NeuroGrip: Autonomous Reshelving using Vision Language Models

Yu-Hsin (Thomas) Chan[1†], Boxiang (William) Fu[2†], Joshua Pen[3†], and Jet Situ[4†]

***Abstract*— NeuroGrip is a robotic reshelving system that integrates Vision-Language Models (VLMs) to enable adaptive pick-and-place operations. Traditional warehouse robotics rely on fiducial markers and fixed structures, limiting flexibility. NeuroGrip utilizes a Franka Panda manipulator and a RealSense depth camera to interpret visual and language-based inputs for object handling. This paper reviews related work in industrial pick-and-place automation, including chain-of-thought (CoT) reasoning and vision-language-action (VLA) models, and provides a detailed overview of NeuroGrip's setup and methodology. We evaluate our system and critical subsystems for repeatability and accuracy, and discuss challenges faced during implementation. Finally, we explore stretch goal implementations and potential future work for improvements. Our final implementation video can be found here.**

I. INTRODUCTION

NeuroGrip is designed to improve robotic reshelving by reducing reliance on structured environments and predefined object locations. Traditional pick-and-place systems depend on fiducial markers, prepositioned shelving, and rigid localization techniques, limiting adaptability. By integrating Vision-Language Models (VLMs), NeuroGrip processes visual and language-based inputs to handle objects more flexibly. This paper examines related work in pick-and-place automation and vision-language integration. NeuroGrip's hardware setup includes a Franka Panda manipulator, a RealSense depth camera, and a structured prop setup consisting of a pickup zone and shelf. It's perception stack includes a prompt-to-location system for translating user commands into actionable placements. It's manipulator control stack uses position control provided by the `Frankapy` package. We also evaluate the system's performance and discuss challenges faced during implementation. Finally, we explore some stretch goal functionalities and provide remarks for potential future work.

II. RELATED WORK

A. Industrial Pick and Place

A primary motivation for pick and place behavior in our VLM context is the use within industrial warehouse robotics, referred to as material handling. Current solutions have a strong focus on prior organization, specialized labeling, which can then be fed into a simplified ArTag or AprilTag model that provides localization for mobile robots. In this variation, a Universal Robotics 5 cobot, a robust 6DOF mounted robot has rapidly become an industry and scientific standard, and in many cases, have been attached to a Ridgeback robot, enabling it to move autonomously in pick and place environments.

However, the existing pick-and-place space is strongly reliant on prepositioning components for the robots to work, including guidelines, markers, signal extenders, shelving design, walkway and railway design, etc. While effective, it results in a highly static and maladaptive configuration, losing all versatility and cross-applicability.

B. Chain-Of-Thought (CoT) Vision Language Models

A similar project to ours was published late last year by Wang and Zhang et. al [6], in which they proposed the SeeDo framework (Figure 1), whereby they provide their own methodology for the translation of VLM output into robot-parsable language.

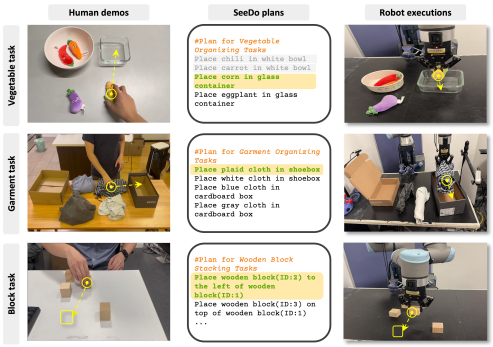

Fig. 1: SeeDo framework, as implemented by Wang and Zhang

Their variation is moderately more complex, utilizing chain-of-thought reasoning to decide what and where to perform object segmentation on a wide class of objects, thus performing generalizable tasks. From there, the CoT pipeline outputs predetermined positions, from which they can execute it with the robot arm, similar to our method of prepositioning the shelving unit and our existing calibration methods.

C. Vision-Language-Action (VLA) Models

More recent advances also propose the use of vision-language-action models. Such models forgo the translation into robot-parsable language entirely, and instead utilize an end-to-end learning model that takes in user prompts and directly outputs manipulator commands. Similar projects include the OpenVLA framework (Figure 2), TinyVLA

[1]yuhsinch@andrew.cmu.edu
[2]boxiangf@andrew.cmu.edu
[3]jpen@andrew.cmu.edu
[4]jets@andrew.cmu.edu
[†]Robotics Institute, Carnegie Mellon University

framework, Google RT framework, and the OmniManip framework [7, 4, 1, 2].

