Text2Arm: Natural Language & Computer Vision-Driven Robotic Manipulation

Alaa Khamis

January 2025

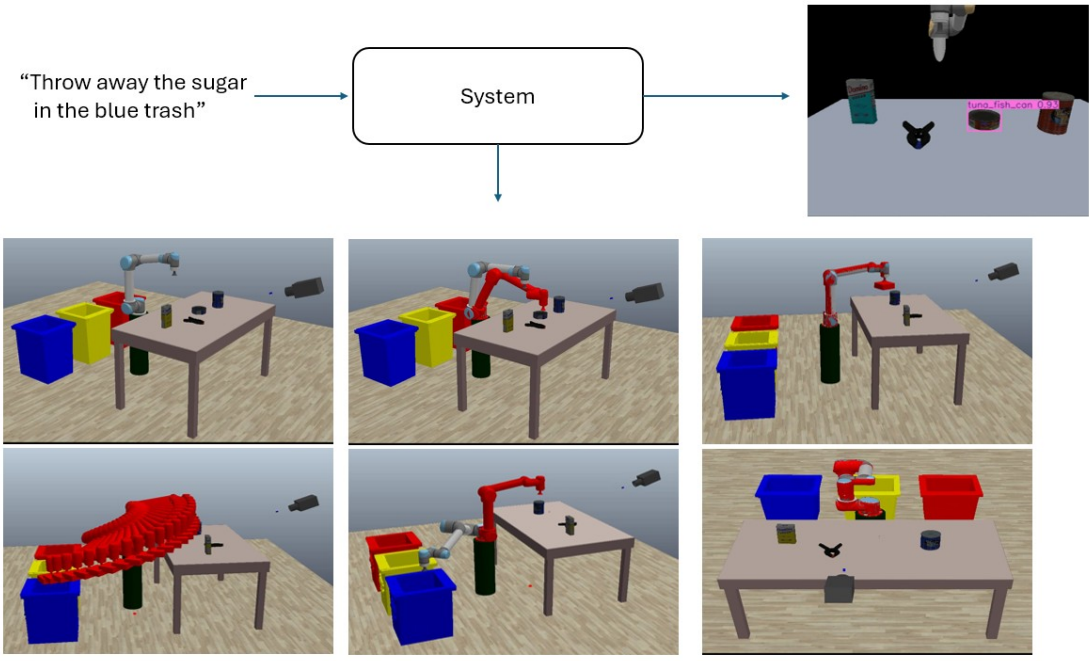

Figure 1: Full pipeline from prompt to execution

Abstract

This paper presents an integrated robotic manipulation pipeline that uses a 6-degree-of-freedom (DOF) robotic arm (UR5), a fine-tuned Large Language Model (LLM), and an RGB-D camera for object detection via YOLOv8. Users control the robotic arm through natural language prompts, which are processed by the LLM to generate task-specific commands. These commands are then translated into object-location tuples and passed to a detection module

that identifies items and points of interest. The robotic arm then executes a pick-and-place task in a simulated environment using CoppeliaSim. Our approach is designed to be robust, adaptable, and deployable on lightweight hardware.

The proposed system demonstrates high accuracy and efficiency while maintaining minimal computational requirements. We validate our approach using a simulated environment and discuss its real-world deployment potential. Future work includes optimizing real-world deployment and exploring alternative control systems.

The code for this project can be accessed at https://github.com/alaa-khamis/Text2Arm.

1 Introduction

Robotic arms have been widely used in industrial automation for tasks such as assembly, welding, and packaging. The advent of artificial intelligence (AI), particularly large language models (LLMs) and multi-modal neural networks, has enabled more intuitive and intelligent control of robotic systems. Major companies are now focusing on developing multi-purpose robots for both industrial and domestic applications, yet these solutions often require expensive hardware and complex integration [1, 2, 3].

This project introduces a cost-effective, natural language-driven robotic manipulation system that enables users to command a 6-DOF robotic arm using intuitive prompts. By using a fine-tuned LLM with an RGB-D camera and YOLOv8 [4] for object detection, we provide a streamlined pipeline that translates human intent into robotic actions. Our solution aims to make robotic systems more accessible and adaptable while minimizing hardware and usage complexity.

