FastAPI server running on a laptop handles speech recognition and text-to-speech synthesis, communicating with the robot over the network. This design eliminates the need for onboard audio hardware and provides real-time, natural interaction—significantly enhancing accessibility for users without programming experience, including elderly or mobility-limited individuals.

Our framework has demonstrated effective and reliable performance, allowing users to intuitively instruct the robot through conversational and voice interactions. By leveraging speech-to-text and text-to-speech pipelines, users without programming experience—including children, elderly individuals, or those with limited mobility—can seamlessly perform robotic manipulation tasks using only spoken commands. This focus on accessibility lowers traditional barriers and opens the door for broader applications in assistive robotics. With further development, this prototype could serve as the foundation for affordable, voice-controlled robotic helpers in home or healthcare settings.

At the same time, the platform is designed to be modular, affordable, and reproducible, aligning with educational goals. Its integration of theory and practice across AI, system modeling, and robotics kinematics provides an engaging case study for students interested in courses such as ESE-205, ESE-446, and ESE-4480. As such, it stands not only as a technical achievement but also as an educational outcome—bridging classroom knowledge with real-world implementation and inspiring students to explore interdisciplinary robotics development.

2 Background

Before discussing the technical approaches in section 3, this Background section provides basic **Robotic Kinematics Definitions** as well as discussions about the **established works and directions in robotics manipulation in both industry and academia**.

2.1 Robotics Kinematics

Understanding the kinematics of a robotic arm is fundamental for motion planning and manipulation tasks. The mathematical formulation of Forward Kinematics (FK) and Inverse Kinematics (IK) enables us to determine the position and orientation of the robot's end-effector based on joint parameters. [1]

End-Effector Pose and Its Importance

A 6-degree-of-freedom (6-DOF) robotic arm consists of six revolute joints, allowing full control over the position and orientation of the end-effector in 3D space. The end-effector (e.g., a robotic gripper or tool) is the final link of the robotic arm that interacts with the environment. The pose of the end-effector in the base Cartesian frame is represented as:

$$\mathbf{x} = (x, y, z, \theta_x, \theta_y, \theta_z)$$

where:

- (x, y, z) defines the **position** of the end-effector in **Cartesian space**.
- $(\theta_x, \theta_y, \theta_z)$ represents the **orientation** of the end-effector.

These parameters fully describe the spatial configuration of the end-effector, making them critical for tasks such as grasping, welding, painting, and assembly.

Forward and Inverse Kinematics

In robotic manipulators, Forward Kinematics (FK) and Inverse Kinematics (IK) serve complementary roles in computing end-effector motion.

Forward Kinematics **Given:** Joint angles $\theta_1, \theta_2, ..., \theta_6$. **Find:** The end-effector pose:

$$\mathbf{x} = f(\mathbf{q})$$

where:

$$\mathbf{q} = (\theta_1, \theta_2, ..., \theta_6)$$

FK is computed by multiplying homogeneous transformation matrices, using the Denavit-Hartenberg (DH) convention. [1]

Inverse Kinematics **Given:** A desired end-effector pose $(x_d, y_d, z_d, \theta_x, \theta_y, \theta_z)$. **Find:** The required joint angles:

$$\mathbf{q} = f^{-1}(\mathbf{x})$$

IK is more complex than FK because it involves solving nonlinear equations, which may have multiple or no solutions. The primary focus of this project relies on IK as we plan the end-effector position and orientation based on the manipulation planning (pose planning) algorithm.

2.2 Current Trends in Robot Arms

Industrial Robot Arms: Applications and Limitations

Robot arms are one of the most commonly used robotic systems in industrial automation, playing a key role in manufacturing and assembly. These robotic manipulators are designed to perform fast and repetitive tasks like welding, material handling, painting, and assembling products in industries such as automotive and electronics. With actuators, sensors, and pre-programmed motion trajectories, they can operate with high accuracy and efficiency, making them essential for modern production lines. However, traditional robot arms lack adaptability to unstructured environments as they rely on predefined motion trajectories and fixed kinematic models [2]. Without learning-based models, they struggle with uncertainties, requiring manual recalibration for variations in object position, shape, or external disturbances.

Learning-Based Robot Arm Manipulation and its Limitations

In academia, there is more and more attention on powering robot arms to be robust and adaptive. Deep learning has made robotic arms more adaptable and resilient, allowing them to work in dynamic environments and handle new objects without explicit programming. However, because deep learning relies purely on data, it has some major limitations. Unlike traditional kinematics-based control, deep learning doesn't inherently understand the robot's physical constraints, such as joint limits, singularities, or workspace boundaries. This nature implies that neural networks can sometimes generate impossible or unsafe motions, leading to failures, instability, or risky actions, especially in high-precision fields such as surgical robotics or industrial automation. Another drawback is that deep learning models require huge, high-quality datasets to work well, and they struggle in situations they haven't seen before, which can cause unpredictable behavior in real-world applications.

The Need for Hybrid Learning-Kinematics Integration

To overcome these limitations, a hybrid approach that integrates deep learning with traditional kinematics-based models is essential. Kinematic models provide deterministic, physics-based constraints, ensuring feasibility, stability, and safety, while deep learning enables adaptability and perception-driven control. This fusion allows robotic arms to generalize across diverse conditions while maintaining accurate, physically consistent motions.

3 Methods

3.1 System Overview

The developed system consists of a 6-DOF Elephant Robotics myCobot 280 Pi robotic arm equipped with an Adafruit Ultra Tiny Camera for top-down image capture and an adaptive parallel-jaw gripper for object manipulation. A Raspberry Pi serves as the onboard computer, running a modular ROS 2 (Humble) [3] workspace in Python 3 on Ubuntu 22.04. The system communicates with an external Vision-Language Model (Qwen2.5-VL [4]) via API calls to interpret user commands and detect objects within the captured images. A lightweight 2D hand-eye calibration enables mapping between image pixel coordinates and the robot's workspace for accurate pick-and-place execution.

3.2 Devices and Materials

A 6-DOF robot arm was acquired to facilitate hands-on experimentation and benefit the academic community.

- **Robot Arm:** myCobot 280 PI, a 6-DOF robot arm with a Raspberry Pi 4B controller and a 280 mm working range. More information and detailed descriptions in Appendix 9 and Figure 17.

In order to prepare the robot arm ready for operation, after long time reflection and decision making, the following materials are prepared and installed on the robot arm as shown in Figure 1. **Detailed hardware descriptions in Appendix 9**.

- **Gripper:** An adaptive gripper controlled with GPIO pins and installed on the end-effector of the robot arm.
- **Camera:** A six-dollar Adafruit Ultra Tiny Camera. It is attached to the gripper and connected with the Raspberry Pi controller with a USB-2.0 cable. Even if this is a low-cost camera, careful framework design can provide a reliable performance.
- **Stabilization Base:** A 3D printed PLA base. After finishing the 3D printing training at WashU Spartan Makerspace and refining the CAD design, the base was printed successfully.
- **G-clamp:** A G-clamp for stabilizing the 3D-printed base on table.

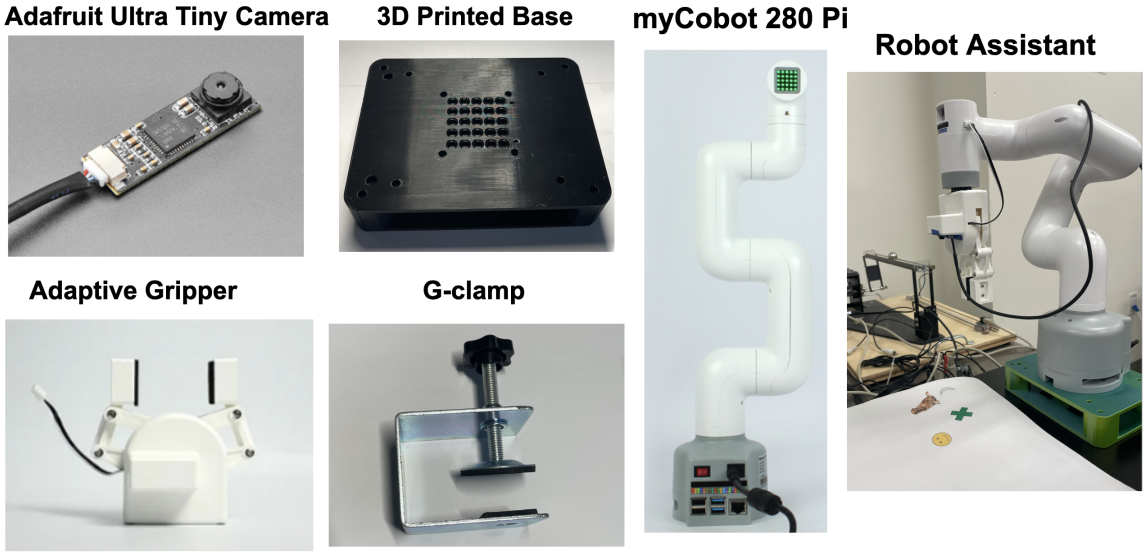


Figure 1: Devices and Materials for building the Robot Assistant. Adafruit Ultra Tiny Camera image from [5]. Adaptive Gripper and myCobot 280 Pi images from [6].

3.3 Existing Resources and Tools

Just as Adafruit libraries in ESE-205 enable students to abstract away low-level GPIO and PWM control, and as TurtleBot platforms allow users to issue high-level motion commands rather than directly controlling actuators, this project builds on existing robotics software to streamline development. Specifically, we use the `pymycobot` library [7] to handle motor-level control, predefined motion execution, and inverse kinematics computations for the robot arm.

By leveraging this software interface, we avoided the need to manually derive forward or inverse kinematic equations or implement custom servo control. This allowed us to focus on higher-level system integration, perception, and task execution, aligning with the practical goals of building a functional and extensible robotics framework.

3.4 Language Understanding and Environment Perception in the Task Execution Framework

3.4.1 Motivation for Integrating Language and Vision

A key challenge in building a general-purpose robotic assistant lies in enabling both robust instruction understanding and flexible visual perception. Traditional approaches rely on rigid keyword extraction (e.g., regular expressions) or domain-specific language models, which often fail on long, complex instructions. Similarly, standard computer vision pipelines—though strong at object detection or segmentation—lack semantic grounding and require manual post-processing to resolve spatial relationships.

For example, while a segmentation model may detect an "apple" and a "pear," it typically lacks the ability to reason about relational terms like "next to" or resolve ambiguous user commands such as "put the green one near the logo." These gaps become critical in open-world tasks that involve both language reasoning and spatial grounding.

To address these limitations, we explored recent advances in Vision-Language Models (VLMs), which unify instruction parsing and scene understanding into a single architecture. After surveying relevant literature in AI and robotics, we selected a VLM-based solution as the backbone of our framework—capable of interpreting natural language prompts, reasoning over visual inputs, and generating structured plans for manipulation tasks.

3.4.2 What are Vision-Language Models?

Vision-Language Models (VLMs) are multimodal architectures that jointly process visual and textual inputs to generate language outputs. [8, 9] These models enable a wide range of high-level tasks such as object recognition, image captioning, visual question answering (VQA), and grounded instruction following. Unlike conventional vision-only models, VLMs learn to align linguistic semantics with image regions by training on massive corpora of image-text pairs.

Formally, given a sequence of textual tokens $x = (x_1, \ldots, x_N)$ and visual features v extracted from an image, a VLM generates an output sequence $y = (y_1, \ldots, y_T)$ autoregressively. The model estimates the conditional likelihood:

$$P(y \mid x, v) = \prod_{t=1}^{T} P(y_t \mid y_{<t}, x, v)$$

where $y_{<t}$ denotes the previously generated tokens. At each step t, the model outputs a probability distribution (logits) over a vocabulary of subword tokens—such as "robot", "the", "move", or punctuation like ","—and selects the most probable next token via an `argmax` operation. This formulation enables the model to dynamically attend to both linguistic context and visual semantics during generation.

Training Phases (Background). Although we do not train any VLMs in this work, it is helpful to understand how modern Vision-Language Models (VLMs) are constructed, which explains the capability of VLMs in our robotics framework. Most state-of-the-art VLMs are trained through a two-stage pipeline [8]:

- ***Pretraining*:** The model begins with a pretrained or instruction-tuned Large Language Model (LLM) and a Vision Encoder. A learnable **projector** is trained to map visual features into the LLM's token embedding space. The model is trained on large-scale image-text pairs using objectives such as image captioning, masked language modeling, and contrastive alignment. This stage injects broad world knowledge and visual grounding ability.
- ***Instruction-tuning*:** The vision encoder, projector, and LLM are fine-tuned end-to-end on curated datasets consisting of image-instruction-response triples. This enables the model to follow user instructions and produce structured outputs in various formats (e.g., natural dialogue, JSON commands).

System and User Prompts. Instruction-tuned VLMs interact through a combination of:

- *System Prompt*: A predefined hidden text block added to the user prompt. [10] It defines the assistant's capabilities, tone, and response formatting. For example, it can require outputs in structured JSON [11] and encourage the model to respond helpfully and humorously.
- *User Prompt*: The visible instruction provided by the user—e.g., "What's on the table?" or "Put the red block on the green logo." The model interprets this instruction in light of the system prompt and the associated visual input. [12]

These prompts work together to control model behavior. In our application, we carefully design the system prompt to encourage the VLM to behave as a conversational and planning-capable robotic assistant. Depending on the user prompt, the model either engages in friendly dialogue or returns structured responses that trigger robot actions. No model retraining is required—only prompt design and downstream integration. **A detailed discussion on the infeasibility of training VLMs and appreciation for API access is provided in Appendix 9.**

VLM Naming Conventions and Model Selection

Vision-Language Models (VLMs) are typically named using a combination of their base language model family, modality, model size, and tuning strategy. For example, the model name `Qwen2.5-VL-3B-Instruct` indicates the following: `Qwen2.5` is the backbone language model, `VL` denotes that the model supports both vision and language inputs, `3B` refers to its size—3 billion parameters—and `Instruct` specifies that the model is instruction-tuned for following user prompts. Similarly, the model used in the conversation module, `Qwen2.5-VL-32B-Instruct`, follows the same convention but with a significantly larger parameter count of 32 billion.

In this project, we selected the `3B` model for the object detection pipeline due to its faster response time and concise answers, which are ideal for quick and frequent vision queries. Conversely, we employed the larger `32B` model in the conversational interface to support richer, more engaging dialogue, and improved task planning through its greater reasoning capacity. **Detailed comparison and analysis on model behaviors are covered in Appendix 9.**

Generalization and Semantic Flexibility

Traditional object detectors like YOLO are limited to fixed label sets (e.g., *car*, *person*, *bicycle*), making them less adaptable to unfamiliar or task-specific objects. In contrast, Vision-Language Models (VLMs) such as `Qwen2.5-VL-3B-Instruct` enable open-vocabulary detection by grounding visual features in natural language prompts.

In our experiments, the VLM successfully identified objects like *"ear plug"*, *"orange"*, and *"plastic block"*, which lie outside typical detection datasets. It also responded accurately to semantic variations (e.g., *"red block"* vs. *"plastic block"*), showcasing greater flexibility and generalization than traditional models that rely on fixed output heads.

3.4.3 Integrating Vision-Language Models into the Framework

To interpret user instructions and ground them visually in the environment, our system integrates a Vision-Language Model (VLM) via an **Application Programming Interface (API)**. Structured prompts—including

a system message, user instruction, and a captured camera image—are sent from the robot to a remote inference server. The server processes the input and returns both a natural language response and a set of structured function calls for robotic execution.

We began development with the locally hosted Qwen-VL-7B [13], which proved effective for early-stage prototyping. However, its memory demands and slower inference speeds posed challenges for real-time usage on consumer-grade hardware. Even relatively lightweight models require tens of gigabytes of GPU memory for smooth performance.

Training a VLM from scratch, while theoretically feasible, would involve massive multimodal datasets, extended training cycles, and industrial-scale compute infrastructure—conditions typically beyond the scope of capstone-focused robotic systems [14]. Instead, we focused on leveraging pretrained models through modular APIs to deliver powerful capabilities within a lightweight ROS 2 framework.

For cloud-based deployment, we adopted the `openai` Python package [15] and authenticated via an API key from Aliyun [16], enabling access to the Qwen2.5-VL [4] model family. This approach allowed us to deploy high-capacity models such as `Qwen2.5-VL-32B-Instruct`, which provided robust multimodal reasoning and response generation without requiring dedicated local hardware.

Alternative techniques like fine-tuning and Retrieval-Augmented Generation (RAG) [17] were explored but considered unnecessary for our needs. Carefully designed prompts with few-shot in-context examples proved sufficient to control model behavior and elicit structured responses. **Detailed analysis in Appendix 9.**

By combining local ROS2 nodes for perception and actuation with scalable cloud-based VLM inference, our system effectively bridges low-level control and high-level understanding—making advanced AI accessible and deployable in real-world robotics applications. See Appendix9 for further discussion.