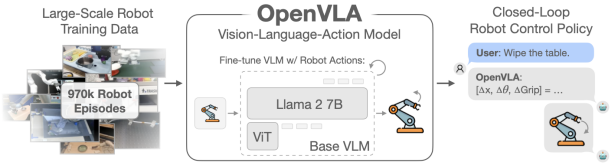

Fig. 2: OpenVLA framework

This type of model greatly generalizes the task space and can be used for a variety of end-to-end manipulation tasks. However, the compute requirements are far higher compared to traditional methods, often requiring more than 16 gigabytes of RAM and recent NVIDIA GPUs and CUDA drivers.

III. METHODOLOGY

Our methodology involves using a prompt-to-text Large Language Model to convert the user input prompt to a pick-up item and drop-off location. Simultaneously, the RealSense on the end-effector takes a screenshot of the pickup zone. Based on the input prompt, the center pixel location of the item is calculated in the pickup zone. This is converted to an end-effector location through camera calibration. The end-effector picks up the item and moves to the drop-off location specified. Finally, the robot arm moves to its reset position ready to take another user input.

A. Hardware Setup

The entire hardware setup of our project is depicted in Figure 3. The setup consists of a Franka manipulator, shelf, pickup zone, RealSense stereo camera, and an off-location PC that runs the prompt-to-location software.

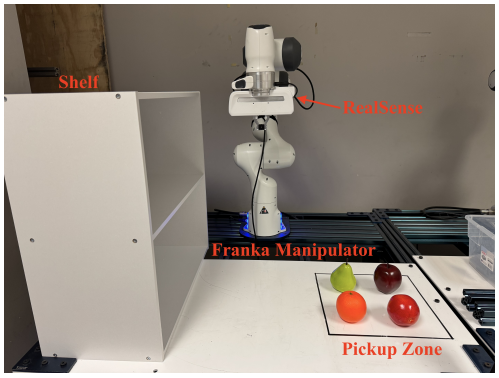

Fig. 3: Hardware setup of NeuroGrip

1) Franka Manipulator Setup: The manipulator used is a Franka Panda manipulator made available to us in the REL lab. In particular, we were allocated to the `iam-sneezy` manipulator. The robot arm has 7 degrees-of-freedom and can grab a payload of 3 kilograms. The hardware-software interface between the Franka manipulator and the control PC was already set up beforehand. The documentation for the interface setup can be found in Reference [8].

2) Shelf Setup: The shelf is placed in the left section of the world-frame table. We split the shelf into four quadrants (top-left, top-right, bottom-left, and bottom-right) for the end-effector to place items in. If the user inputs a prompt outside these four quadrants, the input will be invalidated and require the user to input a new prompt.

3) Pickup Zone Setup: The pickup zone is placed in the right section of the world-frame table. This is the location where the end-effector will attempt to pick up items and place them on the shelf. We have used dummy fruits for our setup, although the model is much more general and can recognize any object in the SAM2 database [5].

4) RealSense Setup: We installed a RealSense depth camera behind the Franka robot arm's end-effector and used it to capture images of the picking area. Figure 4 shows a clearer view of the RealSense mount location.

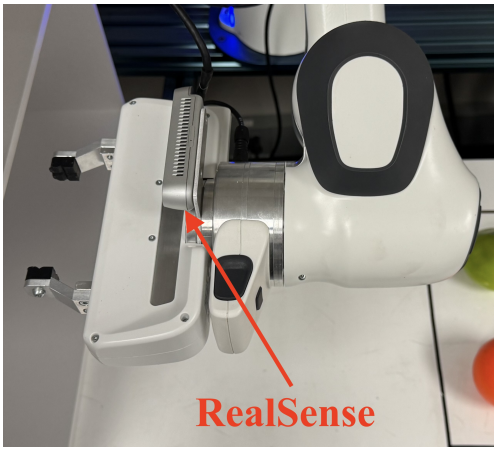

Fig. 4: RealSense setup of NeuroGrip

5) Prompt to Location Setup: Since we intend to run a visual language model in our project, we require a PC with a GPU. We discovered that the PC in the lab has an RTX3060Ti GPU installed, but the CUDA version is outdated for the model we want to run. While a software update could potentially resolve this issue, we were concerned that it might disrupt the entire environment. Consequently, we decided to deploy the model on Thomas' PC. Once a RealSense camera captures an image, we will send a request to Thomas' PC, which will process the image and provide the location of the target.