The key contributions of this work include:

- Development of a fine-tuned LLM for command interpretation, reducing dependency on large-scale cloud-based models.
- Integration of natural language processing with real-time object detection for effective robotic control.
- Implementation of a computationally efficient pipeline that runs on lightweight hardware.

This paper is structured as follows:

- Section 2 describes the simulation environment used for robotic control.
- Section 3 details the LLM training and fine-tuning process.
- Section 4 discusses object detection using YOLOv8.
- Section 5 shows some issues we ran into using CoppeliaSim and the solutions we came up with.
- Section 6 presents the robotic manipulation pipeline, followed by a conclusions in Section 7.

2 CoppeliaSim

Robotic simulations are used in research and industry to validate robotic control algorithms before deployment in real-world environments. Popular simulators such as Gazebo, MuJoCo, and CoppeliaSim provide realistic physics-based environments for testing robotic systems. Many robotic frameworks rely on the Robot Operating System (ROS) combined with MoveIt for motion planning and control.

Initially, we wanted to use ROS, Gazebo, and MoveIt as our framework, since these tools are well-established for simulating robotic systems and support transition real-world implementation. However, running these programs on Windows was be unstable, with one compatibility issue often leading to another. Even when using the Windows Subsystem for Linux (WSL) with an Ubuntu distribution, the problems persisted. As a result, we opted for CoppeliaSim (previously known as V-Rep) because its native Windows support and robust feature set.

CoppeliaSim offers several advantages that align with our project requirements:

- It supports the Open Motion Planning Library (OMPL)[5], for efficient motion planning of robotic manipulators with it's simOMPL plugin.
- It provides built-in inverse kinematics (IK) solvers for the UR5 robotic arm, which simplifies trajectory planning with the simIk plugin.
- It includes support for cameras and sensors for vision tasks and collision checks.
- The ZeroMQ API enables communication with Python, allowing us to build the entire control system in Python while interacting with the simulation asynchronously.

3 Large Language Model (LLM)

3.1 Model Selection

Current work in both research and industry revolves around using SOTA LLMs or VLMs (Vision-Language Models) to process the user's command. While these models such as GPT-4 and LLaMA offer exceptional performance, their cost of training, fine-tuning and inference is very high, not to mention they require high-end GPUs with considerable memory resources. Even using hosted solutions through API's like OpenAI's can incur significant cost for consistent usage, making them impractical for small-scale systems.

To balance performance and efficiency, we selected Google's **Flan-T5-Base** model as the foundation model for fine-tuning. Flan-T5-Base has 250 million parameters, requires only 1.1 GB of VRAM for inference, and is optimized for instruction-following tasks [6]. This makes it well-suited for generating structured outputs for robotic control in resource-constrained environments.

3.2 Fine-Tuning

To fine-tune the LLM for robotic control, we created a task-specific dataset that includes **500 unique command prompts** mapped to structured outputs. The dataset generation ensures variety in commands from items, locations and format:

- **Single-task commands (60%)**: e.g., `"Move the sugar box to the red bin"` → `[('sugar_box', 'redBin')]`.
- **Two-task commands (30%)**: e.g., `"Move the tuna can to the red bin and the clamp to the blue bin"` → `[('tune_fish_can', 'redBin'), ('large_clamp','blueBin')]`.
- **Three-task commands (10%)**, To ensure the ability to process compound commands.

We fine-tuned the model using a google-colab notebook running on the free T4 GPU, with a **90% training** & **10% evaluation** split. The training took approximately 2 minutes. The notebook used for training heavily inspired by my friend Rami Simaan's (Also taking the lab this semester) code from his deep learning course, can be found here.

Training Details:

- **Batch size**: 8
- **Learning rate**: 2e-5
- **Epochs**: 5
- **Optimizer**: AdamW
- **Loss function**: Cross-entropy

3.3 Results

After fine-tuning, we evaluated the model on unseen prompts to measure its ability to extract structured outputs. We compared our fine-tuned Flan-T5 model against:

- The **base Flan-T5 model** (without fine-tuning).
- **LLaMA-2-7B**, a significantly larger model, for a cost-performance comparison.

