A Framework for LLM-based Lifelong Learning in Robot Manipulation

by

Jerry W. Mao

S.B. in Electrical Engineering and Computer Science and in Mathematics
Massachusetts Institute of Technology (2023)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Jerry W. Mao. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Jerry W. Mao
Department of Electrical Engineering and Computer Science
January 19, 2024

Certified by: Pulkit Agrawal
Assistant Professor, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Framework for LLM-based Lifelong Learning in Robot Manipulation

by

Jerry W. Mao

Submitted to the Department of Electrical Engineering and Computer Science on January 19, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

While robotic agents have become increasingly adept at low-level manipulation skills, increasingly they are being guided by large language model planners that decompose complex tasks into subgoals. Recent works indicate that these language models may also be effective skill learners. We develop HaLP 2.0, a modular and extensible framework for lifelong learning in human-assisted language planning, using GPT-4 to propose a curriculum of skills that is learned, used, and intelligently reused. Our system is designed for large-scale experiments, is equipped with a user-friendly interface, and is extensible to new skill learning frameworks. We demonstrate extensibility by comparing alternative implementations of our abstractions and improving overall performance by incorporating novel frameworks. Moreover, we conduct a focused study of GPT-4, using crowd-sourced scene and task datasets, finding that language models are capable agents of skill reuse and adaptation. We observe that while performance is dependent on language context, supplying optimized prompts can yield exceptional skill reuse behaviors. We envision that as manipulation primitives and large language models become more powerful, our system will be ready to synthesize their capabilities to create an autonomous system for lifelong learning, that can one day be deployed in the real world.

Thesis supervisor: Pulkit Agrawal
Title: Assistant Professor

Acknowledgments

This work would not have been possible without the unwavering support of my mentors and advisors. I cannot express enough thanks to Professor Pulkit Agrawal for his belief in me over the years; for the opportunity not only to learn, but also to grow from his wisdom. I am also grateful to Anthony Simeonov for his confidence in me, and for his continued guidance in my work and beyond.

I would also like to acknowledge my colleagues at the Improbable AI lab, especially Marcel Torne, Anurag Ajay, Bipasha Sen, and Younghyo Park, for their many valuable words of advice.

During my studies I was fortunate to serve as a Teaching Assistant for course 6.1220, Design and Analysis of Algorithms. I would like to express my sincere thanks to Professors Srinivasan Raghuraman, Bruce Tidor, Aleksander Mądry, Virginia Vassilevska Williams, Jonathan Kelner, and Julian Shun for trusting this opportunity to me, and for the wondrous experience of sharing in knowledge.

Throughout this chapter of my time at MIT, Myriam Berrios and Aurora Brulé offered me their constant support and advice. They brought periods of order and calm when I was most in need, and I am very appreciative for everything they have done for me.

Thank you also to all my communities here at MIT, who have brought me so much happiness. I would also like to express my appreciation for the companionship of Battlecode, Next Act, Next House and Sidney-Pacific. And to all of my friends, especially to Cindy, Michael, Ashhad, Ian, Mark and Nathaniel: thank you so, so much for every adventure, and

for every laugh, and simply for being there for me.

Finally, I am grateful for my family, for being with me along every step of this journey. Thank you for your words of encouragement, for your energy and motivation, for the comfort of your company, and for all the shared moments of joy.

Contents

1 Introduction **13**

2 Background **16**

2.1 Primitive skills for robot manipulation 16

2.2 Large language models for zero-shot task planning 17

3 HaLP 2.0 System Architecture **19**

3.1 Language-powered robot manipulation 20

3.1.1 Connection to physics backend 22

3.1.2 Scene perception and object segmentation 24

3.1.3 Natural language scene annotation 25

3.1.4 Large language model connection 25

3.1.5 Skill classes 26

3.1.6 Saving demonstrations 28

3.1.7 Recording interactions for further analysis 28

3.1.8 Whole system pipeline 30

3.2 Web-based user interaction interface 31

3.2.1 User interface design 32

3.2.2 Skill demonstration interface 35

3.2.3 Message protocol between orchestrator and robot system 36

3.2.4 Concurrency management 38

4 Evaluation of System Modules 40
4.1 Semantic segmentation performance 40
4.2 Collision-free motion planning 43
4.3 LLMs as lifelong learners 44
4.3.1 Experiment design 45
4.3.2 Metrics for lifelong learning performance 47
4.3.3 Characterization of large language model behavior 49
5 Limitations and Further Work 53
5.1 Evaluation of skill and plan correctness 53
5.2 System component extensions 54
5.3 Even larger scale studies 55
6 Conclusions 56
A Scene and Task Crowd-sourcing Survey 57
A.1 Survey content 57
A.2 Scene characteristics 59
B LLMs in Long Task Progressions 60
References 67