3.5 Prototype 0: Pick-and-place Objects via Vision Language Understanding

To assess the feasibility of this project, we initially tested the basic functionalities of object recognition, robot arm movement, and pick-and-place objects in a constrained and clean environment.

3.5.1 System Description

The robot arm is fixed on one side of a table, and the only objects on the table are a small number of objects, i.e., a pen and an orange. Once the program starts, the robot arm moves to a default position such that it is positioned to have a top-down view of the table and the end-effector is point vertically above the table with a height $\mathbf{h}$.

At this default position, the pose of the end-effector is represented as:

$$\mathbf{x}_{\text{default}} = (x_0, y_0, z_0, \theta_{x_0}, \theta_{y_0}, \theta_{z_0}) = (x_0, y_0, \mathbf{h}, \theta_{x_0}, \theta_{y_0}, \theta_{z_0}) \tag{1}$$

3.5.2 2D Hand-Eye Calibration via Affine Transformation

To localize objects within the robot's workspace, it is essential to translate 2D camera image coordinates (x', y') into real-world robot end-effector coordinates (x, y). Initially, we implemented a simple linear interpolation method based on two manually labeled calibration points. While this provided coarse estimation, it could not account for image distortion, rotation, or non-uniform scaling, and proved insufficient for precise manipulation.

To overcome this, we reformulated the problem as a 2D affine transformation [18] from camera to robot coordinates. The mapping is expressed as:

$$\begin{bmatrix} x \\ y \end{bmatrix}_{\text{robot}} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix}_{\text{camera}} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{2}$$

or equivalently, in homogeneous coordinates:

$$\begin{bmatrix} x \\ y \end{bmatrix}_{\text{robot}} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}_{\text{camera}} \tag{3}$$

During calibration, the robot end-effector moves in a plane parallel to the workspace at a fixed height $z = h$, with a constant orientation $(r_x, r_y, r_z) = (r_{x0}, r_{y0}, r_{z0})$. We manually guide the end-effector to rest precisely above three known points (e.g., paper corners) visible in the image and record both the image coordinates (x'_i, y'_i) and the corresponding robot positions (x_i, y_i).

From these three correspondences, we derive the following linear system:

$$\mathbf{M} \cdot \mathbf{u} = \mathbf{y}$$

where:

$$\mathbf{M} = \begin{bmatrix} x'_1 & y'_1 & 1 & 0 & 0 & 0 \\ x'_2 & y'_2 & 1 & 0 & 0 & 0 \\ x'_3 & y'_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x'_1 & y'_1 & 1 \\ 0 & 0 & 0 & x'_2 & y'_2 & 1 \\ 0 & 0 & 0 & x'_3 & y'_3 & 1 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} a \\ b \\ t_x \\ c \\ d \\ t_y \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

This system can be solved using least squares:

$$\mathbf{u} = \arg\min_{\mathbf{u}} \|\mathbf{M}\mathbf{u} - \mathbf{y}\|_2^2$$

Once solved, the resulting affine transformation matrix $\mathbf{A} \in \mathbb{R}^{2\times 3}$ is used to map any new image coordinate to the robot frame:

$$\begin{bmatrix} x \\ y \end{bmatrix}_{\text{robot}} = \mathbf{A} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}_{\text{camera}}$$

This calibration process substantially improved localization accuracy and enabled robust visual grounding for 2D pick-and-place operations.

3.5.3 Object Recognition via Vision-Language Prompting

After the 2D hand-eye calibration step, the next challenge is identifying where to pick and place the objects. A key capability of Vision-Language Models (VLMs) is their ability to detect and describe objects in images through natural language interaction. We first tested this by prompting the model informally without a predefined system prompt. For example, given a user query such as *"Where is the orange?"*, the VLM could return the coordinates of the detected orange object within the image. A sample result is shown in Figure 2.

In object detection, the output bounding box is typically represented by two corners: $(x_{\min}, y_{\min})$ refers to the top-left of the box, and $(x_{\max}, y_{\max})$ refers to the bottom-right. In camera coordinate space, the origin $(0,0)$ is at the top-left corner of the image, with x increasing to the right and y increasing downward. These coordinates are used to calculate the center of the bounding box, which is treated as the grasp or place location in pixel space.

To support more complex tasks like pick-and-place, we extended this setup by designing a system prompt that enables the VLM to detect both the starting and target objects from a single instruction. This structured prompt improves response consistency and allows the model to return JSON-formatted bounding boxes for both objects simultaneously. For instance, when given the instruction *"Put the ear plug on the dog"*, the VLM returns the bounding boxes for both the "ear plug" and the "dog."

The following `SYSTEM_PROMPT` was used:

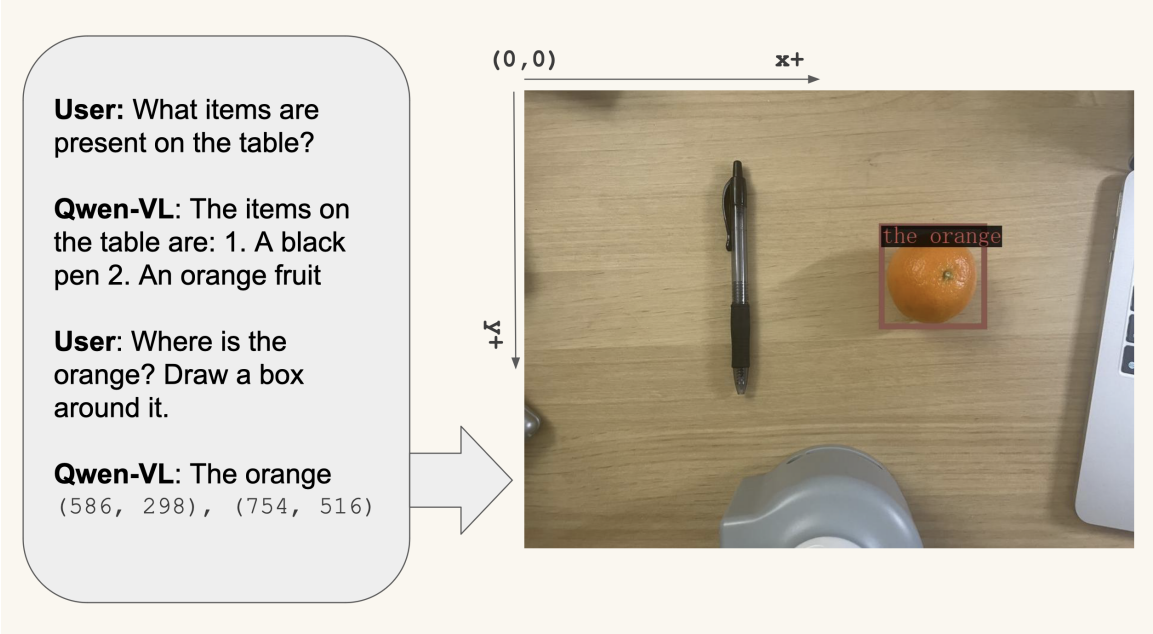


Figure 2: Object recognition through a conversation with Qwen-VL. The user asks "Where is the orange?" and the model returns the top-left and bottom-right pixel coordinates of the bounding box. The image coordinate origin is at the top-left; values increase rightward and downward. (This example was captured using an iPhone prior to integrating the Adafruit Tiny Camera.)

Listing 1: System Prompt used to control the conversational behavior of the robot assistant.

```
SYSTEM_PROMPT = '''
You will receive an instruction intended for a robotic arm. Your task is to extract
the starting object and the destination object from the given instruction, identify
their respective top-left and bottom-right pixel coordinates within the provided
image, and return the data in JSON format.

For example, if the instruction is: "Please place the red block on the house sketch
."
You should return the following JSON structure:
{
"start": "red block",
"start_coordinates": [[xmin, ymin, xmax, ymax]],
"end": "green block",
"end_coordinates": [[xmin, ymin, xmax, ymax]],
 }

Ensure that your response contains only the JSON output, without any additional text
.

Instruction:
'''
```

This prompt-based interaction turns the VLM into an instruction-aware object detector. Given an image and natural-language instruction, the model outputs structured spatial data that identifies both source and destination objects. These bounding boxes are then passed through the affine transformation described earlier to map pixel-space centers into robot-frame coordinates for execution.

3.5.4 Execute Planned Motion via ROS 2 Pipeline

The execution pipeline for Prototype 0 is implemented using a modular ROS 2 architecture comprising four main nodes: `main_node`, `camera_node`, `vlm_detection_node`, and `execution_node`. Figure 3 illustrates

the full data flow.

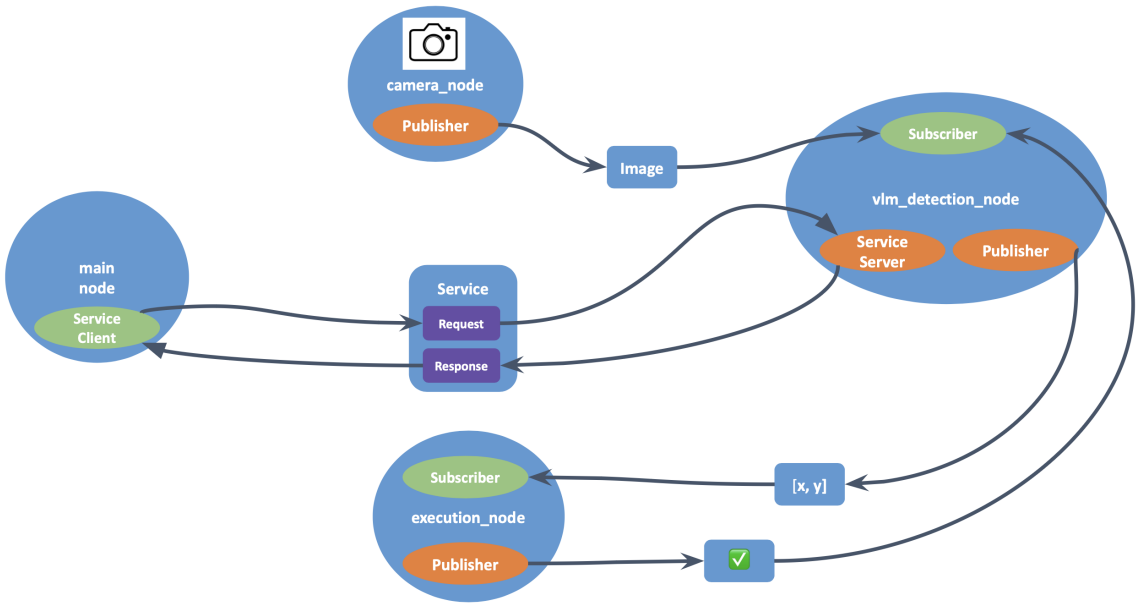


Figure 3: ROS 2-based pipeline of Prototype 0 for detecting and grasping the target object. `main_node` requests object detection service from `vlm_detection_node`, which is actively listening for images published by the `camera_node`. After receiving VLM outputs, `vlm_detection_node` publishes the coordinates, which is listened by the `execution_node`. `execution_node` controls robot motion and publish a success message back to `vlm_detection_node` after execution.

The `camera_node` continuously captures and publishes images from the Adafruit Ultra Tiny Camera at a fixed frequency (e.g., 1 Hz). These images are subscribed to by the `vlm_detection_node`, which performs visual inference when triggered.

The `main_node` listens for user instructions via keyboard and acts as the central controller. Upon receiving a new instruction (e.g., "place the orange on the box"), it sends a request to the `vlm_detection_node`. That node, which remains idle until called, then processes the latest image using Qwen2.5-VL and parses object locations according to the defined system prompt. It extracts bounding boxes for both the source and destination objects and publishes them using a custom message type:

```
robot_interfaces/msg/ObjectDetection.msg

string start_object          string end_object
int16 start_xmin             int16 end_xmin
int16 start_ymin             int16 end_ymin
int16 start_xmax             int16 end_xmax
int16 start_ymax             int16 end_ymax
```

This message format is optimized for speed by using `int16` fields, which are sufficient given the small working range of the robot. The message is then forwarded to the `execution_node`, which uses the center of the start bounding box as the target location. These pixel coordinates are mapped into robot frame coordinates via the calibrated affine transformation.

The final end-effector pose is defined as:

$$\mathbf{x}_{\text{start}} = (x_{\text{start}}, y_{\text{start}}, z_0 - h, \theta_{x_0}, \theta_{y_0}, \theta_{z_0}) \tag{4}$$

$$\mathbf{x}_{\text{target}} = (x_{\text{target}}, y_{\text{target}}, z_0 - h, \theta_{x_0}, \theta_{y_0}, \theta_{z_0}) \tag{5}$$

Using inverse kinematics provided by the `pymycobot` API, the robot moves above the object h lower than the default z_0 height, lowers in the $-z$ direction toward $z_0 - h$ to account for object height, and actuates the gripper to complete the grasp. After grasping, the robot arm returns to the default z_0 plane, moves to the target coordinates, and lowers towards $z_0 - h$ to complete the placement. Once execution is complete, the `execution_node` sends a completion acknowledgment using another custom message:

`robot_interfaces/msg/ExecutionComplete.msg`

This enables the `vlm_detection_node` to verify that the action has finished before proceeding with additional commands or feedback loops. This modular design allows easy integration of additional feedback control or sequential task chaining in future versions.

3.5.5 Evaluation Metric: Pick-and-Place Accuracy

To evaluate the precision of pick-and-place operations, we define a simple but effective metric: the Euclidean distance between the predicted placement location (where the object is supposed to be) and the actual final position of the object in the image after execution.

Given the bounding boxes returned by the Vision-Language Model for both the source and destination objects, we compute the center point of each box using:

$$(x_c, y_c) = \left(\frac{x_{\min} + x_{\max}}{2}, \frac{y_{\min} + y_{\max}}{2} \right) \tag{6}$$

Then, the distance between two objects in pixel space is calculated as:

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{7}$$

This computation is implemented in the ROS 2 workspace and is automatically triggered after each manipulation task.

To report results in physical units, the distance in pixels is converted to centimeters using a scaling factor derived from 2D hand-eye calibration. Based on known dimensions (e.g., A4 paper), we empirically determine a pixel-to-centimeter ratio:

$$\alpha = \frac{\text{real-world length (cm)}}{\text{pixel length (px)}} \tag{8}$$

Thus, the final error in centimeters is:

$$D_{\text{cm}} = \alpha \cdot D_{\text{px}} \tag{9}$$

This metric enables quantitative comparison across multiple trials and provides an interpretable sense of physical accuracy. Detailed trial results are reported in Section 4.

3.5.6 Execution Feedback and Retry Logic

To improve reliability in pick-and-place operations, we implemented a feedback loop within the `vlm_detection_node`. After each execution cycle, the system captures a new image of the workspace and re-evaluates whether the start object has been successfully placed within the bounding box of the target object. This logic is critical for correcting minor errors due to visual drift, robot uncertainty, or slippage during execution.

The verification process works as follows:

1. After execution, the `execution_node` publishes an `ExecutionComplete` message to indicate that motion has finished.

2. The `vlm_detection_node`, upon receiving this message, retrieves the most recent image and re-runs the object detection.
3. It computes the center of the placed object's bounding box and checks whether it falls within the target bounding box using a utility function `is_successful_placement()`.
4. If successful, the task is marked complete.
5. If unsuccessful, the system retries up to `max_retries` times. On each retry, the object detection is repeated and a new pick-and-place command is issued.

The retry count and success condition are managed entirely in the `vlm_detection_node`. The parameter `max_retries` can be configured based on task tolerance and application demands.

This feedback-driven approach not only improves task completion success rate but also enables autonomous correction without requiring external human monitoring. It is especially valuable in low-cost vision-based control setups where small deviations in object location may be common.

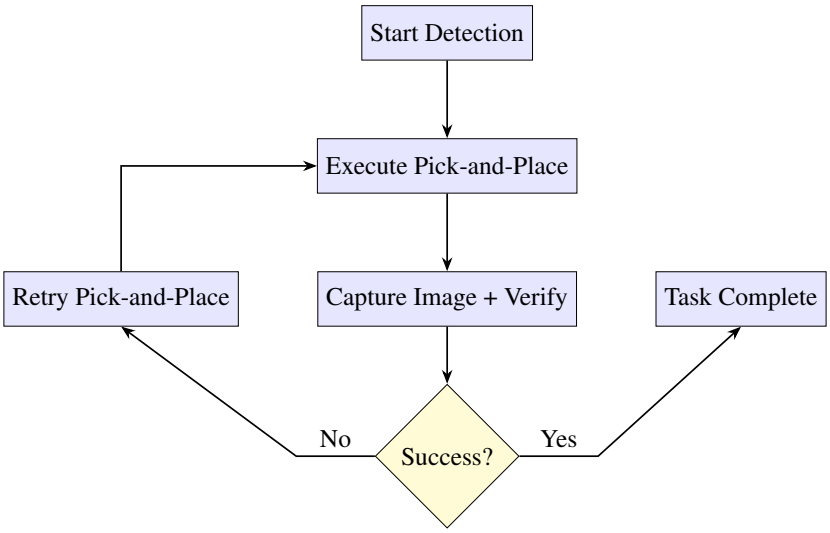


Figure 4: Feedback loop for verifying successful placement and retrying if needed.