Table 1 presents the performance of each model. Our fine-tuned Flan-T5 consistently outperformed the base model and even provided more structured outputs than the much larger LLaMA-2-7B. Inference of the base model (and our fine-tuned) was ¡ 1 second, and for the LLaMA-2-7B model was approx. 10 seconds while using 14 GB of VRAM.

* I also recently came across this Linkedin post by a director at Meta for fine-tuning smaller LLMs which I found to be interesting and relevant to our use case: Linkedin Post

Prompt	Ours	Base Model	LLaMA-2 7B
Place the clamp into the yellow can.	[('large_clamp', 'yellowBin')]	target location	Sure! Here are the ('item', 'target location') pairs for the given sentence: Clamp - Yellow can
Take the master chef can to the blue bin.	[('master_chef_can', 'blueBin')]	blue bin	Sure! Here are the ('item', 'target location') pairs extracted from your input: Item: Master Chef Can Target Location: Blue Bin
Set the sugar box by the red bin. after, do large clamp close to the blue bin.	[('sugar_box', 'redBin'), ('large_clamp', 'blueBin')]	blue bin	Sure! Here are the ('item', 'target location') pairs extracted from your input: 1. Sugar box - near the red bin 2. Large clamp - near the blue bin

Table 1: Comparison of Model Outputs for Three Prompts

4 Object Detection

4.1 YOLO

After the user enters their prompt, our system verifies if the specified item exists withing our visual scene. This is done by using a YOLOv8 model that has been fine-tuned to detect objects from the YCB benchmarks dataset[7, 8]. To integrate this into our simulation, we used 3D scans of the objects from the following Github repository YCB Tools. From these items, we used 4 - tuna_fish_can, master_chef_can, sugar_box and large_clamp.

The choice for using this specific setup:

1. **Real Objects:** The items in the YCB dataset are real-life objects that can be found at home.
2. **Proven Performance:** The YOLOv8 model is a robust and well-known model with a diverse range of applications, documentation and usages in real-life scenarios.
3. **Efficient:** The specific YOLOv8 model we used is a fine-tune of the YOLOv8x model which is the largest sub-model, yet, it only uses 0.5gb of vram and only second for inference and detection.

For these reasons, we can see that the choice of this model and way of integration with most applications (specifically using the Ultralytics library) offers a good solution for quick and accurate object detection.

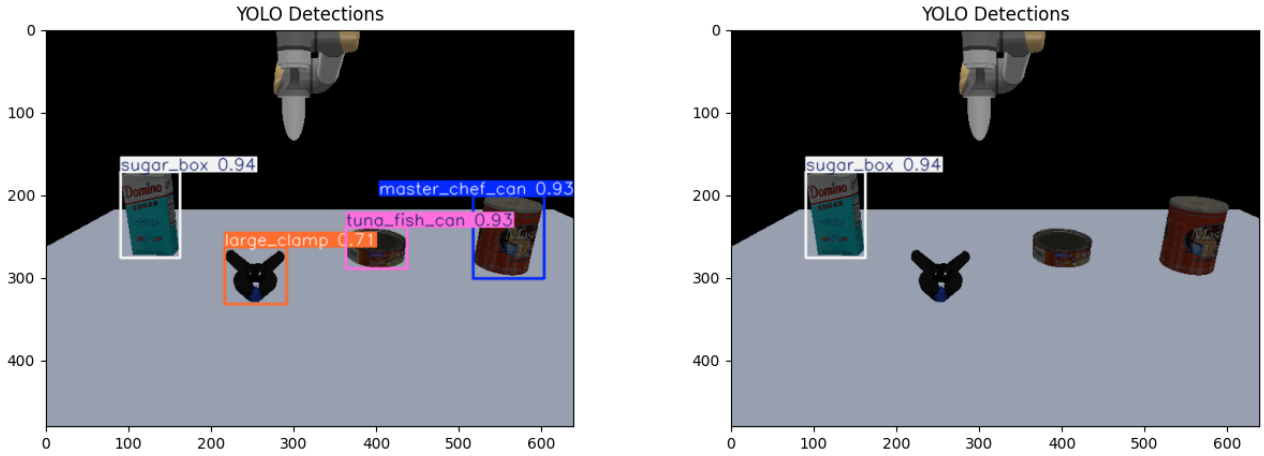

Figure 2: Left - Running the 'detect' command to see all of the items in the scene detected by YOLO. Right - Detection visualization of sugar_box from prompt: "move the sugar to the yellow trash"