B. User Prompt Input Subsystem

Our implementation uses OpenAI's GPT-4o mini API for the user prompt input subsystem. Once `main.py` is ran, terminal queries the user to enter a command. Once the command is entered and `Return` is pressed, a server query request is sent to GPT-4o mini's API. Prompt engineering is used to specify the model to take in a natural language sentence and return a dictionary with two elements (Figure 5). The first element specifies the `object` to pick up, and the second element specifies the `destination` to put down. The `object`'s value is very generic, and our setup permits the use of any reasonably sized object below 3 kilograms

that exists in the SAM2 dataset [5]. The `destination`'s value permits only four values corresponding to the quadrant on the shelf. They are:

- **Top-left**: 0
- **Top-right**: 1
- **Bottom-left**: 2
- **Bottom-right**: 3

```
def parse_sentence(self, sentence):
    prompt = f"""
    Extract the object and destination from the given command.
    Return a JSON object with "object" and "destination" fields.

    We have 4 possible destination, top-left, top-right, bottom-left, bottom-right.
    the number of top-left is 0, top-right is 1, bottom-left is 2, bottom-right is 3.

    Example:
    Input: "Pick the apple to the top-left corner."
    Output: {{"object": "apple.", "destination": 0}}

    Now, process this sentence:
    Input: "{sentence}"
    Output:
    """
```

Fig. 5: GPT prompt specification

C. Perception and Segmentation Subsystem

The perception and segmentation subsystem segments the target object in the image and determines its position in the global frame. This process involves drawing a fixed picking zone to place objects, ensuring that the Franka arm is reset to a fixed position so that the RealSense camera can capture an image of the entire area. After camera calibration, if we know the target's position in the 2D image, we can determine its position in the world frame through simple interpolation. The entire subsystem pipeline is depicted in Figure 6.

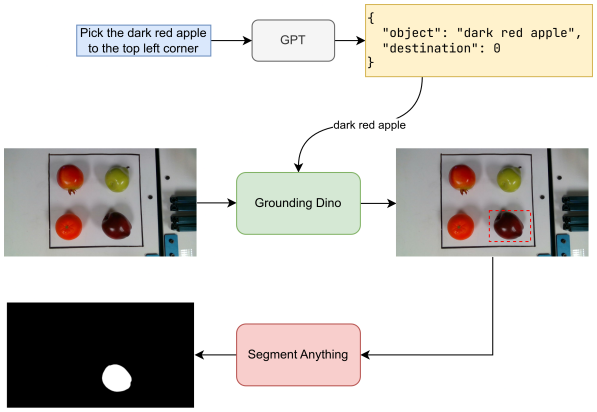

Fig. 6: Segmentation subsystem pipeline

1) Prompt to Bounding Box: Before segmenting the image, we need to obtain the bounding box of a target object. In this study, we opted to use Grounding DINO to achieve this [3]. Grounding DINO demonstrates impressive zero-shot capabilities, effectively identifying targets when provided with the correct prompt.

2) Bonding Box to Segmentation: After getting a bounding box is get, we use Segment Anything 2 to get a mask of the object [5]. In Figure 7, we can see that when we input *pear* as the prompt, Grounding Dino generates a bounding box and SAM2 generates a mask based on the bounding box.

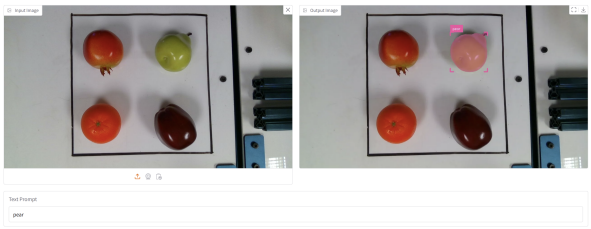

Fig. 7: Prompt to Segmentation

3) Mask to Global Position: After we place the mask on the object, we can determine its center in the image by simply averaging the coordinates of its mask. However, this position in the image coordinate alone is insufficient for the arm to move to the target's position in the real world. Therefore, we need to perform a transformation from the image coordinate system to the world coordinate system. To achieve this, we use three points: the top-left, top-right, and bottom-left corners of the picking area. We obtain the coordinates of these three points in both the image and global coordinate systems. For future masks, we can directly use their coordinates to interpolate with the three reference points, thereby obtaining the global coordinate of the object. The procedure is shown in Figure 8. Note that we only need three reference points as our world frame is guaranteed to be parallel to the image frame.