3.5.7 Motion Repeatability Test

In the early stages of this project, we observed notable discrepancies between the target and actual positions of the robot end-effector during pick-and-place execution. These inaccuracies prompted us to investigate two possible sources of error: (1) pixel-to-world coordinate calibration and (2) the mechanical repeatability of the robot arm. While calibration improvements are discussed in an earlier section, this subsection focuses on evaluating the robot's intrinsic motion repeatability.

To assess the system's baseline precision, we conducted a controlled test using a printed checkerboard pattern affixed to the workspace. The procedure was as follows:

1. Move the robot to a known "default pose" above the checkerboard and capture an image using the onboard camera.
2. Randomly move the robot to a different position in the workspace.
3. Command the robot to return to the original pose and capture a second image.
4. Repeat the process for ten trials to evaluate consistency.

Each pair of before-and-after images was processed using the Harris Corner Detector [19] to extract a fixed set of keypoints (e.g., checkerboard corners). We then computed the Euclidean distance between corresponding keypoints across trials to quantify the deviation introduced during motion. This image-based method provides a non-contact, high-resolution evaluation of the arm's ability to return to the same physical location—ensuring the underlying hardware is stable enough for high-level planning tasks.

3.6 Objective 1: Empowering Natural Conversation

Objective 1 expands the framework from a direct command-execution system to a conversational assistant capable of understanding free-form language, responding naturally, and invoking robot actions only when appropriate. This conversational interface makes the robot more intuitive and accessible, especially for non-expert users. It also lays the groundwork for scalable, multimodal control through language and vision.

3.6.1 Vision-Language Planning via Structured Dialogue

At the core of this objective is a new planning interface that allows users to interact with the robot using free-form natural language—not only to issue commands, but also to chat, ask questions, or chain multiple tasks in one input. Instead of directly executing every user instruction, the Vision-Language Model (VLM) is designed to interpret the intent, generate a structured action plan, and decide whether to trigger physical motion based on content.

The interaction is mediated through a structured JSON format [11] returned by the VLM for every user input:

```
{
  "function": ["function1(arguments)", "function2(...)"],
  "response": "your short, funny or helpful reply (max 20 words)"
}
```

This dual-field output serves two purposes:

- **function:** A list of robot functions to execute, based on the user's instruction. If no action is needed, this list is left empty.
- **response:** A natural, human-friendly reply used for conversation and clarification.

Two core function types are supported in this framework:

- `vlm_move("instruction")`: This function triggers a manipulation task such as object relocation. The argument is a free-form instruction that describes the intended motion, e.g., `vlm_move("Put the red block on the green logo")`. The system parses this to determine the source and destination objects as well as their coordinates, and controls the robot arm to pick and place corresponding objects.
- `vlm_vqa("question")`: This function enables visual question answering. It allows the user to query the current visual scene—e.g., `vlm_vqa("How many objects are on the table?")`—and prompts the VLM to generate a semantic interpretation of the image.

This structured dialogue mechanism ensures that the robot responds appropriately and safely—executing physical actions only when explicitly required, and providing helpful or humorous replies otherwise. It also separates declarative understanding (via `response`) from imperative actions (via `function`), allowing the same interface to support casual conversation, visual reasoning, and multi-step robotic planning.

By encapsulating robot control as language-driven function calls, this system transforms the robot from a scripted executor into a conversational assistant—capable of dynamic, multimodal task coordination through flexible user input.

3.6.2 System Prompt as an Interface Specification

To define the behavior of the conversational robotic assistant, we employ a structured system prompt that serves as an interface contract between the user and the Vision-Language Model. This prompt outlines the assistant's identity, capabilities, expected output format, and task-specific instructions—all in natural language. It also provides in-context examples that demonstrate how to interpret user input and generate well-structured action plans.

The `SYSTEM_PROMPT` provides role context ("You are my robotic arm assistant..."), defines task-specific expectations, and includes several annotated examples to demonstrate the expected input-output structure. These examples illustrate how the model should respond to both casual dialogue and robot-related instructions.

This approach uses:

- **In-context learning [20]**: The model learns task behavior by observing patterns in the examples during inference time—no fine-tuning or gradient updates are needed.
- **Few-shot prompting [21]**: We provide 5–6 examples directly in the system prompt to teach the model how to structure JSON responses.
- **Function planning logic**: The model is instructed to include robot action calls (e.g., `vlm_move(...)`) only when physical manipulation is required.

Below is the actual system prompt used in the current implementation. It combines language-based behavior specification with embedded few-shot demonstrations to instruct the model on how to distinguish between casual conversation and executable robotic tasks:

Listing 2: System Prompt used to control the conversational behavior of the robot assistant.

```
SYSTEM_PROMPT = '''
You are my robotic arm assistant. You have eyes (a camera) and hands (a robot arm).
You can chat naturally with me like a smart and funny friend, and you can also
understand and execute my commands to move objects.

# Abilities
You can:
1. Respond naturally and casually to my questions and comments.
2. Understand commands like "Please put the red block on the blue box."
3. If a command involves physical interaction, return a JSON object describing the
function(s) to run and your short response to me.

# Built-in Functions
- vlm_move("Analyze the image and move [object A] to [object B]")
- vlm_vqa("Analyze the image and answer a question, e.g., 'How many blocks are on
the table?')

# Output Format
For each user input, respond **only** with a JSON object, and **nothing else**.
Use this structure:
{
  "function": ["function1(arguments)", "function2(...)"],
  "response": "your short, funny or helpful reply (max 20 words)"
}

# Instructions
- If the user input is casual chat (no command), return just a response with an
empty function list.
- If the input includes a clear pick-and-place instruction, use `vlm_move(...)` and
include the key info as its argument.
- Feel free to include jokes, memes, or references
- DO NOT wrap your response in '''json - just return the JSON object directly.

# Examples

User: Hello there
```

```
Output: {"function": [], "response": "Hey there! Always ready to lend a robotic hand
."}

User: What have you been up to?
Output: {"function": [], "response": "Just waiting for you to send me more blocks to
 move!"}

User: Please place the red block on the blue box
Output: {"function": ["vlm_move('Place the red block on the blue box')"], "response
": "Red block incoming! Air-drop onto the blue box!"}

User: Can you check what's on the table?
Output: {"function": ["vlm_vqa('Please check what objects are on the table')"], "
response": "Hold on, opening my eyes wide!"}

User: I had such a productive day!
Output: {"function": [], "response": "You're on fire! Block master level unlocked."}

# Example
User: Put the ear plug on the dog and put the red block on the green logo
Output: {
  "function": [
    "vlm_move('Put the ear plug on the dog')",
    "vlm_move('Put the red block on the green logo')"
  ],
  "response": "Roger that! Ear plug on the dog, red block incoming!"
}

# The next user message is:
'''
```

3.6.3 ROS 2 Integration and Modular Node Design

The conversational interface is implemented within the ROS 2 architecture that connects high-level language understanding to low-level robot control. The `conversation_node` replaces the earlier `main_node`, handling user input and querying the Vision-Language Model (VLM). The model returns a JSON object with two fields: `function` (a list of action calls) and `response` (a user-facing reply).

If the `function` list contains actions like `vlm_move(...)`, the node invokes a service in `vlm_detection_node`, which handles image processing and object localization. The resulting target coordinates are published via a custom `ObjectDetection` message, triggering movement in the `execution_node`.

3.6.4 Voice Interaction for Accessible Control

To enhance accessibility, especially for non-expert users or individuals with mobility impairments, we added a voice interface that supports both speech input and spoken output. This module implemented in `conversation_node` allows users to interact with the robot entirely through natural conversation, without the need to type or use physical interfaces.

The voice interaction is implemented using two open-source Python libraries:

- **Speech-to-Text (STT):** The `speech_recognition` package [22] captures and transcribes user speech into text using a local microphone.
- **Text-to-Speech (TTS):** The `pyttsx3` library [23] converts text responses from the robot into natural-sounding speech, played back via speakers.

Both STT and TTS are hosted on the developer's laptop via a lightweight FastAPI server. The Raspberry Pi running ROS 2 sends user prompts to the server and receives spoken replies, allowing the robot to converse fluidly without requiring onboard microphones or speakers. This design provides real-time responsiveness, supports multilingual voice libraries, and avoids additional hardware costs—making it suitable for inclusive deployments in education or assistive robotics.

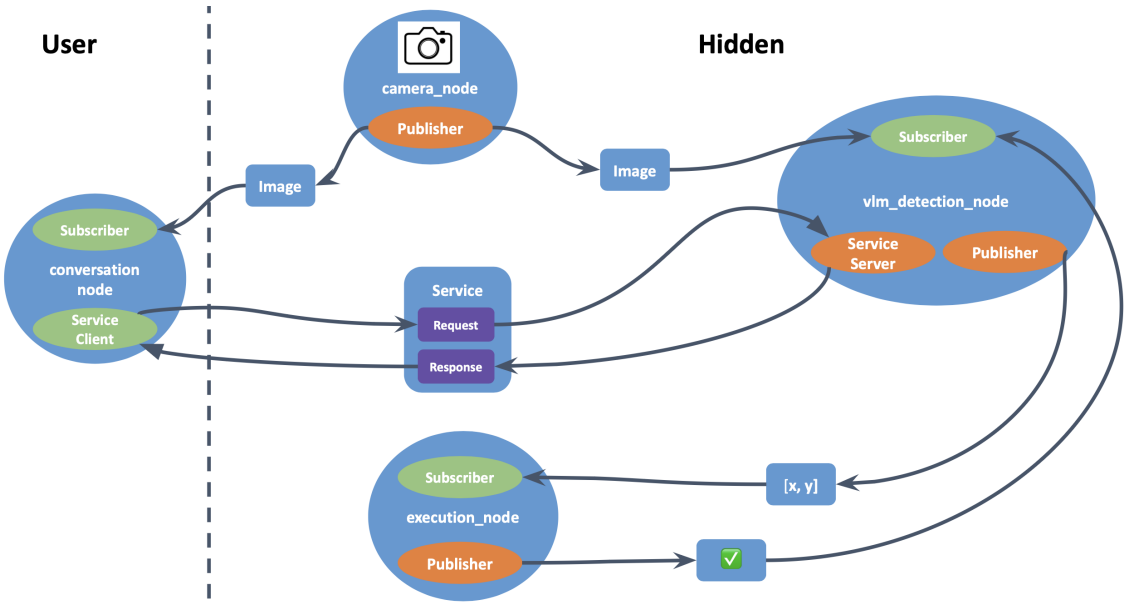


Figure 5: ROS 2-based pipeline of Objective 1 for translating languages to robot actions. This pipeline works similarly as ROS2-based pipeline in Prototype 0. In this pipeline, `conversation_node` serves as the conversation interface for communicating with user through keyboard or voice input. It listens to image published by the `camera_node` and can respond to visual language questions and request `vlm_detection_node` service when necessary.

3.7 Objective 2: 2D Frame Prediction for Non-Uniform Objects

While bounding box centers are sufficient for grasping uniform objects, they often fail for non-uniform shapes like brushes, tools, or bottles, where stable grasping requires orientation-aware alignment. Objective 2 introduces a vision-based method to estimate a local 2D Cartesian frame for each object, enabling geometry-aware manipulation.

Before arriving at the final approach, several alternatives were explored. The initial plan involved training a custom computer vision model to predict object-centric frames directly from RGB images. Using the Grasp Anything dataset [24], over 1 million annotated images were processed. Each image contained multiple labeled grasp rectangles with associated confidence scores; only the most confident grasp was retained. These grasp rectangles—originally defined by center coordinates, width, length, and angle—were converted into a frame representation using three point pairs: the center, x-axis endpoint, and y-axis endpoint, an example shown in Figure 6.

Multiple architectures were tested. A lightweight Convolution Neural Network (CNN) [25, 26] trained with mean squared error proved slow to converge and failed to generalize. A follow-up attempt using a frozen ResNet-18 backbone [27] with a custom output head also performed poorly, likely due to ResNet's semantic focus rather than geometric structure prediction. Figure 6 (right) illustrates a typical weak prediction from this setup. We then explored modifying the output head of the Segment Anything Model (SAM) [28], an image segmentation model with strong semantics understanding and spatial reasoning capability, to predict confidence maps over frame axes. However, the model required extensive training time, failed to converge under resource-limited conditions, and lacked robustness on small datasets. These experiments revealed the limitations of end-to-end learning in this context and motivated a pivot toward a simpler and more interpretable solution.

The final pipeline builds on the output of the Vision-Language Model (VLM), which provides bounding boxes of the source and target objects. These bounding boxes are used to prompt a segmentation model (EfficientSAM [29]), which is hosted on an external laptop and accessed via FastAPI by the Raspberry Pi controller. The segmentation model returns a binary mask isolating the object, which is then processed using

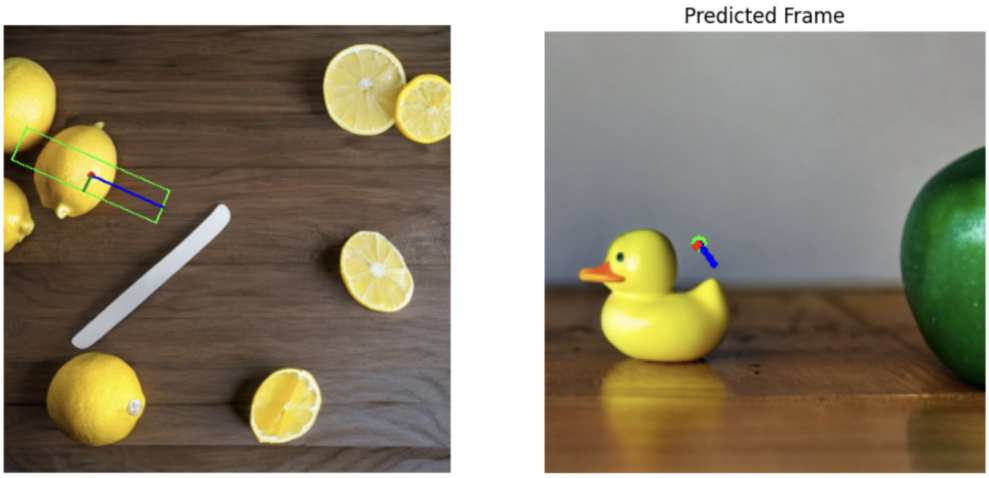


Figure 6: Left: Processed frame label for a Grasp Anything image, converted from rectangle format. Right: Weak frame prediction from a ResNet-18 model fine-tuned for frame regression.

Principal Component Analysis (PCA) to extract the dominant geometric axes of the object.

This predicted 2D frame is used to compute the orientation of the robot's gripper, aligning it with the principal direction of the object for more stable grasping.

3.7.1 2D Frame Estimation Using PCA

Given a binary segmentation mask, foreground pixel coordinates are extracted:

$$\mathbf{X} = \{(x_i, y_i) \mid \text{mask}(x_i, y_i) > 0\}$$

Step 1: Compute the Center of Mass

$$\boldsymbol{\mu} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{x}_i$$

Step 2: Compute the Covariance Matrix

$$\Sigma = \frac{1}{N} \sum_{i=1}^{N} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T$$

Step 3: Extract PCA Axes

$$\text{PCA}(\mathbf{X}) = \text{eigenvectors}(\Sigma) = [\vec{X}, \vec{Y}]$$

Here, $\vec{X}$ and $\vec{Y}$ are the orthogonal principal directions, and $\boldsymbol{\mu}$ represents the grasp center.

3.7.2 Gripper Pose Construction from the Predicted Frame

The predicted frame is used to align the gripper's orientation with the object's shape:

- **Origin:** $\boldsymbol{\mu}_{robot}$ — object's center of mass
- **X-axis:** $\vec{X}_{robot}$ — grasp alignment direction
- **Y-axis:** $\vec{Y}_{robot}$ — orthogonal axis

The resulting 3D homogeneous transformation matrix used for motion planning is:

$$T_{\text{gripper}} = \begin{bmatrix} X_x & Y_x & 0 & \mu_x \\ X_y & Y_y & 0 & \mu_y \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, h is the fixed height of the gripper above the surface during planar grasping. This transformation defines both the target position and orientation of the robot's end-effector for stable and geometry-aware manipulation. In the current 2D experiment, the orientation angle of the principal component is utilized for adjusting the gripper alignment.