4.2 From Detection to Interest Point

Only detecting the object is not sufficient, we want our arm to find the object in the environment and be able to pick it up. To do this, we did the following:

1. **Object Detection in Image Space:** The YOLO model processes the RGB image captured by the camera and returns a bounding box coordinates around detected objects
2. **Depth Estimation:** The depth map from the RGB-D camera provides the distance of each pixel from the camera. Using the object's **bounding box center**, the corresponding depth value is extracted (In tall objects we used a pixel close to the top of the bounding box instead of the center).
3. **3D Projection and Transformation:** Using the camera's intrinsic matrix $\mathbf{K}$ and depth value z, the object's 3D position in the camera coordinates is computed as:

$$x_c = \frac{(u - c_x)z}{f_x}, \quad y_c = \frac{(v - c_y)z}{f_y}, \quad z_c = z \tag{1}$$

where (u, v) are pixel coordinates, (c_x, c_y) are principal points, and (f_x, f_y) are focal lengths.

Finally, the object's world coordinates are obtained by applying the camera's extrinsic transformation matrix $\mathbf{T}$:

$$\mathbf{P}_{\text{world}} = \mathbf{T} \cdot \mathbf{P}_{\text{camera}} \tag{2}$$

where $\mathbf{T}$ represents the camera's pose relative to the robot's reference frame.

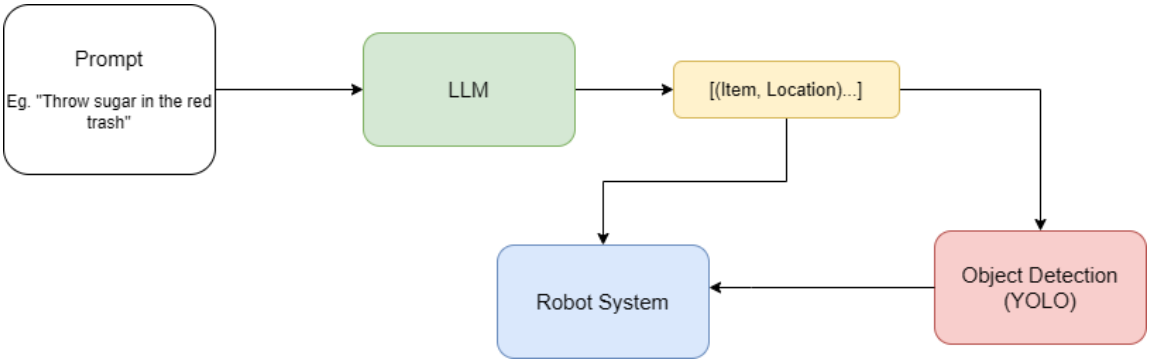

Figure 3: The pipeline from prompt to system command (In blue is inside the robot system).

* Small Note: the calculations in the code might differ slightly, this is due to the fact that coppeliasim's vision sensor returns data as flipped in the Y and X axis, so i adjusted the calculations in the code accordingly but the math is the same.

5 Lua vs Python

In our pipeline, we require an efficient and reliable approach to control the arm. CoppeliaSim's primary script language is **Lua**, using it offer some advantages such as optimized execution speed and built-in support for asynchronous and sequential processing. However, since the rest of the system is written in **Python**, it seemed practical to maintain a unified approach.

This, unfortunately, caused a critical challenge. The implementation using the ZeroMQ API operates in asynchronous mode by defualt, having to pass `setStepping(True)` function to enforce sequential execution. This led to an issue where the ZMQ server would enter a lingering state and time out if no commands were received withing a sort duration. Given that our system involves intermittent user input and object detection (and the time for API calls themselves), delays were bound to happen, which lead to simulation crashes. Running the simulation in full asynchronous mode resulted in incorrect motion-planning and inverse kinematics calculations.

To address these issues, we adopted a **hybrid approach**:

- **Lua** was used for performance-critical computations, such as `Inverse Kinematics (IK) calculations, OMPL Path Planning, and Collision Detection`. These computations are encapsulated in a Lua script within CoppeliaSim, allowing rapid execution without interrupting the simulation flow.
- **Python** was retained for high-level control, ensuring that the execution sequence remains synchronized. Python handles `User Input Parsing, Object Detection, and Command Sequencing`, while invoking the Lua functions in a structured.