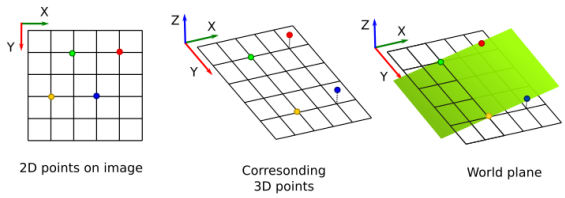

Fig. 8: Pixel coordinate to world frame transformation

D. Manipulator Control Subsystem

A finite state machine (FSM) is initialized to track the progression of the manipulator control task. The progression is a set of Cartesian way-points, and depends on the user prompt input. The manipulator control to reach these Cartesian way-points uses the `FrankaArm.goto_pose` method provided by the `Frankapy` package [8]. In essence, we transform the desired goal position into a homogeneous transform matrix and pass this as an argument to the method. The states of the FSM is discussed below in turn:

1) RESET state (Figure 9): This is the state the manipulator will move to once `main.py` is ran. This state is approximately 30 centimeters directly above the center of the pickup zone. Once the manipulator reaches this position, the terminal then queries the user for a command input. When a query is received and the valid parsed output is returned from OpenAI's API, the FSM transitions onto the next state.

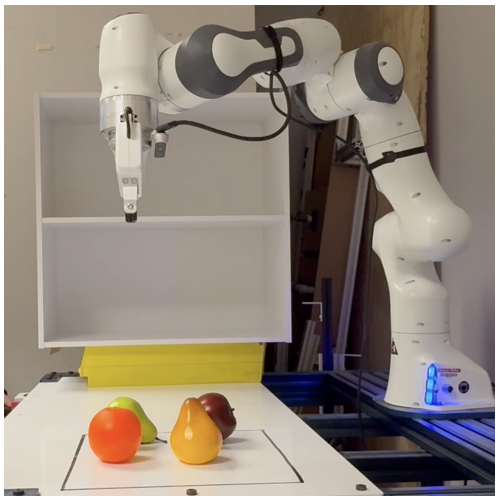

Fig. 9: RESET state

The reason we set this as the reset position is because it provides a direct view for the RealSense camera to view the pickup zone below. This allows us to calibrate the Cartesian x- and y-coordinates of any pixel in the picture taken by the RealSense camera. If at any time a state transition could not be completed, the manipulator returns to the RESET position and asks the user for a new prompt input.

2) PRE_PICKUP state (Figure 10): Once the Cartesian x- and y-coordinates of the desired pickup item is obtained from the perception & segmentation subsystem, the end-effector moves to approximately 10 centimeters above the center of the desired pickup item. This state acts as an intermediate step to prevent the end-effector from colliding with the item. Once this state is successfully reached, PICKUP is initiated.

3) PICKUP state (Figure 11): This state attempts to pick up the item by bringing down the end-effector and closing the gripper. Once this is successful, the manipulator is returned to the RESET pose as an intermediate pose. Note that we do not return to the RESET state, but rather only its pose. The reason for this extra step is to prevent cumulative errors from stacking up. Once the RESET pose is reached, the FSM initiates a state transition.

4) PRE_DROPOFF state (Figure 12): Depending on the `destination` value from the API callback, the manipulator will move to one of the four predetermined poses that correspond to the desired drop-off quadrant on the shelf. Each pose is approximately 5 centimeters in-front of the shelf to prevent colliding with the shelf itself. The end-effector is also rotated so that the gripper is horizontal (see Figure 4). This is to allow the end-effector to move further into the shelf while holding an object without colliding with the shelf walls.

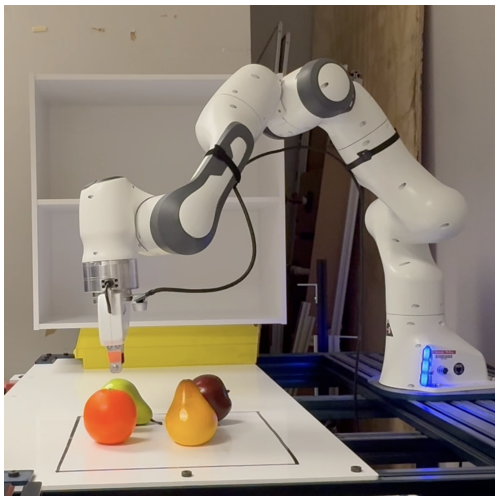

Fig. 10: PRE_PICKUP state

Fig. 11: PICKUP state

5) DROPOFF state (Figure 13): Finally, once the PRE_DROPOFF location is reached, the state transitions to initiate DROPOFF. During this state, the end-effector is extended 15 centimeters linearly into the shelf. Once this is reached, the gripper is opened and the object is dropped onto the shelf. The manipulator then reverses its extension and returns to the PRE_DROPOFF pose as an intermediate step. It then returns to the RESET state and pose ready to take in another user command.