List of Figures

3.1 System architecture diagram for HaLP 2.0. 21

3.2 Sample images of scene renderings from PyBullet and Isaac Sim. 23

3.3 Detic and Segment Anything pipeline for object perception and segmentation. 24

3.4 System execution pipeline diagram for HaLP 2.0. 30

3.5 Sample screenshot of the web-based user interaction interface. 32

3.6 Different input modes supported by the web-based user interface. 34

3.7 Sample screenshot of the skill demonstration interface. 35

3.8 HTTP protocol between the user interface and the orchestrator. 36

4.1 Object detection success rates under different simulator renderings. 42

4.2 Sample Isaac Sim images where Detic fails to detect the object. 42

4.3 Pick object success rate using various motion planners. 44

4.4 Sample ShapeNet objects that induce failure modes in cuRobo grasping. . . 45

4.5 Skill growth and learning frequency for tasks in sample task progressions. . . 50

A.1 Sample kitchen image shown to survey-takers as an example of a robot scene. 58

List of Tables

3.1 Interface methods for classes in the world module. 23

3.2 Higher-level abstractions for instantiations of the chat module. 26

3.3 Protocol of messages and their fields, used to communicate between the robot system and the user web interface. 37

4.1 Pairwise confusion scores for semantic labeling under different simulator renderings. 42

4.2 Quantitative metrics for the evaluation of skill reuse in large language model lifelong learning. 47

4.3 An approximate categorization of the types of skill verbs learned by the large language model. 49

4.4 Proportion of unnecessary skills learned under differently engineered docstrings. 51

A.1 Number of scenes and tasks of each category in the crowd-sourced dataset. . 59

B.1 Skills learned (marked with asterisk) and used in crowd-sourced tasks proposed for kitchen scenes. 62

B.2 Skills learned (marked with asterisk) and used in crowdsourced tasks proposed for kitchen scenes. 64

B.3 Skills learned (marked with asterisk) and used in crowd-sourced tasks proposed for laundry scenes. 66

Chapter 1

Introduction

Humans have the very unique ability of easily solving complex spatial reasoning problems. When encountering a new environment for the first time, humans can efficiently adapt prior experience to manipulate their surroundings effectively. For example, when entering a kitchen, it is very easy for us to perceive individual utensils and to locate the many positions where they may belong in a drawer, shelf, or dishwasher.

Moreover, humans exhibit what we call "lifelong learning": the ability to accumulate and remember skills as they are learned over time. Rather than having to repeatedly learn the same skill each time it is required, humans retain this knowledge and compose them in novel combinations. Someone who has once loaded a dishwasher and once learned how to empty a mug, can logically deduce the steps for cleaning a mug that is still full—even if they have never performed this sequence of actions before.

Whether intelligent systems can also achieve lifelong learning has for a long time been an interesting problem to robotics researchers. Recent advances in Large Language Models suggest that this is a promising field; these language models demonstrate superior knowledge retention and reasoning abilities [1–3]. It is natural to conjecture that by equipping a LLM with the ability to perceive an environment, it can also learn to decompose high-level tasks into learnable skills. Indeed, systems have already been developed for mastering some

computer games that involve complex skill progressions [4].

While many systems based on language use an fixed class of skill types or policy architectures [5–8], limiting their learning potential, recent work involving robot manipulation has suggested that a LLM-based agent may also be effective in settings where the skill library is *open*. That is, LLMs may be capable of prompting for assistance when its skillset is deficient, and moreover learn to reuse those skills for subsequent tasks [9–11]. This is a unique challenge as the agent needs to be capable not only of solving tasks, but also of being aware of its own limitations in order to identify when new skills must be requested. Attaining a balance between *skill acquisition* (requesting new skills when needed) and *data efficiency* (not requesting new skills when not needed) is crucial to the usefulness of the system.