This approach leverages the strengths of both languages: Lua for fast in-simulation execution and

Python for intuitive high-level logic. By integrating them effectively, we ensure accurate and efficient robotic control while mitigating potential synchronization issues.

6 The Robot Arm Scene

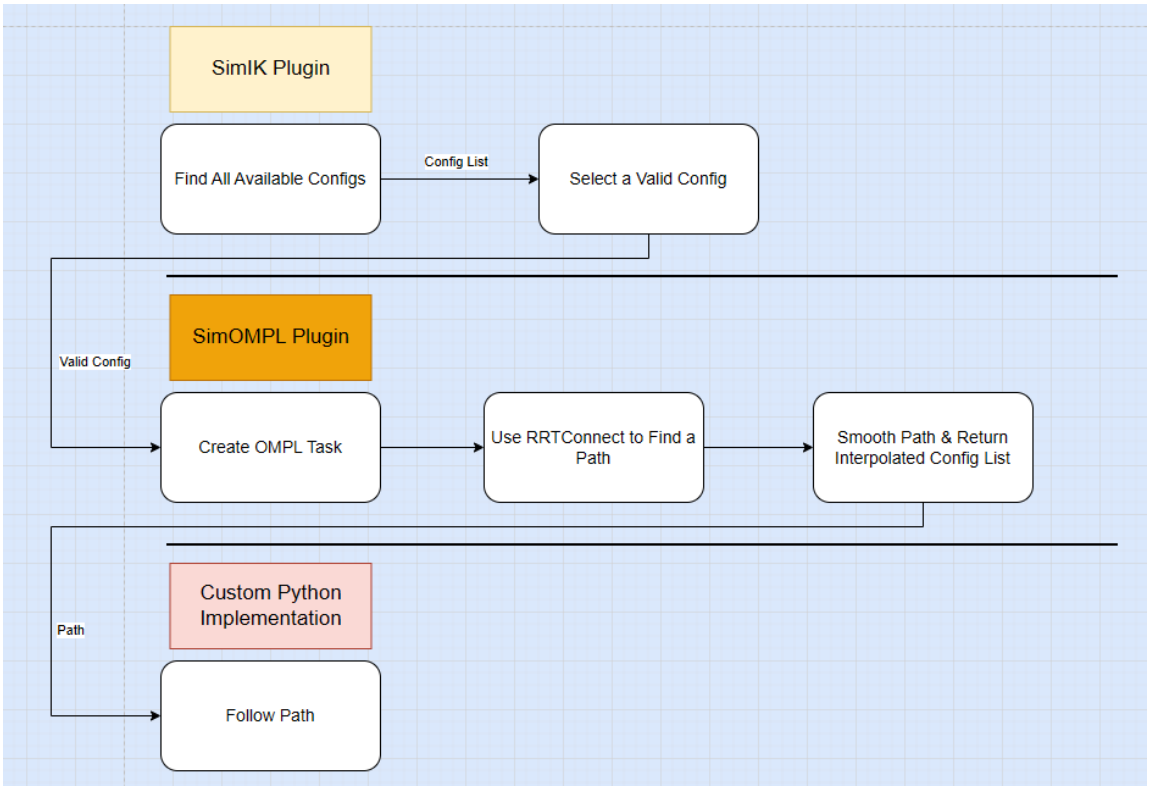

Figure 4: The pipeline from prompt to system command (In blue is inside the robot system).

6.1 Items and Locations

To construct a controlled environment for evaluating our robotic manipulation pipeline, we designed a simulation scene within CoppeliaSim that includes predefined objects and target locations. The setup consists of:

- **Target Locations:** Three distinct bins representing different placement areas. The 3D models of these bins were sourced from: [Recycle bin by Yuwei Xi [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/92Iv-tdQ49h)].