IV. EVALUATION

This section presents an analysis of our integrated robotic system, which combines a Franka arm with dedicated control and perception stacks. The system interprets textual instructions to identify specific fruits within a designated pickup zone and execute precise pick-and-place operations. The following subsections detail the performance of the control stack in motion execution and the perception stack in object recognition, highlighting both the system's strengths and areas for improvement.

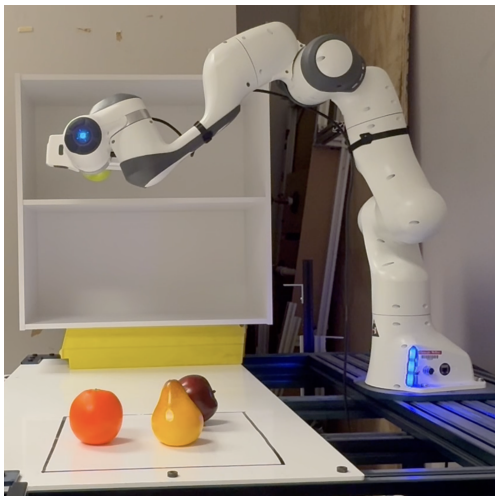

Fig. 12: PRE_DROPOFF state

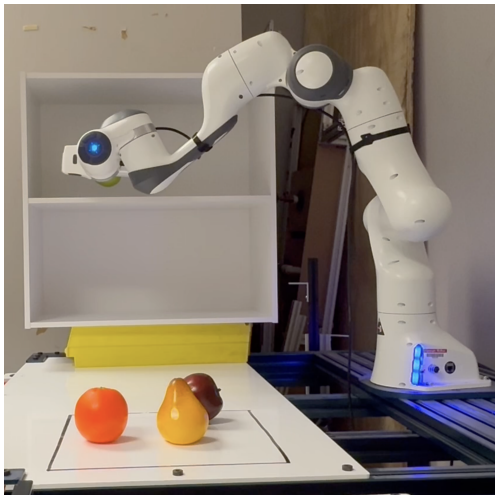

Fig. 13: DROPOFF state

A. Control Stack

We conducted a qualitative assessment of the control stack and identified several key issues. The Franka Emika Panda arm encountered constraints when attempting to reach positions beyond its defined workspace, effectively "hitting an invisible wall". To address this, we repositioned the pickup zone closer to the center of the table to ensure all targets remained within the reachable area, thereby preventing the arm from getting stuck. Additionally, the arm occasionally became trapped in certain configurations due to self-collisions. We resolved this by implementing a routine that returns the arm to a default starting pose after each operation, reducing the risk of persistent stuck joint configurations.

Another issue arose during the pickup phase, where the arm sometimes moved slightly off-center, causing the fruit to slip from its grasp (Figure 14). This was primarily due to minor inaccuracies in object position estimation from the perception stack's segmentation, combined with precision limitations in the control stack. The fruit used for testing was made of plastic and particularly slippery, making millimeter-level misalignments significant enough to cause slippage.

Despite these challenges, the control stack performed reliably, completing pickup and drop-off actions accurately approximately 75% of the time.

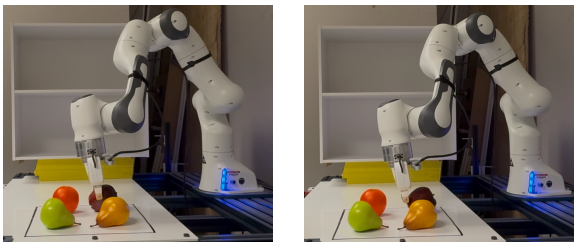

Fig. 14: Grasping misalignment: fruit before and after gripper closes

B. Perception Stack

1) Evaluation Setup: We evaluated the perception pipeline (Figure 6) using the following configuration:

- **Grounding DINO model:** grounding-dino-base
  - box confidence threshold: 0.30
  - text confidence threshold: 0.25
- **SAM model:** sam2.1-hiera-small

We collected a set of test images with objects arranged in various positions (Figure 15). Each trial consisted of supplying one image and one corresponding text prompt (Figure 16).