4 Results

4.1 2D Hand-Eye Calibration: Visual Validation

To enable image-to-robot coordinate mapping, we implemented a 2D hand-eye calibration using OpenCV [30]. Three calibration points were manually labeled in a test image captured from the Adafruit Ultra Tiny Camera and matched with the robot-frame coordinates collected. An affine transformation matrix was computed using `cv2.getAffineTransform()`:

```
M = cv2.getAffineTransform(pts_image, pts_robot)
```

We validated the mapping by clicking on random image points and commanding the robot to move to the predicted locations. The end-effector consistently landed within 0–0.3 cm of the target, with smaller errors near the image center. Compared to an earlier linear interpolation approach (1–3 cm error), the affine method provided reliable positioning for downstream manipulation tasks and was integrated into the final ROS 2 system.

4.2 Repeatability of Robot Arm Positioning

Our repeatability test revealed that the robot arm consistently returned to the same pose with minimal variation. Across 10 trials, the Harris Corner Detector extracted 12 consistent feature points per image as shown in Figure 7. As shown in Table 1, the majority of keypoints showed less than 5-pixel standard deviation, with all variations falling below 14 pixels. For the experiment setup, 1 pixel ≈ 0.0189 cm. The variation corresponds to a spatial error of approximately 1–2 mm in the workspace, confirming reliable mechanical precision.

Combined with affine-based hand-eye calibration, this result provided a stable foundation for the system's grasp planning and visual alignment tasks.

4.3 Prototype 0: Node Communication and Functionality

To validate the baseline system integration, four ROS 2 nodes were developed and launched in sequence. Each node was tested for correct communication, service behavior, and functional responsibilities within the pick-and-place pipeline.

1. camera_node

This node captures real-time images using the Adafruit Ultra Tiny Camera and publishes them to the `/camera/image` topic at a 2-second interval as shown in Figure 8. Upon startup, the terminal confirms successful initialization and camera access:

```
[INFO] [camera_node]: Camera node started. Streaming images
[INFO] [camera_node]: Image published.
```

Index	Mean X	Mean Y	Std X	Std Y	Min X	Min Y	Max X	Max Y	**Range X**	**Range X (cm)**	**Range Y**	**Range Y (cm)**
0	461.5	478.0	2.9	0.0	456.0	478.0	465.0	478.0	9.0	0.17	0.0	0.00
1	457.7	1.0	4.6	0.0	449.0	1.0	463.0	1.0	14.0	0.26	0.0	0.00
2	462.9	325.4	3.9	2.2	457.0	322.0	467.0	329.0	10.0	0.19	7.0	0.13
3	199.8	1.0	4.1	0.0	192.0	1.0	204.0	1.0	12.0	0.23	0.0	0.00
4	462.1	69.0	4.3	2.9	454.0	65.0	467.0	74.0	13.0	0.25	9.0	0.17
5	198.1	478.0	2.8	0.0	193.0	478.0	201.0	478.0	8.0	0.15	0.0	0.00
6	638.0	325.0	0.0	2.7	638.0	322.0	638.0	330.0	0.0	0.00	8.0	0.15
7	199.2	327.1	3.3	2.9	194.0	322.0	203.0	331.0	9.0	0.17	9.0	0.17
8	198.4	67.0	3.9	3.3	193.0	63.0	204.0	73.0	11.0	0.21	10.0	0.19
9	638.0	68.1	0.0	2.9	638.0	65.0	638.0	73.0	0.0	0.00	8.0	0.15
10	1.0	68.9	0.0	3.5	1.0	63.0	1.0	75.0	0.0	0.00	12.0	0.23
11	1.0	328.0	0.0	3.3	1.0	323.0	1.0	334.0	0.0	0.00	11.0	0.21

Table 1: Statistics for each point captured by Harris Corner Detector: mean, standard deviation, min, max, and range for X and Y camera coordinates. There are 12 feature points detected indexed from 0 to 11. The Adafruit Ultra Tiny Camera has a resolution of 640x480 pixels.

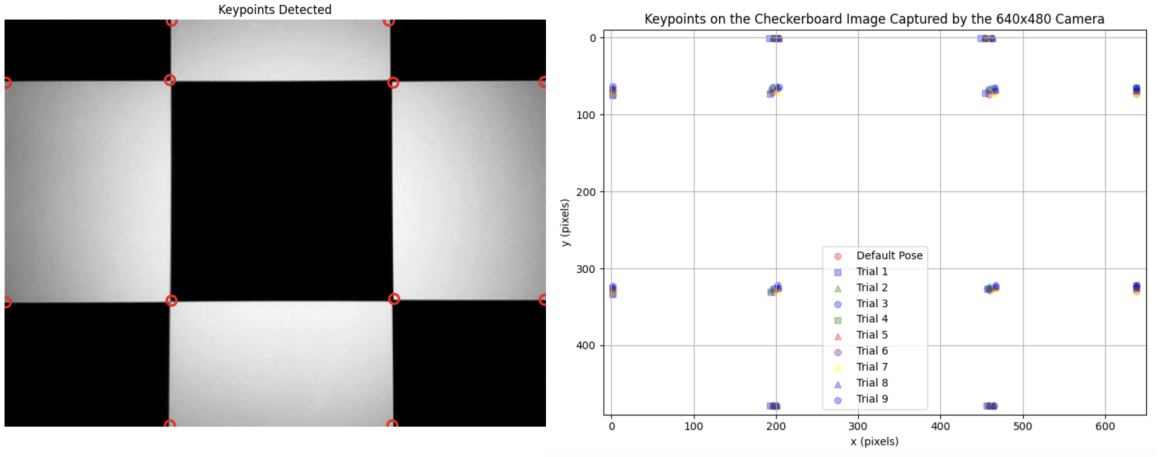


Figure 7: The captured key points of the checkerboard from the Harris Corner Detector. Ten trials are recorded and utilized for evaluating the repeatability of robot arm positioning. There are 12 key points detected from the checkerboard in the field of view of the Adafruit Ultra Tiny Camera.

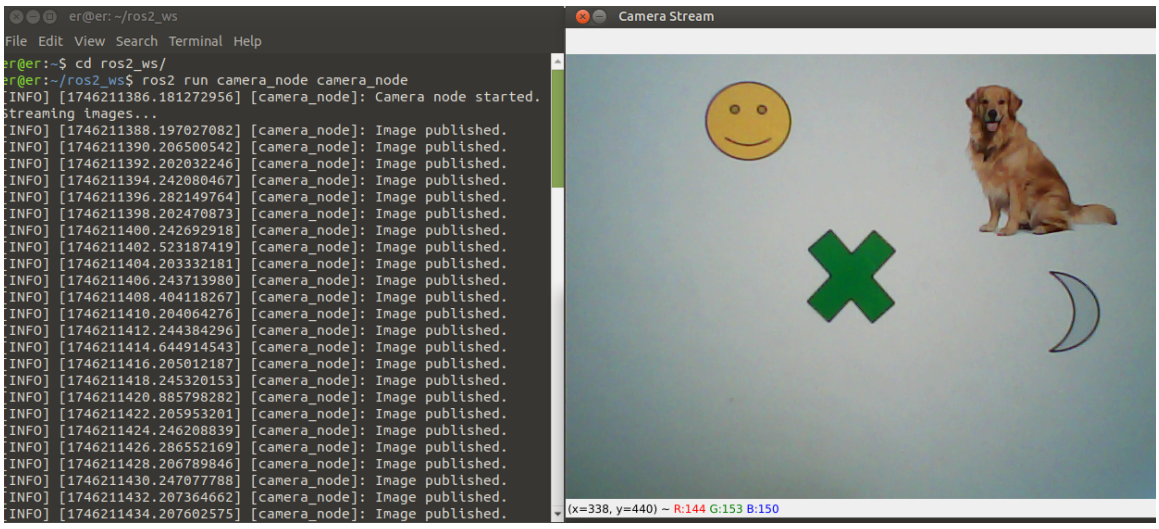


Figure 8: Left: The terminal running camera_node keeps publishing image. Right: The video stream shows the view of the Adafruit Ultra Tiny Camera in real time.

2. main_node

The `main_node` listens to user input and sends prompt strings to the `vlm_detection_node` using a ROS 2 service call as shown in Figure 9. It waits for service availability and logs each prompt request:

```
[INFO] [main_node]: Waiting for detection service
[INFO] [main_node]: Sending detection request: "Put the red block on the dog."
```

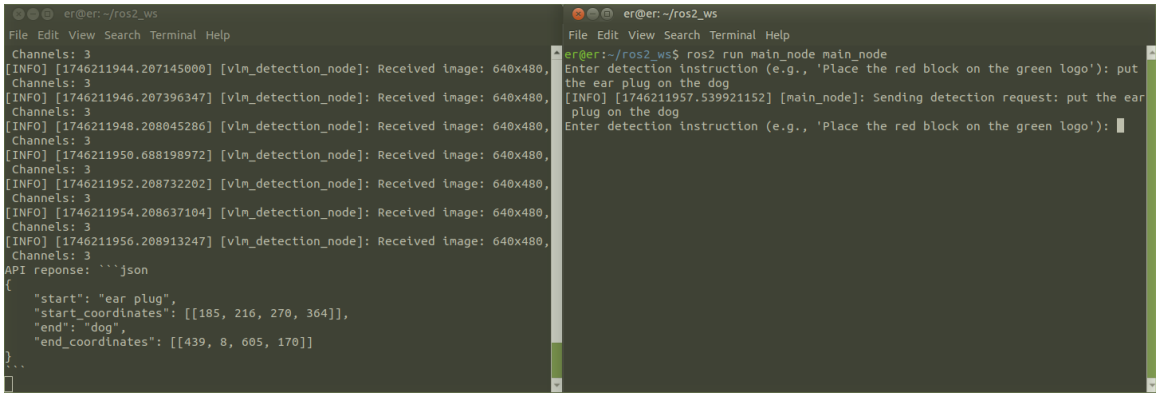


Figure 9: Right: User types commands in main_node. Left: vlm_detection_node is listening for images continuously. It is requested for object detection by main_node to move ”ear plug” to ”dog”.

3. vlm_detection_node

This node subscribes to `/camera/image`, extracts pixel coordinates using the VLM API, and publishes detected bounding boxes as a `ObjectDetection` message. It also listens for robot execution completion to verify task success and trigger retry logic:

```
[INFO] [vlm_detection_node]: Object Detection Node started and subscribed to /camera/image
[INFO] [vlm_detection_node]: Received image: 640x480, Channels: 3
[INFO] [vlm_detection_node]: Placement unsuccessful, retrying
[INFO] [vlm_detection_node]: Placement successful!
```

An example of receiving the request for *”put the ear plug on the dog”* is shown in Figure 10. The vlm_detection_node prints the detection results in the terminal, and the bounding boxes are visualized in an image.

4. execution_node

After receiving a detection message, the `execution_node` performs the grasp-and-place motion using inverse kinematics and sends a confirmation upon task completion:

```
[INFO] [execution_node]: Execution node initialized and listening for detected objects.
[INFO] [execution_node]: From red block -> dog
[INFO] [execution_node]: Pick-and-place operation complete.
[INFO] [execution_node]: Robot execution complete message published.
```

This modular test confirms that all four ROS 2 nodes can communicate reliably across topics and services, and that visual perception and physical manipulation are correctly connected. Each module behaves predictably, laying a robust foundation for expanded logic in subsequent objectives.

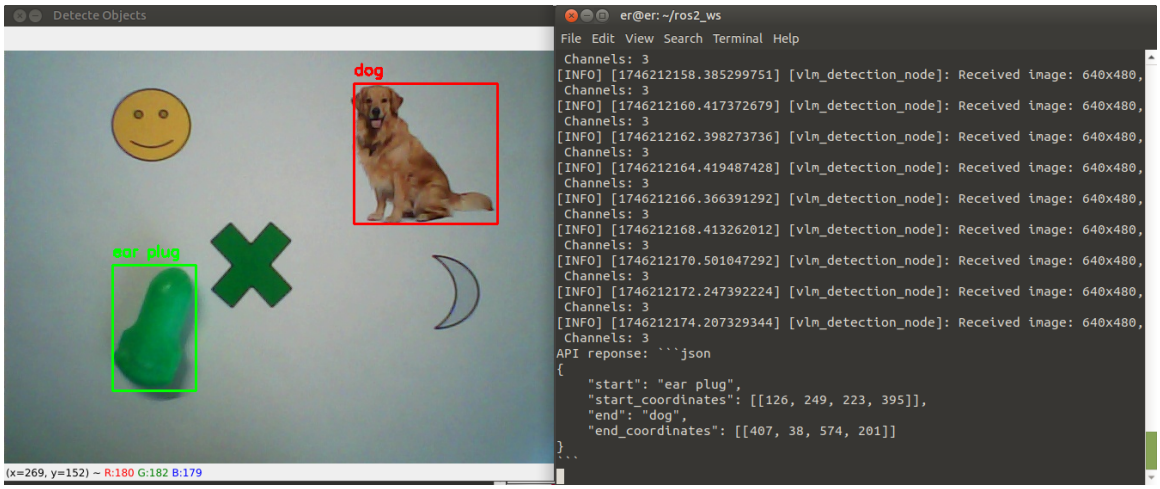


Figure 10: Right: vlm_detection_node receives a request to detect objects and returns the outputs in JSON format as shown in the terminal. Left: an image with the detected bounding box shown on the desktop. It helps the user to check whether the object detection is accurate.

4.4 Evaluation of Pick-and-Place Accuracy

To assess the precision of our vision-based control system, we conducted 16 trials in which the robot was instructed to place a small object (ear plug) on various target landmarks (e.g., dog, green cross, moon). For each trial, the robot arm executed the task based on bounding box detections from a vision-language model and refined placement through a feedback loop if necessary.

The primary metric for evaluating success was the Euclidean distance between the detected center of the placed object and the center of the target bounding box. Each detection was reported in pixel coordinates and converted to centimeters using a calibrated linear mapping:

$$1 \text{ pixel} \approx 0.0189 \text{ cm}$$

The robot performs placement up to three times if the initial placement was inaccurate. The system reported a placement as successful when the object's center fell within the target bounding box, with retries improving precision as needed. While the pick-and-place typically succeeds qualitatively, users can customize their success criteria in the framework.

Table 2 summarizes the outcome of each trial. The feedback mechanism was especially beneficial, reducing placement errors to as low as **0 pixels ($\approx$ 0 cm)** in some trials. First attempts generally resulted in distances ranging from 36.59 to 60.42 pixels ($\approx$ 0.69 cm to 1.14 cm), demonstrating the potential and limitations of VLM-based detection under real-time robotic control.

Importantly, most failed trials stemmed not from mechanical inaccuracies but from perception errors during the feedback loop. After the object had been correctly placed, subsequent image frames occasionally failed to identify the target object. This was often due to overlapping—for instance, the ear plug (being larger) would block the moon, making it no longer visible. In these cases, the VLM sometimes hallucinated a similar object—like a yellow smiley face—as the moon, likely due to similarities in color and shape.

These failure cases highlight the limitations of semantic-only perception in robotic systems. They underscore the need for additional sensing or better temporal integration of object context over time. More broadly, this emphasizes the importance of building robust, multimodal systems when deploying vision-based control in real-world environments.

Trial	Start Object	Start Coords (px)	End Object	End Coords (px)	Pixel Distance	Error (cm)	# of Tries
1	ear plug	(164, 242)	dog	(461, 111)	47.13	0.89	2
2	ear plug	(261, 129)	dog	(453, 124)	58.54	1.11	2
3	ear plug	(127, 142)	dog	(439, 114)	Fail	Fail	3
4	ear plug	(268, 347)	dog	(440, 103)	37.73	0.71	3
5	ear plug	(438, 332)	dog	(428, 138)	57.24	1.08	2
6	ear plug	(362, 152)	dog	(526, 154)	4.74	0.09	2
7	ear plug	(479, 224)	green cross	(297, 297)	0	0.00	2
8	ear plug	(568, 322)	green cross	(302, 298)	Fail	Fail	3
9	ear plug	(362, 153)	green cross	(317, 290)	32.75	0.62	1
10	ear plug	(372, 224)	green cross	(245, 216)	0	0.00	2
11	ear plug	(263, 101)	green cross	(225, 233)	0	0.00	2
12	ear plug	(181, 257)	moon	(443, 245)	Success	Success	1
13	ear plug	(240, 273)	moon	(479, 275)	38.20	0.72	3
14	ear plug	(207, 229)	moon	(446, 241)	Success	Success	1
15	ear plug	(369, 136)	moon	(426, 269)	Success	Success	1
16	ear plug	(207, 318)	moon	(319, 257)	Success	Success	1

Table 2: Summary of pick-and-place trials showing the detected start and end positions (in pixel coordinates), the pixel distance moved by the object, and the estimated error in centimeters after vision-guided correction. Trials marked as "Success" indicate correct placement in a single trial but fail when checking their distances (i.e when the ear plug hides the moon from the camera view.)