- **YCB Objects:** Four representative household items from the YCB Benchmark Dataset** [8]:
  - **Sugar Box**
  - **Tuna Fish Can**
  - **Master Chef Can**
  - **Large Clamp**
- **Robotic Arm Setup:** A UR5 robotic manipulator positioned on a stand, ensuring that its operational workspace covers both the table (where objects are placed) and the bins (where objects are placed).
- **End-Effector:** The robotic arm is equipped with a suction pad as the gripping mechanism. The choice between a suction pad and a parallel gripper was considered, with the final selection based on the simplicity of simulating object attachment within CoppeliaSim.

To assess the adaptability of our system, we created two variations of the scene:

1. **Unconstrained Environment:** The arm operates without any obstacles in its motion path.
2. **Constrained Environment:** Two additional obstacles are placed to evaluate the arm's ability to compute collision-free paths.

This structured environment ensures that our pipeline is tested in both ideal and constrained conditions, allowing us to validate the system's robustness in real-world-like scenarios.

6.2 Path Planning

Path planning is a critical component of robotic manipulation, ensuring that the robotic arm moves efficiently while avoiding obstacles. CoppeliaSim provides built-in support for motion planning through the **Open Motion Planning Library (OMPL)**, which we used for generating collision-free trajectories.

Given that our robotic arm is a 6-DOF manipulator, we considered a few sampling-based path planning algorithms, including Probabilistic Roadmaps (PRM) and Rapidly-exploring Random Trees (RRT). While PRM is effective in static environments, its reliance on a precomputed roadmap makes it less adaptable to dynamic conditions. RRT, on the other hand, provides a flexible and efficient solution by incrementally building a tree toward the goal state [9].

After evaluating, we selected the RRT-Connect algorithm, which extends RRT by growing two trees from the start and goal configurations, thereby improving convergence speed and success rates. The key advantages of RRT-Connect for our system include:

- **Efficiency:** Compared to RRTstar and PRMstar, RRT-Connect finds valid paths faster, which is crucial for near real-time applications instead of finding the shortest available path.
- **Deterministic Output:** Unlike standard PRM, which relies on probabilistic sampling, RRT-Connect offers a structured and consistent approach

Figure 5 illustrates the planned trajectories for both of our scenes. The results demonstrate that RRT-Connect consistently generates smooth and collision-free paths, even in constrained environments, validating its suitability for our application.

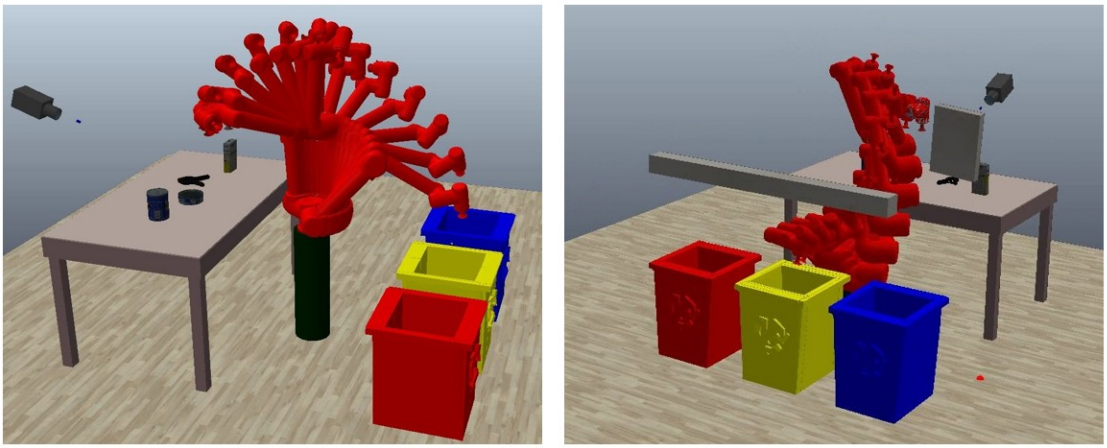

Figure 5: Left - path planned for yellow bin no obstacles. Right - path planned for yellow bin with obstacles.

6.3 IK

Inverse kinematics (IK) is essential in our setup to find point configurations required for out arm to reach its desired end-effector position. Solving this efficiently is also cruitial for our system to work. Thus, we used the CoppeliaSim plugin **simIK** that supports quick and accurate IK calculations for the arm.