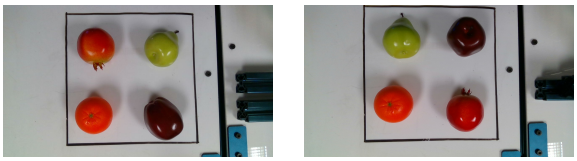

Fig. 15: Sample test images with varied object layouts

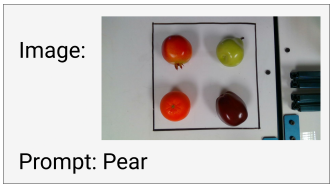

Fig. 16: Test protocol: one image paired with a single text prompt

2) Results: Across all combinations of images and prompts, the perception stack achieved:

- **Accuracy:** 92%
- **Average inference time:** 0.8 s per request

3) Failure Case Analysis: A representative failure is shown in Figure 17. Given the prompt "apple", the system segmented a pear instead. We attribute this to the Grounding DINO model's reliance on prompt specificity. By refining the prompt to "dark red apple", the segmentation succeeded (Figure 18).

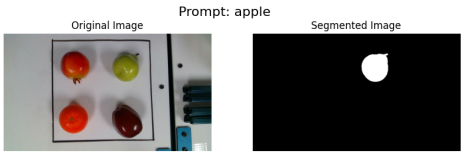

Fig. 17: Failure case: prompt “apple” yields incorrect pear segmentation

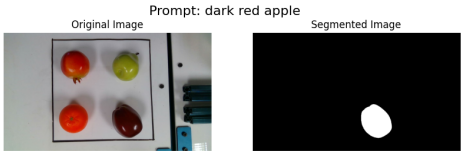

Fig. 18: Corrected prompt “dark red apple” produces accurate segmentation

The integrated system demonstrates promising capabilities to perform language-guided pickup and place tasks. The control stack effectively manages motion planning and execution, while the perception stack accurately identifies and localizes target objects based on textual prompts. Despite encountering challenges such as workspace limitations, occasional self-collisions, and minor perception inaccuracies, the system successfully completed approximately 75% of pick-and-place tasks. These results indicate a solid foundation for the system, with clear pathways for enhancements in control precision and perception specificity to further improve overall performance.

V. CHALLENGES

The NeuroGrip system faced numerous challenges during implementation and integration. Most notably, we initially tried off-the-shelf VLA models for our baseline implementation. This did not work due to various reasons. Additionally, we also faced CUDA version compatibility issues that resulted in numerous challenges that needed to be overcome.

A. OpenVLA Integration

For our preliminary project proposal, we initially wanted to use the open source OpenVLA model for deployment to our reshelving problem. The OpenVLA framework facilitated a pre-trained end-to-end model to directly obtain robot arm Cartesian control commands for the desired pick-and-place tasks. However, the model uses a Llama 2 7B neural network under-the-hood and requires 16 GB of VRAM. The control computer in the REL lab had insufficient VRAM to deploy the model. Additionally, the model was trained on WidowX arm, and would need to be further tuned to work on the allocated Franka arm.

B. TinyVLA Integration

After realizing that the OpenVLA model was not able to be deployed, we shifted our attention to deploying the TinyVLA model to our reshelving problem. This model has the advantage of being both lightweight and data-efficient. However, we encountered numerous challenges during the deployment process. There was very poor documentation on how to deploy the system, and the driver wrappers were customized for usage with a ZED 2i stereo camera. Since our Franka arm is deployed with a RealSense camera, additional work will need to be done to make the system compatible with the RealSense. Finally, the TinyVLA model also had very poor deployment performance (31.6% success rate). Therefore, we decided to implement our own custom solution instead.

C. Our Implementation

The main challenge that we faced during our custom method implementation was getting CUDA to work and enabling GPU acceleration for our model. During implementation, we found that the CUDA version on the control computer in the REL lab was outdated and incompatible with the GPT parser that we used for the user prompt input subsystem. To resolve this challenge, we ended up establishing a two-way connection between the lab control PC and Thomas’ home PC. The RealSense camera sends the image to the home PC to process, and sends back Cartesian locations for the desired gripper pose.

VI. STRETCH GOALS

Three stretch goals were also implemented to increase the capabilities of our system. The stretch goals allows our autonomous reshelving robot to parse spelling mistakes, detect non-valid items, and detect if the shelf is already occupied. Their implementations are discussed below.

A. Spelling Error

The first stretch goal extends the prompt input subsystem to understand slight spelling mistakes and still pick up the correct item and move to the correct location (Figure 19). We implemented this stretch goal by using OpenAI’s GPT-4o mini model and using prompt engineering to tell it to correct for slight spelling mistakes.