4.5 Objective 1: Natural Conversation and Robotic Planning via System Prompts

As introduced in Section 3.6, the behavior of the conversational robot is shaped by a carefully crafted *system prompt*. This prompt establishes the tone, enforces structured output, and enables dynamic function planning by the Vision-Language Model (VLM). The model returns a JSON object containing:

- `function` — a list of robot functions to execute (e.g., `vlm_move()`, `vlm_vqa()`).
- `response` — a short, friendly message for user interaction.

The Table 3 below showcases how the prompt allows the robot to alternate between casual conversation and multimodal task execution. Each output reflects both natural response generation and function-level reasoning.

These examples confirm that the VLM can fluently respond to diverse instructions while invoking appropriate robotic actions. This conversational interface enhances usability and highlights the role of structured prompts in enabling multimodal reasoning within our ROS 2 pipeline.

Crucially, the generated functions are dispatched and executed in sequence within our ROS 2-based architecture, enabling modular and asynchronous integration of perception, planning, and control—described in the next subsection.

4.6 VLM-Powered Interaction via `conversation_node` in ROS 2

The `conversation_node` serves as the central interface for natural interaction in our ROS 2 framework (Figure 5). Replacing the original `main_node`, it handles text or voice inputs, queries the Vision-Language Model (Qwen2.5-VL-32B-Instruct), and dispatches function calls and spoken responses accordingly.

The node subscribes to images from `camera_node` and receives user prompts via keyboard or voice API on the laptop. It then sends the prompt and image to the VLM, which returns a structured JSON response. If the response includes `vlm_move`, the node calls a ROS 2 service in `vlm_detection_node` for object detection and motion planning. If `vlm_vqa` is present, it queries a FastAPI endpoint to analyze the image and returns an answer, which is spoken aloud via a remote Text-to-Speech (TTS) server.

Interaction Type	User Input	VLM Output
Friendly Greeting	`"How are you today?"`	`{"function": [], "response": "I'm just a robot, but I'm here and ready to help!"}`
General Knowledge Query	`"What is Linear Algebra?"`	`{"function": [], "response": "Linear algebra is like the math of organizing things in rows and columns, like a spreadsheet."}`
Multi-step Robotic Plan	`"Put the pen on the orange and the orange on the pen."`	`{"function": [ "vlm_move('Put the pen on the orange')", "vlm_move('Put the orange on the pen')"], "response": "Sure thing! Pen meets orange, then orange meets pen|like a fruit-and-stationery dance!"}`
Visual Q&A with Action	`"Can you tell me what drink is in the cup and put a sugar cube in it?"`	`{"function": [ "vlm_vqa('What drink is in the cup?')", "vlm_move('Put a sugar cube into the cup')"], "response": "Sure thing! Let me check what's in that cozy mug and sweeten it up for you!"}`

Table 3: Representative VLM outputs demonstrating natural dialogue and structured planning via `vlm_move()` and `vlm_vqa()` function calls.

This modular interaction allows the robot to dynamically alternate between conversation and execution, coordinating high-level planning with low-level ROS 2 control.

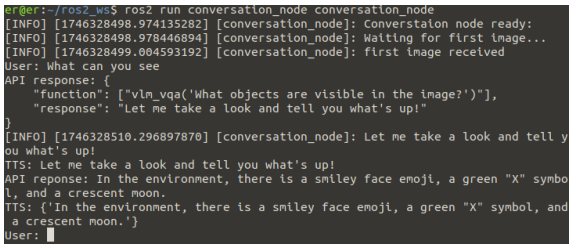
Example ROS 2 logs during an interaction:

```
[INFO] [conversation_node]: Conversation node ready:
[INFO] [conversation_node]: Waiting for first image...
[INFO] [conversation_node]: First image received
[INFO] [conversation_node]: Requesting voice input from laptop...
[INFO] [conversation_node]: Voice input: "Put the red block on the green logo"
[INFO] [conversation_node]: vlm_move completed.
```

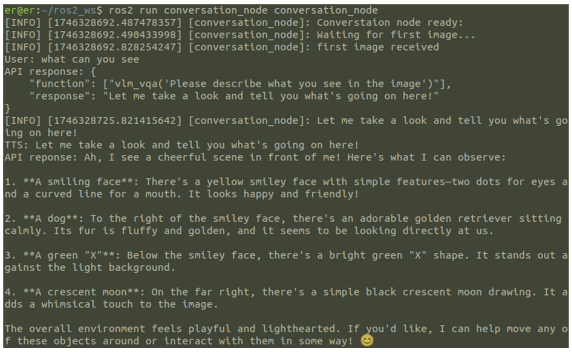
4.7 Grasp Frame Orientation Estimation via EfficientSAM and PCA

To enable orientation-aware manipulation of non-uniform objects, we evaluated the pipeline's ability to extract local 2D frames from segmented masks. When a detected object's bounding box exceeded 200 pixels in width or height, the `vlm_detection_node` triggered a remote API call to EfficientSAM for pixel-level segmentation. The resulting mask was processed using PCA to extract the object's principal axis and center of mass.

Figure 12 shows an example: bounding boxes from the VLM were refined into binary masks, and PCA produced two orthogonal axes visualized as red (X) and green (Y) lines over the object. The X-axis orientation angle θ was computed from the first principal component and published in the `ObjectDetection` message as `rotation_angle_start` and `rotation_angle_end`.



(a) Qwen2.5-VL-3B-Instruct: fast but concise.



(b) Qwen2.5-VL-32B-Instruct: slower but more informative.

Figure 11: Comparison of responses between 3B and 32B Qwen models.

The estimated orientation angle was used to align the robot's gripper with the object's principal axis, improving grasp stability for non-uniform shapes. The full pipeline ran in real time, with segmentation and PCA offloaded to a laptop server, while perception and actuation remained on the Raspberry Pi.

Additional qualitative results of 2D frame labeling tested on the Cornell Grasp Dataset [31] are presented in Appendix 9.

Figure 12: Left: VLM-generated bounding box. Center: segmentation mask from EfficientSAM. Right: 2D frame overlaid using PCA. Red = X-axis (first principal component), Green = Y-axis (second component), $\theta = 59.42°$

4.8 External API Server Deployment for Perception and Voice Modules

To enable lightweight real-time execution on the Raspberry Pi without compromising performance, two external FastAPI servers were deployed on a local laptop: one for image segmentation (EfficientSAM) and one for speech-based interaction. These servers allowed compute-intensive or I/O-heavy modules to run remotely while maintaining modular ROS 2 integration.

EfficientSAM Server for Object Segmentation

The segmentation server hosted EfficientSAM and was launched using:

```
uvicorn fastapi_grasp_server:app --host 0.0.0.0 --port 8001
```

Once running, the server handled segmentation requests sent from the Raspberry Pi when large object bounding boxes were detected. Each request triggered a '/process' endpoint, and successful segmentation

results were confirmed with HTTP 200 responses. Figure 13 shows the running server with POST request logs confirming real-time interaction.

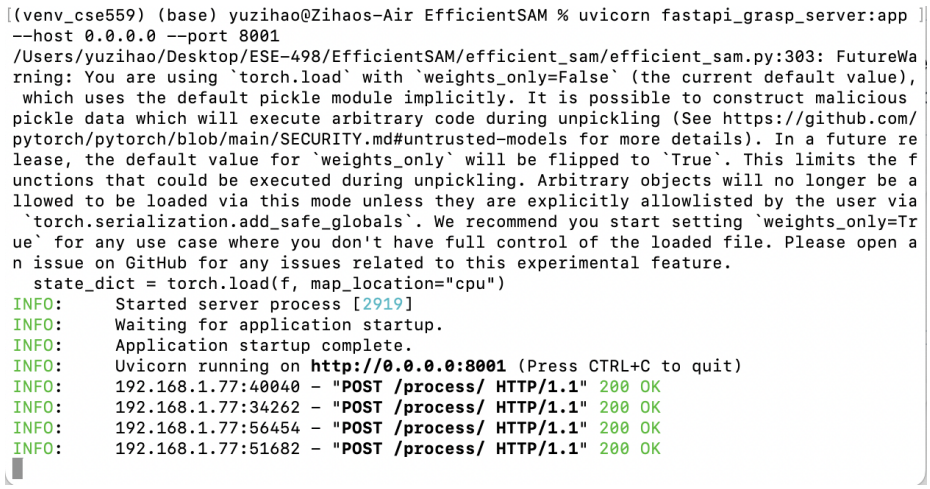


Figure 13: EfficientSAM server logs showing successful image segmentation requests from Raspberry Pi over the local network.

Voice Interaction Server

To support speech input and output, a separate server was deployed on the same laptop using:

```
python voice_server.py
```

The server used the ‘speech_recognition‘ and ‘pyttsx3‘ packages to process user speech and synthesize robot responses. It exposed two endpoints: ‘/listen‘ for capturing user commands and ‘/speak‘ for generating spoken replies. The Raspberry Pi triggered these endpoints during dialogue-based interactions.

Figure 14 shows the server logs capturing voice input, generating a response, and issuing queued speech output for a multi-turn interaction with the robot.

This architecture offloads demanding computations while maintaining low-latency communication between the Raspberry Pi and laptop. It also ensures that voice and vision pipelines are modular, easily replaceable, and adaptable to new hardware or tasks.

4.9 Overall Performance of the Framework

After testing each component individually and in integrated scenarios, the framework demonstrates robust performance and high reliability. Users successfully interacted with the robotic platform using natural conversational commands, accurately interpreted by the Vision-Language Model (VLM). Real-time speech interactions enabled intuitive control and immediate task execution.

Each module was evaluated over five trials, noting that timing varied depending on server load, network latency, and prompt complexity. Specifically, the 32B VLM took longer for `vlm_vqa` due to generating more extensive autoregressive responses, whereas the 3B VLM required additional time for object detection due to lengthier prompts and outputs. Detailed timing results are summarized in Table 4.

```
(base) yuzihao@Zihaos-Air ~ % source venv_cse559/bin/activate
(venv_cse559) (base) yuzihao@Zihaos-Air ~ % cd Desktop/ESE-498/EfficientSAM
(venv_cse559) (base) yuzihao@Zihaos-Air EfficientSAM % python voice_server.py
INFO:     Started server process [3480]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
🎤 Listening...
🧠 Transcript: what can you see
INFO:     192.168.1.77:54848 - "GET /listen HTTP/1.1" 200 OK
🔁 Queued speech: Let me take a look and tell you what I see!
🔊 Speaking via macOS say: Let me take a look and tell you what I see!
INFO:     192.168.1.77:45592 - "POST /speak HTTP/1.1" 200 OK
🔁 Queued speech: Ah, I see a cheerful scene in front of me! Here's what I can observe:

1. **A smiling dog**: A golden retriever is sitting calmly on the right side of the image. It looks happy and r
elaxed.
2. **A yellow smiley face**: To the left of the dog, there's a simple yellow circle with two dots for eyes and
a curved line for a mouth, giving it a friendly expression.
3. **A green "X" shape**: Below the smiley face, there's a green cross or "X" shape.
4. **A crescent moon**: On the far right, there's a black crescent moon drawing.

The background is plain white, which makes these elements stand out nicely. It feels like a playful and positiv
e environment! 😊

What would you like me to do next? Should I move any of these objects around?
🔊 Speaking via macOS say: Ah, I see a cheerful scene in front of me! Here's what I can observe:

1. **A smiling dog**: A golden retriever is sitting calmly on the right side of the image. It looks happy and r
elaxed.
2. **A yellow smiley face**: To the left of the dog, there's a simple yellow circle with two dots for eyes and
a curved line for a mouth, giving it a friendly expression.
3. **A green "X" shape**: Below the smiley face, there's a green cross or "X" shape.
4. **A crescent moon**: On the far right, there's a black crescent moon drawing.

The background is plain white, which makes these elements stand out nicely. It feels like a playful and positiv
e environment! 😊

What would you like me to do next? Should I move any of these objects around?
```

Figure 14: Voice server log showing received user speech, queued spoken response, and robot reply.

Component	**Time Range (s)**
32B VLM (Conversational Interface)	6.09 – 6.70
3B VLM (Object Detection)	4.31 – 4.66
Single Pick-and-Place Operation	15.02 – 15.08
3B VLM (VQA Short Response)	3.56 – 4.06
32B VLM (VQA Informative Response)	8.97 – 10.22
Composite Instructions	**Total Time (Fastest)**
Single Movement (Conversational + Detection + Pick-and-Place)	$6.09 + 4.31 + 15.02 = 25.42$
Conversational + Short VQA Response	$6.09 + 3.56 = 9.65$
Conversational + Informative VQA Response	$6.09 + 8.97 = 15.06$

Table 4: Execution time for individual components and composite tasks.

On a MacBook Air equipped with an Apple M3 chip, the average inference time for EfficientSAM ranged from 1.62 to 1.82 seconds, depending on system load and concurrent processes. The subsequent PCA-based frame estimation was highly efficient, completing in approximately 0.001 seconds. These timings demonstrate that the full 2D grasp frame prediction pipeline is lightweight enough for real-time or near real-time use in typical prototyping setups.

Additionally, the geometry-aware grasping pipeline consistently achieved precise alignment and stable manipulation of non-uniform objects. Representative interactions and detailed performance results are presented in Table 5, highlighting the system's practical effectiveness and potential for real-world deployment. Future enhancements will further optimize usability and accessibility, aiming to support a broader range of users, including elderly and mobility-limited individuals.

User Prompt	API Response	Start Coordinates	End Coordinates	PCA Angle (rad)	VQA Response
Can you put the teddy bear on the dog	{"function": ["vlm_move('Put the teddy bear on the dog')"], "response": "Sure thing! Teddy bear's going for a ride on the dog!"}	[245, 276]	[259, 119]	0.8525	None
Can you tell me what can you see and then put the teddy bear on the dog	{"function": ["vlm_vqa('What objects are visible in the image?')", "vlm_move('Put the teddy bear on the dog')"], "response": "Sure thing! Let me take a look and then place that teddy bear on the dog. Stay tuned!"}	[278, 237]	[518, 117]	1.034	Ah, I see a fun and whimsical scene! Here's what I notice...
Can you explain to me what is signals and system and then put the teddy bear on the moon	{"function": ["vlm_move('Put the teddy bear on the moon')"], "response": "Signals and systems? It's like understanding how Peppa Pig's voice gets to your ears! Now, let's send that teddy bear to the moon!"}	[316, 256]	[556, 274]	2.29	None
What can you see from the environment? After response, please put the red block on the dog	"function": ["vlm_vqa('What objects are visible in the environment?')", "vlm_move('Put the red block on the dog')"], "response": "I see a happy dog, a green plus sign, a red block, a smiley face, and a crescent moon. Red block is heading to the dog!"	[269, 310]	[537, 128]	0.9057	Ah, I see a fun little scene in front of me! Here's what I can observe...
1. What is a transistor? 2. What can you see from the environment? 3. Can you help me put the red block on the sun?	"function": ["vlm_vqa('What objects are visible in the environment?')", "vlm_move('Put the red block on the sun')"], "response": "1. A transistor is like a tiny switch for electricity! 2. I see a red block, a green plus sign, a happy dog, and a sun. 3. Let's aim high|red block to the sun!"	[252, 300]	[525, 101]	0.9876	Ah, I see a fun little scene here! Let me break it down for you...

Table 5: Representative interactions showing VLM planning, 2D grasp frames, and semantic scene understanding. **All pick-and-place manipulation success.** Detail VQA responses at 9.

5 Discussion

Our capstone project achieved three progressively advancing milestones: **Prototype-0** established a foundational ROS 2 framework for fixed-prompt pick-and-place; **Objective 1** advanced Vision-Language Models (VLMs) integration for natural, voice-based task planning; and **Objective 2** addressed manipulation of non-

uniform objects using bounding boxes, EfficientSAM segmentation, and PCA-based 2D frame estimation. Together, these advances yielded a modular, intelligent robotic system capable of executing open-vocabulary tasks with semantic understanding and geometric precision. Real-world testing showed grasp-and-place accuracy with errors as low as 0–1.11 cm, achieved using only RGB input. Beyond performance, the system's voice interface and visual feedback make it an accessible educational tool—suitable as a live demo for courses like ESE-3050, ESE-205, and ESE-446 to inspire student interest in AI-driven robotics.

Technically, our ROS 2 framework modularizes perception, reasoning, and control into independent nodes. The `conversation_node` processes user input via Qwen2.5-VL-32B-Instruct to produce structured plans, while `vlm_detection_node` handles open-vocabulary detection and 2D frame extraction using EfficientSAM and PCA. These components communicate via ROS 2 services and topics, enabling real-time control and extensibility across diverse hardware.

On hardware, the system executed multi-step pick-and-place tasks in response to user commands like "Put the red block on the green logo." When the object size exceeded 200 pixels, the Raspberry Pi invoked a laptop-based EfficientSAM API for segmentation. PCA then estimated object orientation, embedded in the `ObjectDetection` message, enabling frame-aware alignment. The framework abstracts away low-level commands, allowing flexible instructions like "the ear plug next to the logo," lowering the barrier for non-expert users.