For a reliable execution, we did the following steps:

1. **Computing All Valid Configurations:** Using `simIK.findConfigs()`, we retrieved multiple IK solutions and evaluated them.
2. **Collision Checking:** Each configuration was validated using collision detection system to eliminate unsafe solutions.
3. **Selecting an Optimal Configuration:** Among the valid solutions, the one with the smallest joint displacement from the current state was chosen to minimize movement and execution time.

Furthermore, for our pick-and-place task, we required the arm to approach objects from an overhead pose to ensure grasp stability. To achieve this, we enforced an orientation constraint, represented as:

$$\mathbf{R}_{\text{downOri}} = [0, 0, -\pi/2]$$

where $\mathbf{R}_{\text{downOri}}$ defines the end-effector orientation with the Z-axis aligned downward.

7 Conclusion and Future Work

We presented an integrated robotic manipulation pipeline that enables natural language-driven control of a 6-DOF robotic arm. By using a fine-tuned large language model (LLM) for command interpretation, a YOLOv8-based object detection system, and a physics-based simulation environment in CoppeliaSim, we demonstrated an efficient and accessible approach to robotic manipulation.

Our system successfully translates human intent into structured commands, detects and localizes objects within a simulated workspace, and executes pick-and-place tasks using inverse kinematics and path planning via the OMPL library. The results highlight that our approach achieves high accuracy while maintaining a low computational footprint, requiring only a total of 1.6GB of VRAM with inference times of less than one second. This demonstrates that complex robotic systems can be controlled effectively on lightweight hardware, making them more accessible for real-world applications.

Several areas remain open for future exploration:

- **Reinforcement Learning for Motion Optimization:** Recent studies suggest that reinforcement learning can enhance motion efficiency and adaptability in robotic systems [10, 11]. Integrating reinforcement learning for path planning could reduce movement time and improve robustness in dynamic environments.
- **Beyond Pick-and-Place:** While our system is currently optimized for pick-and-place tasks, incorporating generative models for trajectory planning could enable applications such as robotic drawing, 3D printing, or laser engraving.
- **Real-World Deployment via ROS Integration:** Although our system operates in simulation, CoppeliaSim provides an ROS-compatible plugin that ease's real-world deployment. Future work will involve transitioning from simulation to physical hardware, testing the system's robustness in uncontrolled environments [12].

References

[1] B. Dynamics, "Atlas: The world's most dynamic humanoid robot." https://www.bostondynamics.com/atlas.

[2] Tesla, "Tesla optimus: A humanoid robot for general-purpose tasks." https://www.tesla.com/AI.

[3] M. Robotics, "Mentee robotics: Menteebot." https://www.menteebot.com/bot/.

[4] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. http://arxiv.org/abs/1506.02640.

[5] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, pp. 72–82, December 2012. https://ompl.kavrakilab.org.

[6] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," 2022. https://arxiv.org/abs/2210.11416.

[7] S. Hafner, M. Schneider, and B. Stähle, "Performance comparison of yolov7 and yolov8 using the ycb datasets ycb-m and ycb-video." EasyChair Preprint 13111, EasyChair, 2024.

[8] B. Çalli, A. Walsman, A. Singh, S. S. Srinivasa, P. Abbeel, and A. M. Dollar, "Benchmarking in manipulation research: The YCB object and model set and benchmarking protocols," *CoRR*, vol. abs/1502.03143, 2015. http://arxiv.org/abs/1502.03143.

[9] R. Platt, "Prm and rrt: Sampling-based motion planning." Lecture Slides, Northeastern University, 2019. https://www.khoury.northeastern.edu/home/rplatt/cs5335_spring2019/slides/prm_rrt.pdf, Accessed: Feb. 2, 2025.

[10] D. Han, B. Mulyana, V. Stankovic, and S. Cheng, "A survey on deep reinforcement learning algorithms for robotic manipulation," *Sensors (Basel, Switzerland)*, vol. 23, 04 2023.

[11] J. Duan, W. Yuan, W. Pumacay, Y. R. Wang, K. Ehsani, D. Fox, and R. Krishna, "Manipulate-anything: Automating real-world robots using vision-language models," *arXiv preprint arXiv:2406.18915*, 2024.

[12] Y. Ding, X. Zhang, C. Paxton, and S. Zhang, "Task and motion planning with large language models for object rearrangement," *arXiv preprint arXiv:2303.06247*, 2023.