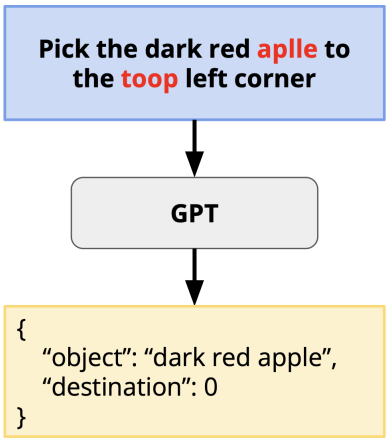

Fig. 19: Stretch goal 1: Spelling error

B. Detecting Non-Valid Items

Our second stretch goal extends the perception subsystem to reject picking up items if the item does not exist in the pickup zone (Figure 20). We implemented this using a similarity threshold so that only items above the threshold are picked up. We tuned this threshold and finalized the similarity value to 0.3. Items with a similarity value below 0.3 will be rejected for pickup and instead prompt the user for a new input.

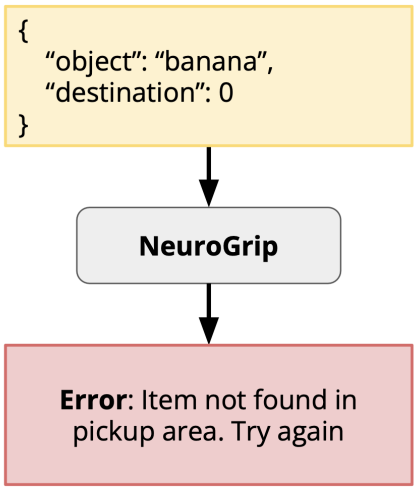

Fig. 20: Stretch goal 2: Detecting non-valid items

C. Detecting Occupied Shelf

Our final stretch goal extends the entire system's capabilities by having the model first check if the desired placement location is occupied. If it is, remove the occupying item first before picking up the desired pickup item (Figure 21). We implemented this using an initial lookup of the shelf to see if location is occupied. If it is, first pick up the occupying item and put it in the drop zone. The manipulator then returns to the pickup zone to pick and place the desired item.

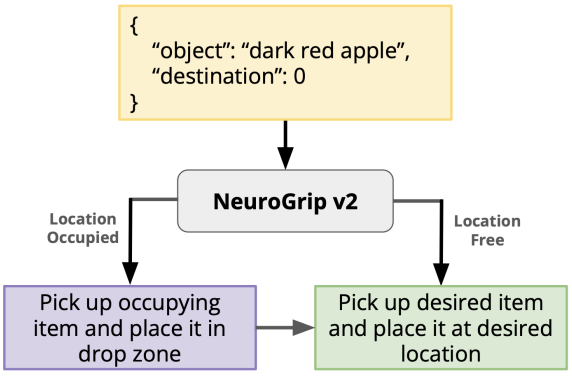

Fig. 21: Stretch goal 3: Detecting occupied shelf

VII. FUTURE WORK

There are two further directions to push our project into, which independently focus on the drawbacks that our existing implementation faces, and the remaining optimizations that would improve the performance of our existing implementation. Specifically, future work can be conducted in the realm of continued generalization along the VLA model side, improving flexibility across a variety of industrial regimes, or the commercialization route, in which our existing pick-and-place can find robust uses in industry as a turnkey solution, provided that we resolve existing minor shortcomings with our existing implementation style.

A. Generalization

While we are capable of a generalized search area and pick and place tasks, our pick and place still had to abide by static boundaries and rules imposed within our work area.

For instance, with proper VLA integration, we could achieve a more dynamic pick and place region, where we'd be able to move the arm and shelf around, and still have the system work optimally. Given more investment into time and training data, we could retrain a merge of the OpenVLA model where we could tune it to the parameters of the Franka arm, and run it on a more sophisticated backend that could handle the parameter size. We believe that in this particular case, we could achieve equivalent performance to our current MVP, but with a pipeline that becomes environmentally agonistic, achieving our original goal more effectively and close to our research intentions.

The next variation that pushes the research track is the generalization of robot arm action. Currently, our implementation revolves around pre-calibration of our cameras and the environment. Even without VLA, we can still use robust computer vision methods to properly scan our environment (assuming standard lighting conditions), and provide inverse kinematic movements that can mimic our performance in an agnostic environment. Achieving a baseline version of this could then allow simplified implementation of other workplace actions, such as pick and drop-off, re-shelving, swapping, and safety action-rejection. While our current actions have to be more hard-coded, implementing an inverse kinematic pipeline that leverages on-the-fly calibration and sequential actions could bypass the need for a VLA in specific circumstances.