The system's implementation journey surfaced key challenges. Learning ROS 2's publisher-subscriber model, message types, and timing logic demanded significant ramp-up time. Hand-eye calibration was also a limiting factor—small shifts in the taped-on camera impacted accuracy. Although affine calibration improved results, precision remained sensitive to pixel-level error. Most failures stemmed from such calibration drift.

Integrating VLMs also required model testing and API customization. Several candidates failed to return usable outputs. Through experimentation, we selected Qwen2.5-VL-32B-Instruct for conversation and Qwen2.5-VL-3B-Instruct for object localization. Using them required reverse-engineering I/O formats and carefully designing system prompts for JSON-based responses.

The development of Objective 2 involved several exploratory paths. Initial efforts to train a custom vision model on the Grasp Anything dataset—including a lightweight CNN and a fine-tuned ResNet-18—yielded weak geometric predictions. We then modified SAM to output confidence maps for frame axes, but training proved unstable and slow under limited resources. These challenges ultimately led to a more reliable and interpretable solution: prompting EfficientSAM for segmentation and applying PCA to extract object-centric frames. This shift demonstrated the practical benefits of combining classical geometric methods with pre-trained vision models for robotic perception.

Despite its capabilities, the system has limitations. The RGB-only camera limits depth perception and field of view, restricting task complexity and object variety. With no 3D data, all manipulation remains planar. Literature suggests eye-in-hand setups benefit from RGB-D cameras; thus, future versions could integrate depth sensors for 6D grasping and more robust calibration. Additionally, the system currently focuses on high-level planning. Further improvements could explore servo-level control, advanced robotics kinematics, or motion planning strategies like RRT.

Looking ahead, the system's core intelligence can scale to larger robot arms and more complex scenes. The voice-based interface makes it usable by anyone, regardless of technical background. While current tasks include object placement and pose estimation, more abstract commands—e.g., "clean the table" or "wash the dish"—require higher-level planning and symbolic reasoning, a natural next step for extending the framework's capabilities.

Critically, the project's feasibility is a reflection of recent AI advancements. Two years ago, building a voice-controlled robot assistant grounded in real-time image understanding would have been infeasible in a university setting. Today, public VLMs and open-source segmentation models make this vision practical. This capstone captures that shift, offering not just a working prototype, but a glimpse into the future of human-centered robotics.

In summary, this project offers a robust implementation and a compelling educational platform. It demonstrates how VLMs, combined with modular robotic frameworks, can transform how humans interact with machines—bringing intuitive, language-driven collaboration into real-world robotics.

6 Conclusion

This capstone project demonstrates a successful application of Electrical and Systems Engineering principles to the design and deployment of an intelligent, extensible robot assistant. By leveraging modular design in ROS 2, we developed a scalable framework that clearly separates perception, planning, and control—enabling seamless integration of vision, language, and actuation. Through practical application of kinematics, calibration, and control logic, we constructed a full pipeline capable of interpreting natural language commands and executing precise object manipulation tasks.

From foundational components like pick-and-place execution and affine hand-eye calibration to real-time visual feedback via onboard sensing, our system brings together multiple ESE domains into a unified and operational whole. The natural-language interface makes interaction intuitive for users of all backgrounds, while the system's modular architecture supports future extensions across hardware platforms and task domains.

Rather than emphasizing the complexity of modern AI models, this work highlights how thoughtful engineering design can incorporate advanced tools into robust, human-centered robotics. The resulting platform is more than a technical prototype—it reflects core values of electrical and systems engineering: modularity, adaptability, and real-world usability. We hope this project serves not only as a functional system but also as a student-led demonstration of how ESE knowledge can enable the next generation of intelligent machines.

7 Deliverables

The deliverables for this capstone project include an open-source robotics software framework, reproducible demonstrations, and supporting documentation to aid future learning and development. While some deliverables evolved during the semester based on project requirements and opportunities, all intended outcomes were successfully met or enhanced in final form.

Python and ROS 2-Based Framework

- A fully open-source ROS 2-based software framework implementing Prototype 0, Objective 1 (voice-integrated vision-language reasoning), and Objective 2 (2D frame prediction using EfficientSAM and PCA).
- A modular FastAPI server for EfficientSAM segmentation and voice interaction, supporting remote execution and integration with the main ROS 2 system.
- Python scripts for evaluating the 2D grasp frame pipeline, tested on the Cornell Grasp Dataset, with example outputs, qualitative results, and visualization.
- All code and documentation are publicly available on GitHub, including a clear README with setup instructions and usage guidelines for future students and instructors.

Documentation and Project Website

- Initially, a standalone progress document was planned to capture insights and guide future students. In practice, the weekly progress reports fulfilled this role.
- In addition, the open-source repository (with detailed README) and the project website now collectively provide comprehensive documentation, implementation details, and reflections on the development process.
- These resources are designed to support reproducibility and make the system accessible to new students, faculty, or future capstone teams.

Poster and Demonstration Materials

- Originally, demonstration videos were planned as the primary presentation medium. As the project progressed, I developed a formal poster and was happy to have the opportunity to present it at the ESE Day event.
- The final project website features this poster, along with embedded demonstration videos and an interactive explanation of the system pipeline.
- Compared to a standalone video, the website offers a more effective and engaging platform for communicating technical workflows and results to both academic and general audiences.

All project materials are publicly available:

- **Project Website:** `https://sites.wustl.edu/fromlanguagetoaction/`
- **GitHub Repository:** `https://github.com/yuzihaowashu/ESE-498_From_Language_To_Action`

8 Schedule & Timeline

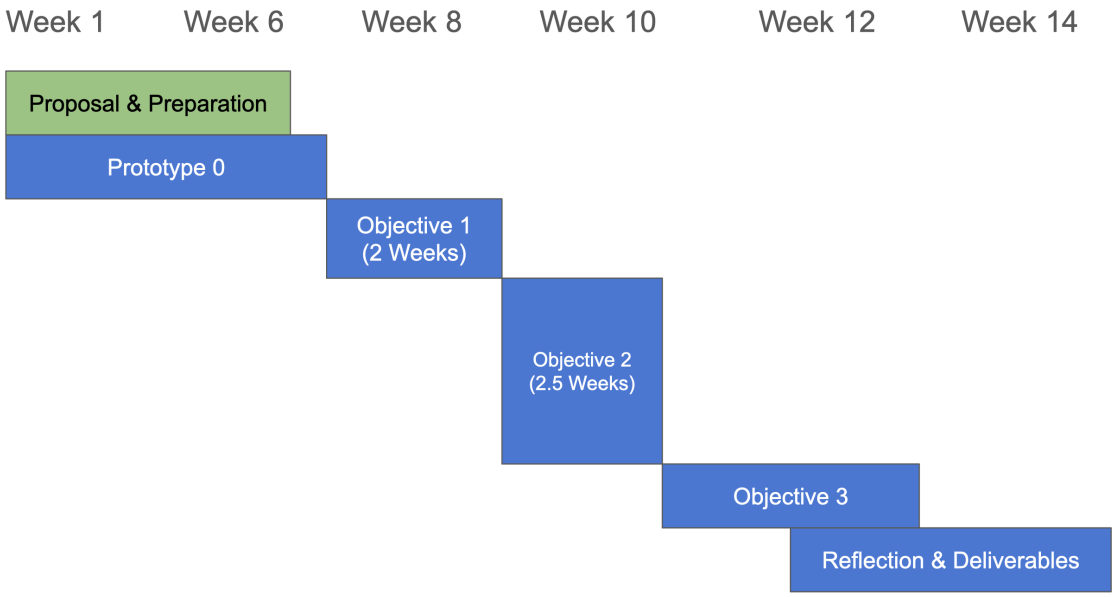


Figure 15: Initial Project Schedule.

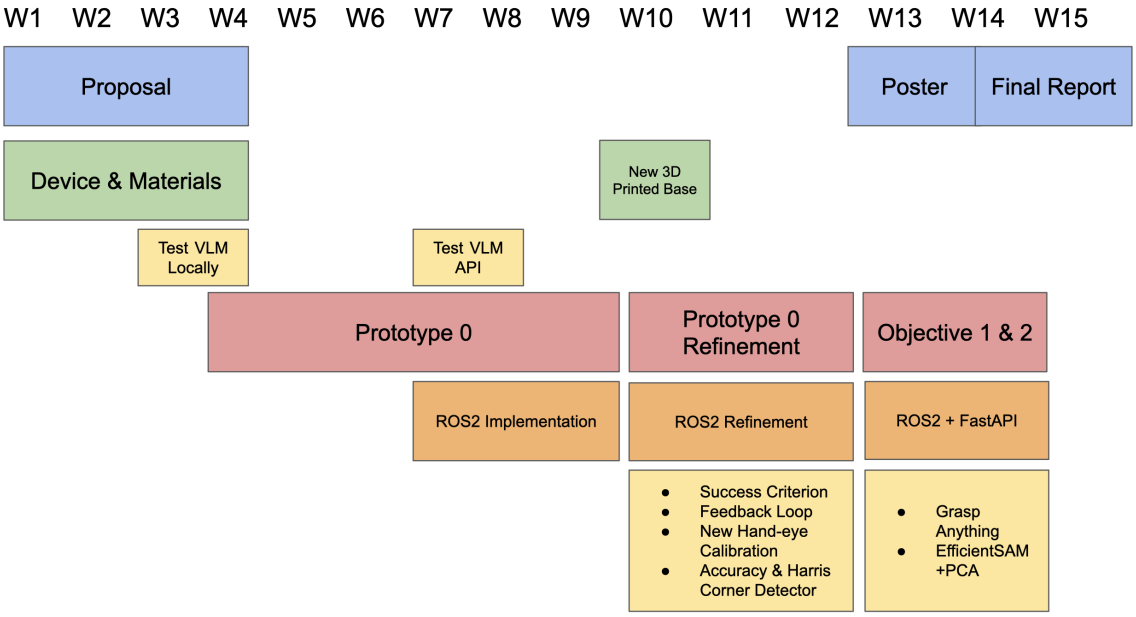


Figure 16: Actual Project Timeline.

The original timeline (Figure 15) presented an overview of the expected progression for key phases of the project, including proposal preparation, prototype development, and completion of major objectives. At the time of planning, some methodologies—particularly those for perception and interaction—were not fully established, which led to an optimistic projection.

As shown in Figure 16, the actual timeline shifted significantly. A substantial amount of time was invested in building and refining Prototype 0, which served as the core system integrating ROS 2 control, natural language understanding, and perception. Although initially positioned as an early-stage milestone, Prototype 0 ultimately represented the bulk of the project's complexity. Objectives 1 and 2 were layered extensions built upon this foundation. Accuracy issues, calibration challenges, and architectural rework led to necessary refinements that extended beyond the initial plan. Overall, while the project met its major deliverables, the development cycle was more iterative and nonlinear than expected.

While the original timeline also included Objective 3—focused on extending the system to support 3D manipulation—this goal was ultimately reframed as future work. The motivation behind Objective 3 was strong; however, as development progressed, it became clear that the current hardware setup, particularly the use of a monocular RGB camera in an eye-in-hand configuration, could not reliably provide depth information. Without accurate spatial perception, true 3D manipulation was infeasible within the constraints of the current system.

Instead of pursuing an underdefined method, the project refocused on building a robust and accurate 2D robot assistant using the available hardware. This shift allowed for more meaningful progress and refinement in visual grounding, calibration, and voice-driven interaction. Moving forward, future iterations of the system will explore 3D manipulation using more suitable sensors, such as an RGB-D camera, integrated into the compact framework developed in this capstone.

References

[1] J. J. Craig, *Introduction to Robotics: Mechanics and Control*. Upper Saddle River, NJ: Pearson Prentice Hall, 3rd ed., 2005.

[2] Author(s), "Recent advances and challenges in industrial robotics," *Processes*, vol. 13, no. 3, p. 832, 2023.

[3] "ROS 2 - robot operating system." `https://docs.ros.org/en/ros2`. Accessed: 2025-05-04.

[4] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang, H. Zhong, Y. Zhu, M. Yang, Z. Li, J. Wan, P. Wang, W. Ding, Z. Fu, Y. Xu, J. Ye, X. Zhang, T. Xie, Z. Cheng, H. Zhang, Z. Yang, H. Xu, and J. Lin, "Qwen2.5-vl technical report," *arXiv preprint arXiv:2502.13923*, 2025.

[5] Adafruit Industries, "Adafruit ultra miniature camera - ov5640 5mp with i2c and spi," 2024. Accessed: 2025-05-05.

[6] Elephant Robotics, "Elephant robotics official website," 2025. Accessed: 2025-05-05.

[7] E. Robotics, "pymycobot: Python api for mycobot serial communication." `https://github.com/elephantrobotics/pymycobot`, 2025. Accessed: 2025-05-04.

[8] J. Zhang, J. Huang, S. Jin, and S. Lu, "Vision-language models for vision tasks: A survey," *arXiv preprint arXiv:2304.00685*, 2023.

[9] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[10] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[11] T. Bray, "The javascript object notation (json) data interchange format." `https://www.rfc-editor.org/rfc/rfc8259.html`, 2017. RFC 8259, Internet Engineering Task Force (IETF).

[12] X. Liu, J. Wang, J. Sun, X. Yuan, G. Dong, P. Di, W. Wang, and D. Wang, "Prompting frameworks for large language models: A survey," 2023.

[13] J. Bai, S. Bai, S. Yang, S. Wang, S. Tan, P. Wang, J. Lin, C. Zhou, and J. Zhou, "Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond," 2023.

[14] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten,

L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, "The llama 3 herd of models," 2024.

[15] OpenAI, "Openai python api package." `https://github.com/openai/openai-python`, 2024. Version 1.23.6, Accessed: 2025-05-04.

[16] Alibaba Cloud, "Qwen: Large language model service." `https://www.alibabacloud.com/en/`

solutions/generative-ai/qwen, 2025. Accessed: 2025-05-04.

[17] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[18] S. J. D. Prince, *Computer Vision: Models, Learning, and Inference*. Cambridge, UK: Cambridge University Press, 2012.

[19] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the Alvey Vision Conference*, (Manchester, UK), pp. 147–151, 1988.

[20] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu, B. Chang, X. Sun, L. Li, and Z. Sui, "A survey on in-context learning," 2024.

[21] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[22] A. Zhang, "Speech recognition (version 3.14.2) [software]." https://github.com/Uberi/speech_recognition, 2025. Accessed: 2025-05-04.

[23] J. Gardiner and Contributors, "pyttsx3: Text-to-speech conversion library in python." https://github.com/nateshmbhat/pyttsx3, 2020. Accessed: 2025-05-04.

[24] A. D. Vuong, M. N. Vu, H. Le, B. Huang, B. Huynh, T. Vo, A. Kugi, and A. Nguyen, "Grasp-anything: Large-scale grasp dataset from foundation models," 2023.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[26] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[28] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[29] Y. Xiong, B. Varadarajan, L. Wu, X. Xiang, F. Xiao, C. Zhu, X. Dai, D. Wang, F. Sun, F. Iandola, R. Krishnamoorthi, and V. Chandra, "Efficientsam: Leveraged masked image pretraining for efficient segment anything," 2023.

[30] G. Bradski, "The opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.

[31] OneOneLiu, "Cornell grasp dataset on kaggle." https://www.kaggle.com/datasets/oneoneliu/cornell-grasp, 2024. Accessed: 2025-05-05.

[32] H. Liu, C. Li, Y. Li, and Y. J. Lee, "Improved baselines with visual instruction tuning," 2024.

9 Appendix

Appendix A: Robotics Kinematics & Materials and Tools

Forward Kinematics

Each joint in the robot is described using the Denavit-Hartenberg (DH) convention with four parameters:

- a_i: Link length (distance along x_i-axis).
- α_i: Link twist (rotation around x_i-axis).
- d_i: Link offset (distance along z_i-axis).
- θ_i: Joint angle (rotation about z_i-axis, variable for revolute joints).

The transformation matrix from frame $i-1$ to i is given by:

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The total transformation from the base to the end-effector is:

$$T_6^0 = T_1^0 \cdot T_2^1 \cdot T_3^2 \cdot T_4^3 \cdot T_5^4 \cdot T_6^5$$

where T_6^0 contains:

- The position (x, y, z) in the first three elements of the last column.
- The orientation as a 3×3 rotation matrix.

Thus, forward kinematics is represented as:

$$\mathbf{x} = f(\mathbf{q})$$

where $\mathbf{x} = (x, y, z, R, P, Y)$ (end-effector pose) and $\mathbf{q} = (\theta_1, \theta_2, ..., \theta_6)$ (joint variables).

Appendix B: Object Recognition and Semantic Understanding with Vision Language Models

Model Selection and Motivation

To enable both robust object detection and rich conversational interaction, this project leverages two variants of the Qwen2.5-VL family of Vision-Language Models (VLMs), developed by Alibaba Group. The Qwen2.5-VL series has achieved state-of-the-art performance on numerous vision-language benchmarks, including SEED-Bench, MMBench, and MME, indicating strong capabilities in perception, grounding, and reasoning.

We use **Qwen2.5-VL-32B-Instruct** for high-level dialogue and semantic understanding in the `conversation_node`, where it interprets nuanced commands, tracks context across user inputs, and generates humorous or helpful replies. For image-grounded object recognition, we use the lighter-weight yet still powerful **Qwen2.5-VL-3B-Instruct** in the `vlm_detection_node`, which is sufficient for accurate bounding box extraction in real-world tabletop scenes.

Qualitative Comparison: 32B vs 3B Output on Same Prompt

To illustrate the difference in expressive capability and environmental perception, we compare the outputs of the 32B and 3B models (i.e., models with 32 billion and 3 billion parameters) on the same image and user prompt. In this case, no system prompt was provided—only a user-level instruction (`"What are there in the environment?"`). The image depicts the entrance to Washington University in St. Louis, with autumn foliage, campus buildings, and signage.

The Qwen2.5-VL-3B-Instruct model returns a concise list of major scene components, such as the university sign, buildings, trees, and a pathway, providing a valid but minimal interpretation of the image. In contrast, the Qwen2.5-VL-32B-Instruct model offers a significantly more structured and informative description. It not only captures the same high-level components but also organizes them into labeled categories with detailed natural language descriptions. The larger model infers additional scene-level context, such as the presence of a flagpole, weather conditions (clear sky), and seasonal cues (autumn foliage), and even speculates on the functional role of certain elements (e.g., pathways used by students and visitors).

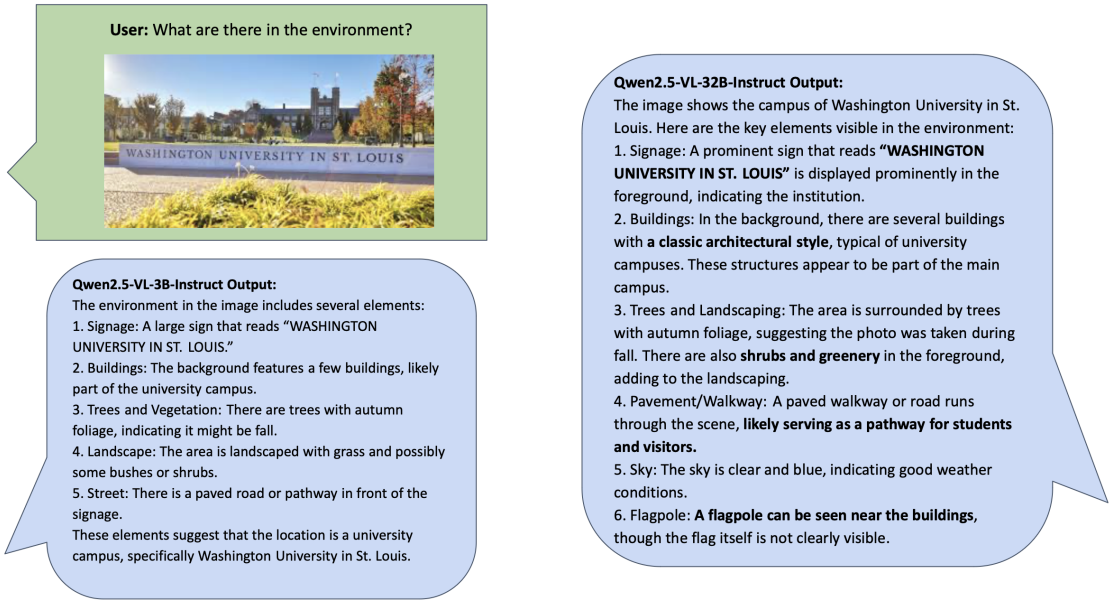


Figure 18: Input image used for the qualitative comparison between Qwen2.5-VL-32B-Instruct and Qwen2.5-VL-3B-Instruct models on the prompt: *"What are there in the environment?"* The models were evaluated based on their descriptive output without any system prompt. The responses encourages us to integrate Qwen2.5-VL-32B-Instruct as the conversation interface and Qwen2.5-VL-3B-Instruct as the object detection module.

This qualitative difference reveals how larger multimodal models like the 32B variant exhibit stronger spatial reasoning, scene understanding, and natural language fluency. The enhanced description from the 32B model better supports downstream robotic decision-making, reinforcing its suitability as the core module for the dialogue-based controller in our robotic system. Meanwhile, the 3B model remains sufficient for object recognition tasks that require less contextual reasoning but still benefit from flexible and open-vocabulary grounding. These complementary strengths motivated our architectural decision to deploy the 32B model in the `conversation_node` and the 3B model in the `vlm_detection_node`.

Generalization and Semantics Beyond Traditional Object Detectors

Unlike traditional object detection models such as YOLO, which are trained with fixed object categories (e.g., car, person, bicycle), Vision-Language Models generalize to open-vocabulary object detection. In

our experiments, `Qwen2.5-VL-3B-Instruct` was able to correctly localize and extract bounding boxes for objects like *"ear plug"*, *"orange"*, *"pen"*, or *"plastic block"*, which are not typically covered in YOLO's label space.

Moreover, the VLM supports semantic variation through natural language: the same object could be queried as *"red block"*, *"plastic block"*, or *"red plastic block"*, and the VLM consistently returns accurate results. This flexibility arises from the VLM's grounding of vision and text through multimodal attention rather than relying on fixed object detection heads. In contrast, YOLO's object classes are encoded in the final output layer and lack adaptability to unseen or compositional language inputs.

Appendix C: Detailed VQA and Planning Examples

Listing 3: Example 1: User query and system response with vlm_move()

```
User: Can you put the teddy bear on the dog?

API Response:
{
  "function": ["vlm_move('Put the teddy bear on the dog')"],
  "response": "Sure thing! Teddy bear's going for a ride on the dog!"
}

Start Coords: [245, 276]
End Coords: [259, 119]
PCA Angle (radians): 0.8525
VQA Response: None
```

Listing 4: Example 2: User query and system response with VQA

```
User: Can you tell me what can you see and then put the teddy bear on the dog?

API Response:
{
  "function": ["vlm_vqa('What objects are visible in the image?')", "vlm_move('Put
  the teddy bear on the dog')"],
  "response": "Sure thing! Let me take a look and then place that teddy bear on the
  dog. Stay tuned!"
}

Start Coords: [278, 237]
End Coords: [518, 117]
PCA Angle (radians): 1.034
VQA Response: Ah, I see a fun and whimsical scene! Here's what I notice in the
environment:

1. **A smiling face**: There's a simple yellow smiley face with two dots for eyes
and a curved line for a mouth. It looks cheerful!

2. **A lemon-shaped object**: Next to the smiley face is an object that resembles a
lemon. It has a textured surface with holes, like a mesh or net, and a small green
leaf on top, giving it a playful, cartoonish look.

3. **A dog**: To the right of the lemon, there's a cute golden retriever sitting
calmly. The dog looks happy and relaxed.

4. **A crescent moon**: In the bottom-right corner, there's a simple black crescent
moon drawing, adding to the whimsical feel of the image.

The background is plain white, which makes all the elements stand out clearly. This
setup feels lighthearted and creativeperfect for a fun chat or playful task!

What would you like me to do next?
```

Listing 5: Example 3: User query and system response of vlm_move()

```
User: Can you explain to me what is signals and system and then put the teddy bear
on the moon?

API Response:
{
  "function": ["vlm_move('Put the teddy bear on the moon')"],
  "response": "Signals and systems? It's like understanding how Peppa  P i g s  voice
  gets to your ears! Now,  l e t s  send that teddy bear to the moon!"
}

Start Coords: [316, 256]
End Coords: [556, 274]
PCA Angle (radians): 2.29
VQA Response: None
```

Listing 6: Example 4: User query and system response with VQA

```
User: What can you see from the environment? After response, please put the red
block on the dog

API Response:
{
  "function": ["vlm_vqa('What objects are visible in the environment?')", "vlm_move
  ('Put the red block on the dog')"],
  "response": "I see a happy dog, a green plus sign, a red block, a smiley face, and
   a crescent moon. Red block is heading to the dog!"
}

Start Coords: [269, 310]
End Coords: [537, 128]
PCA Angle (radians): 0.9057
VQA Response: Ah, I see a fun little scene in front of me! Here's what I can observe
:

1. **A smiling face**: There's a cheerful yellow smiley face with simple
f e a t u r e s two  dots for eyes and a curved line for a mouth.
2. **A dog**: A cute golden retriever or similar breed is sitting calmly on the
right side. It looks friendly and relaxed!
3. **A green plus sign**: A bright green ""+"" symbol is positioned near the center.
4. **A red object**: On the left,  t h e r e s  a red item that looks like part of a
tool or  t o y it  could be a handle or grip.
5. **A crescent moon**: A black crescent moon shape is on the far right, adding a
whimsical touch.

This setup feels playful and creative, almost like a child's drawing or an
educational activity. What would you like me to do with these objects?
```

Listing 7: Example 5: User query and system response with VQA

```
User: 1. What is a transistor? 2. What can you see from the environment? 3. Can you
help me put the red block on the sun?

API Response:
{
  "function": ["vlm_vqa('What objects are visible in the environment?')", "vlm_move
  ('Put the red block on the sun')"],
  "response": "1. A transistor is like a tiny switch for electricity! 2. I see a red
   block, a green plus sign, a happy dog, and a sun. 3.  L e t s  aim high-red block
  to the sun!"
}

Start Coords: [252, 300]
End Coords: [525, 101]
PCA Angle (radians): 0.9876
VQA Response: API reponse: Ah, I see a fun little scene here! Let me break it down
for you:

1. **Red Object**: There's a red object on the left side of the image. It looks like
```

```
 a tool or part of a device, possibly related to my robotic arm.

2. **Green Plus Sign**: Above and to the left, there's a green plus sign (+).

3. **Sun Symbol**: On the right side, there's an orange sun symbol with rays
extending outward.

4. **Dog Image**: Below the sun, there's a cute cartoon-style dog sitting and
looking happy.

5. **Equal Sign**: In the middle, there's a white equal sign (=), suggesting some
kind of equation or relationship between the elements.

It seems like this setup might be part of a playful math problem or a creative
puzzle! What do you want me to do with these objects?
```

Appendix D: Infeasibility of Training Vision-Language Models and Appreciation for Open Access APIs

While this project integrates a Vision-Language Model (VLM) for robotic perception and instruction following, it is important to clarify that training such a model from scratch was beyond the scope of our work. **Modern VLMs, such as Qwen2.5-VL, are trained on massive multimodal datasets and require thousands of GPUs and millions of dollars in resources, making them impractical for academic projects.** Even relatively lightweight VLMs contain billions of parameters, each stored as 16-bit or 32-bit floating-point numbers, resulting in models that consume tens or hundreds of gigabytes of memory [14]. State-of-the-art commercial VLMs, such as the one powering ChatGPT, are estimated to exceed *trillions* of parameters, placing them firmly in the domain of industrial-scale computing.

VLMs go far beyond traditional small-scale machine learning models such as logistic regression or random forests. They must learn and reason over complex linguistic structures, semantics, visual spatial grounding, and multi-turn dialogue. In essence, VLMs must understand language—arguably the most information-dense modality humans use—which demands capabilities like semantic parsing, coreference resolution, and instruction disambiguation. These are not tasks that can be solved with classical ML methods or even standard neural networks without significant architectural innovation and training scale.

Instead of building such models from scratch, we build upon open-source pretrained VLMs, focusing our contributions on how these powerful tools can be effectively integrated into real-world robotic systems. Open access APIs provided by research organizations and companies allow projects like ours to benefit from cutting-edge AI without the prohibitive costs of model development or infrastructure management.

Unlike traditional fixed-purpose computer vision models, VLMs are highly dynamic: their behavior adapts to user instructions through prompt design, without requiring any changes to model weights. In our system, we applied strategic prompt design and few-shot in-context learning, carefully structuring input examples to guide the VLM's behavior during inference. One reason this approach is effective is that the underlying VLM (Qwen2.5-VL) is instruction-tuned after pretraining, making it naturally aligned with user-guided, task-specific workflows.

Other adaptation methods, such as Retrieval-Augmented Generation (RAG) [17] and fine-tuning [32] for custom tasks, were considered but determined to be neither feasible nor necessary for this project. RAG requires maintaining a large knowledge base and retrieval mechanism at runtime, adding infrastructure complexity unsuited to lightweight, real-time robotic systems. Fine-tuning, while powerful, requires GPU clusters, careful gradient tuning, and significant domain-specific data. Moreover, because our robotic task was carefully framed through prompt design to focus on instruction parsing and spatial grounding, direct prompting of the instruction-tuned VLM was both sufficient and effective.

Our goal was not to reinvent deep learning models, but to demonstrate how emerging AI technologies can be thoughtfully applied to make robotic manipulation more intuitive, accessible, and educational.

Appendix E: Additional 2D Frame Labeling Results

The following figures show qualitative results of 2D object-centric frame prediction tested on the **Cornell Grasp Dataset** [31]. Although the Cornell dataset is originally designed for 3D grasp detection, our 2D frame labeling approach demonstrates qualitatively successful results for top-down pick-and-place tasks. The proposed pipeline integrates object segmentation using EfficientSAM with PCA-based axis estimation to produce object-aligned 2D grasp frames. In most cases, the predicted frames align well with the geometry of non-uniform objects.

Failures, where grasp axes appear unreasonable, are primarily attributed to inaccurate bounding boxes produced by the Vision-Language Model (VLM). When the target object is visually similar to its background or partially occluded, the VLM often fails to isolate the correct region, which leads to incorrect segmentation and, subsequently, flawed frame estimation. Despite this limitation, the overall approach remains robust and interpretable for the majority of samples evaluated.

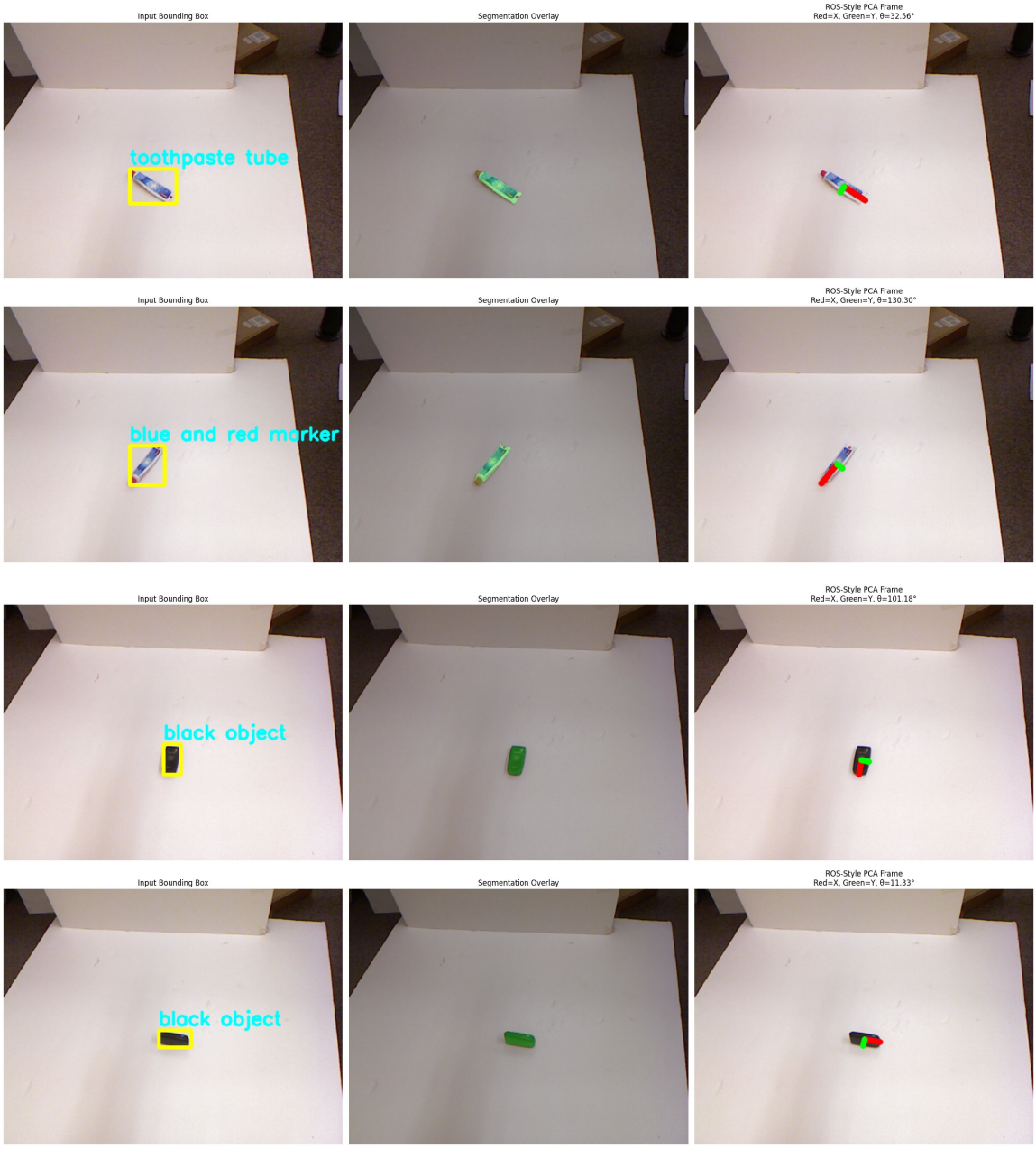


Figure 19: 2D frame predictions on samples `pcd0201r`, `pcd0202r`, `pcd0300r`, and `pcd0302r` from the Cornell Grasp Dataset [31].