B. Commercialization

The other direction we could work is robustifying our current methods for commercial and industrial deployment. As we already have achieved a generalized recognition and pick and place product, this could be mounted in industrial settings already for unloading and loading. By mounting a larger arm, and by improving our model to be generalizable and calibrated for any industrial arm or camera system with prior setup, we could begin performing autonomous loading and unloading in a pick and place environment, integrating with existing warehouse solutions without requiring costly relabeling or retooling investments on behalf of the client.

On our end, this would require work on single-shot calibration, generalization of our workspace to an agnostic, but confined work region, and generalization of our camera and

robot intrinsics for either a variation on any robot arm, or to a specific industrial model, such as the UR5. However, we can then build upon our existing VLM model for recognition of non-standard objects, which should still be effective provided standard operating conditions.

VIII. CONCLUSION

Existing reshelving methods in the robotic logistics industry, while effective for larger systems with sufficient investment, have significant shortcomings that require costly workarounds that reduce flexibility in operations. Our implementation, NeuroGrip, provides a framework for an agnostic warehouse reshelver that not only provides a framework for one of the hardest tasks, specifically, pick and place, but also provides an generalizable method for product recognition in pick and place, providing a turnkey adaptable solution that does not require expensive labeling and significant investment on the environmental side.

Our implementation relies on the baseline foundation of a Vision-Language Model, or VLM, which provides detection and segmentation of objects in the environment frame, providing a recognition system that can be used for any product. We then use a Franka arm and precalibrated Realsense depth camera and a designated pickup and dropoff zone, and demonstrate autonomous robotic movement for both the pick and place action. With a text-based user input that is human-parsable, commands can be sent to the robot in human language and transformed into viable action routes that accomplishes the action without further input. With an accuracy of 75%, and an inference time of less than a second, we have achieved a highly effective concept demonstrator that provides a strong foundation for a truly agnostic industrial pick and place robot.

We believe that with further work, we can reliably create a model that could be used in future industrial environments, solving one of the largest robotic bottlenecks of the last decade and pushing the envelope of autonomous robotic applications in dynamic logistical environments, branching past the guidewire and tagged systems that form the current foundation of the highly expensive fully integrated autonomous warehouse systems of today.

IX. RESOURCES

Our MVP implementation video can be found below:

`https://youtu.be/WbBWrO_oRvI`

Our final implementation video can be found below:

`https://youtu.be/2TF8aYAhmj0`

Our GitHub repository is available here:

`https://github.com/NeuroGrip/NeuroGrip`

X. REFERENCE

[1] Anthony Brohan et al. *RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control*. 2023. arXiv: 2307.15818 [cs.RO]. URL: https://arxiv.org/abs/2307.15818.

[2] Moo Jin Kim et al. *OpenVLA: An Open-Source Vision-Language-Action Model*. 2024. arXiv: 2406.09246 [cs.RO]. URL: https://arxiv.org/abs/2406.09246.

[3] Shilong Liu et al. *Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection*. 2024. arXiv: 2303.05499 [cs.CV]. URL: https://arxiv.org/abs/2303.05499.

[4] Mingjie Pan et al. *OmniManip: Towards General Robotic Manipulation via Object-Centric Interaction Primitives as Spatial Constraints*. 2025. arXiv: 2501.03841 [cs.RO]. URL: https://arxiv.org/abs/2501.03841.

[5] Nikhila Ravi et al. "SAM 2: Segment Anything in Images and Videos". In: *arXiv preprint arXiv:2408.00714* (2024). URL: https://arxiv.org/abs/2408.00714.

[6] Beichen Wang et al. *VLM See, Robot Do: Human Demo Video to Robot Action Plan via Vision Language Model*. 2024. arXiv: 2410.08792 [cs.RO]. URL: https://arxiv.org/abs/2410.08792.

[7] Junjie Wen et al. *TinyVLA: Towards Fast, Data-Efficient Vision-Language-Action Models for Robotic Manipulation*. 2024. arXiv: 2409.12514 [cs.RO]. URL: https://arxiv.org/abs/2409.12514.

[8] Kevin Zhang et al. "A modular robotic arm control stack for research: Franka-interface and frankapy". In: *arXiv preprint arXiv:2011.02398* (2020).