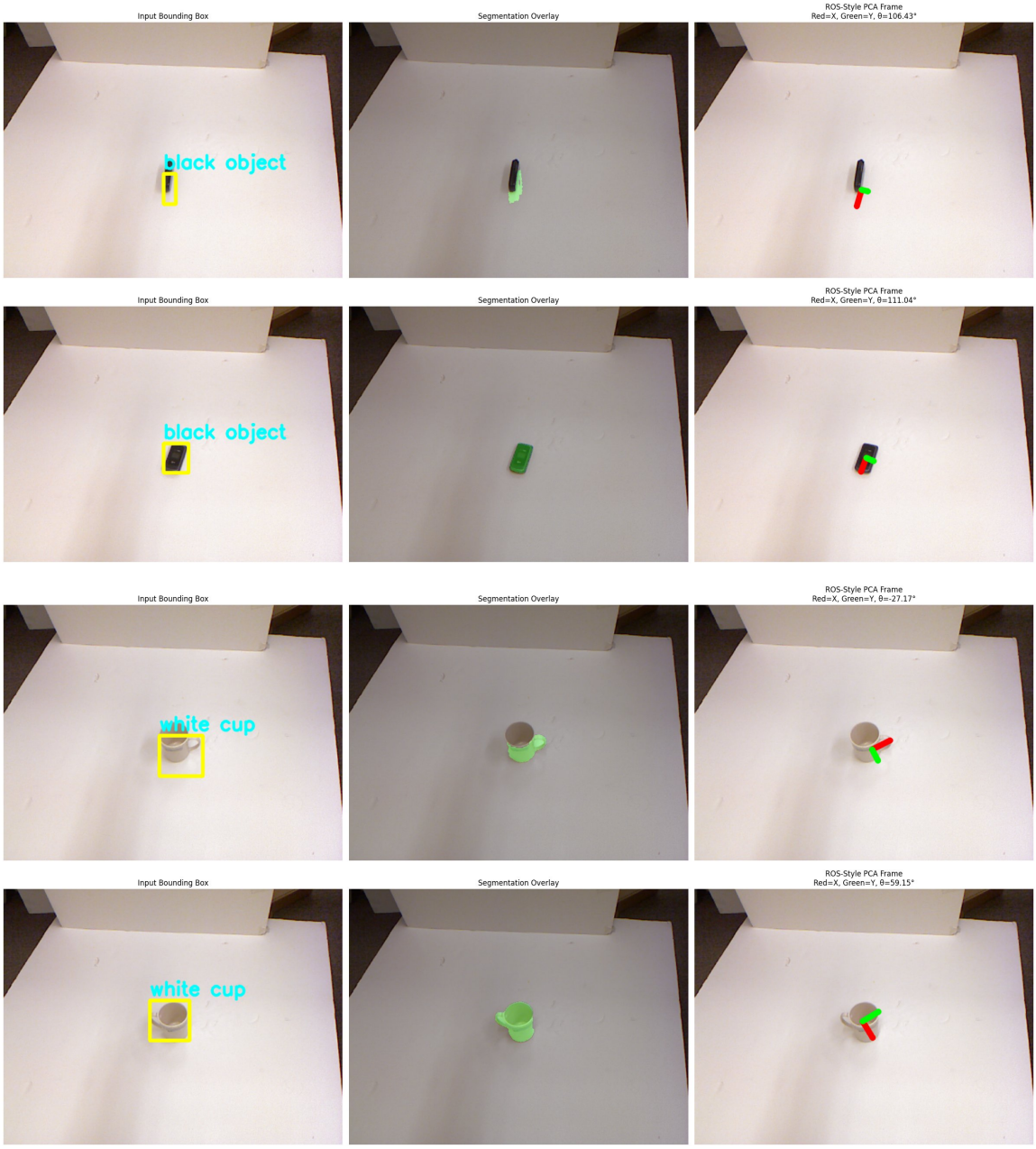


Figure 20: 2D frame predictions on samples `pcd0303r`, `pcd0304r`, `pcd0320r`, and `pcd0321r`.

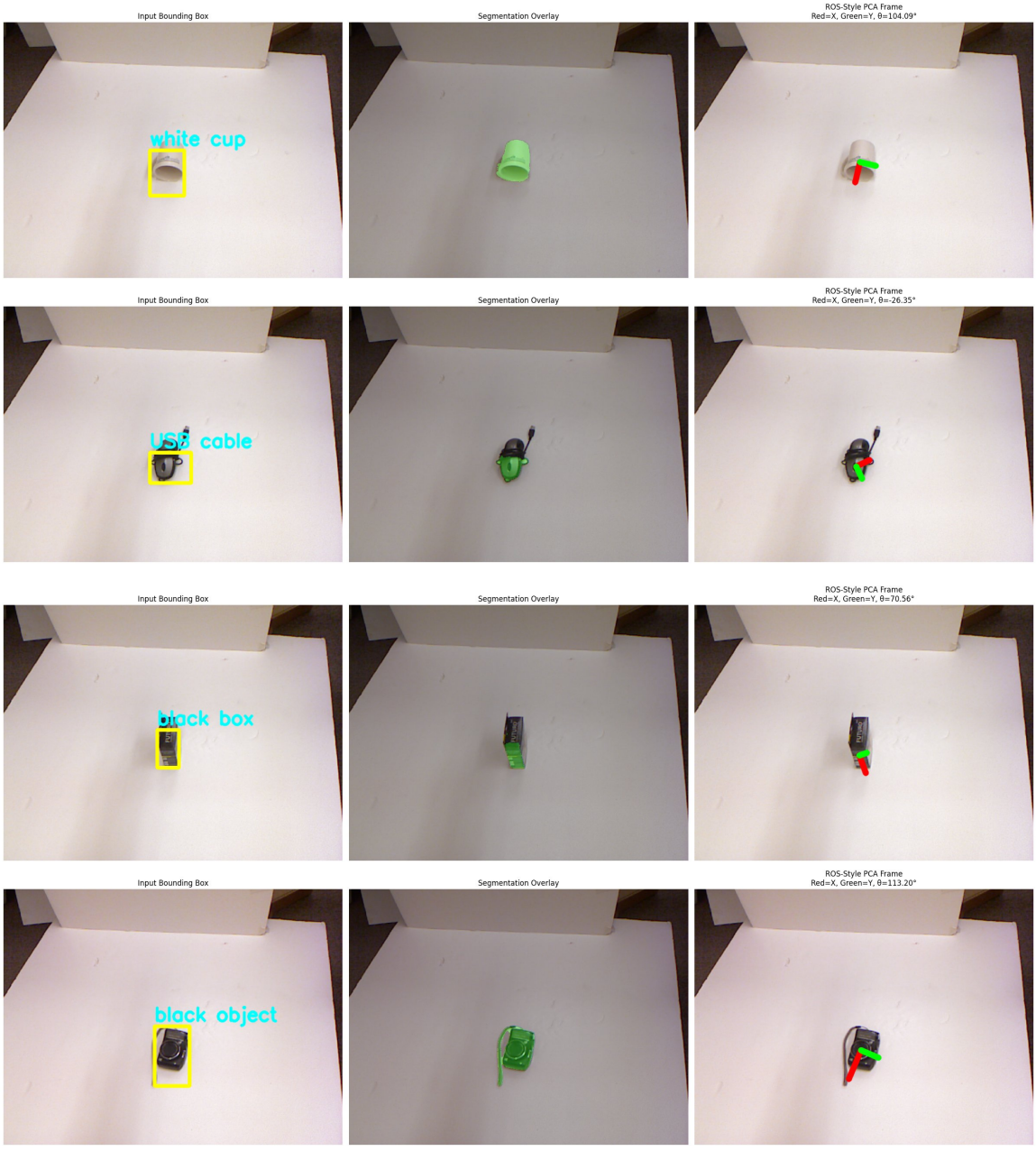


Figure 21: 2D frame predictions on samples `pcd0322r`, `pcd0518r`, `pcd0530r`, and `pcd0544r`.

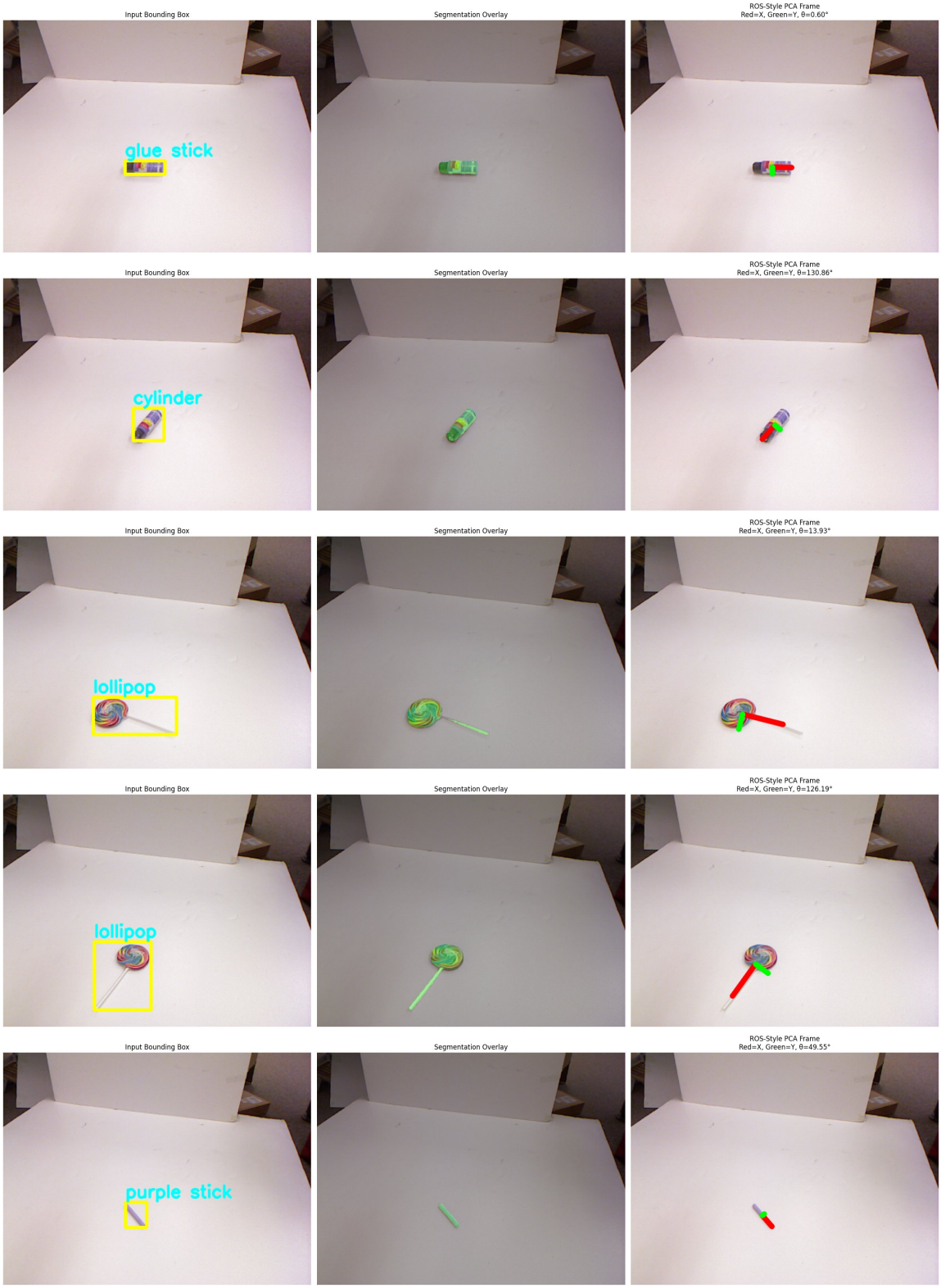


Figure 22: 2D frame predictions on samples `pcd0900r`, `pcd0901r`, `pcd1001r`, and `pcd1003r`.

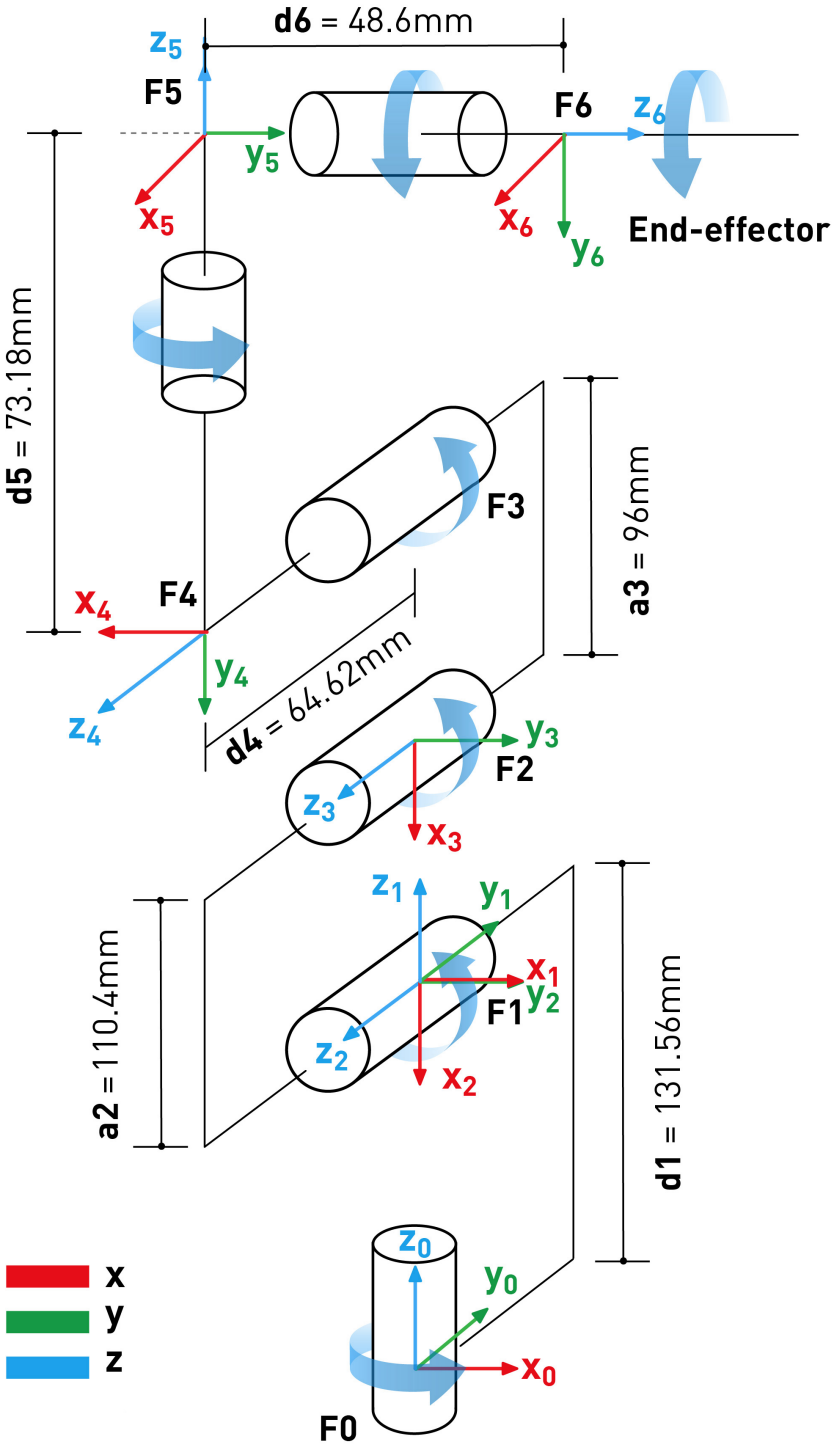


Figure 17: The schematic of myCobot 280 used in this capstone.