Our work is founded on HaLP [9], one such system for robotic lifelong learning. We posit that there is a wealth of knowledge to be gained from scaling up empirical studies of these systems, to more thoroughly characterize the lifelong learning capabilities of language models. As such, we build HaLP 2.0, a system that is more modular and more extensible, and ready to be run at scale, and we conduct focused studies on its individual modules to understand its behavior. In particular, our main contributions are:

- **A new system architecture design.** We design HaLP 2.0 to maximize system modularity across its many components. Modularity enables easy ablation studies for algorithm choices in those components, including the perception, scene annotation, and robot primitive behaviors. Furthermore, this would support using different skill implementations that rely on different foundational models, enabling wider support for a variety of skills that may be requested for the open skill library. We demonstrate this property by introducing support for a selection of new algorithms.

- **System interaction interface.** We create a user interface for easily interacting with the system in a simulated physics environment. Large-scale studies of our system require a user-friendly interface in order to elicit useful human prompts. We create a web interface that allows a streamlined user experience for supplying task commands,

execution feedback, and skill demonstrations to a virtual robotic agent.

- **System evaluation and crowd-sourced data collection.** We perform further studies on the performance of individual system components to complement and extend the results presented in HaLP [9]. We evaluate new choices of algorithms in separate modules, and also perform a large-scale crowd-sourced data collection experiment to characterize the lifelong learning capabilities of large language models. Our experiments demonstrate that language models are capable of very efficient skill reuse, and that while their behavior is dependent on context-engineering efforts, they have significant potential as agents of lifelong learning.

Chapter 2

Background

There have been many advances in the area of equipping robotic primitives with the planning capabilities of a language model. In this chapter we discuss some of the most relevant results that relate to our work, and formally define our objectives.

2.1 Primitive skills for robot manipulation

We focus on achieving complex tasks through applying sequences of pre-trained models for primitive *skills*. There are many end-to-end frameworks for executing pick-and-place actions in scene rearrangement tasks.

TransporterNets [12] are an imitation learning model for table-top pick-and-place rearrangement tasks. Whereas TransporterNets use a suction gripper to move objects to their destination, there are also solutions for other robot grippers. Contact-GraspNet [13] can generate grasps to execute "pick" actions on objects with complex geometries. Neural Descriptor Fields [14] generalize to arbitrary relational alignments, capable not only of aligning a robot end-effector to an object, but also of objects to each other [15].

These models can be extended with language models to create language-conditioned policies. CLIPort [16] combines TransporterNets with CLIP [17] to execute table-top manipulation tasks specified in natural-language. PerAct [18] is a behavior-cloning agent that

uses a transformer to convert language commands into voxel features that can be decoded into robot policies.

While these models are effective at executing individual skills from natural language commands, there is promise that large language models may also be capable at decoding complex tasks into planned sequences of skills.

2.2 Large language models for zero-shot task planning

A common paradigm for performing complex natural-language tasks is to train a system for decomposing such tasks into lower-level steps. The goal is to have these individual *skills* be simple enough to learn a policy for. Equipped with a collection of these primitives, as well as algorithms for language-conditioned manipulation, we turn our attention to the problem of *planning* the sequence of skills.

Large language models are well-suited for this problem; they are capable of providing task plans in a natural language format [5, 19], as symbolic task plans [6], as sequences of calls to high-level skill primitives [8, 9, 20], or even as computer programs with lower-level code [21]. At the same time, vision-language models have been used to learn symbolic plans from videos of human demonstrations [22].

Inner Monologue [8] demonstrates that large language models can be particularly effective when additionally provided with environmental feedback. Voyager [4] utilizes feedback to learn complex skill progressions, proposing its own curriculum as a sequence of intermediate subgoals. This yields more complex skills as existing abilities are composed; by learning its own skills, the system becomes increasingly capable over time. Likewise, in real-world systems, HaLP requests human demonstrations for learning new skills when existing ones are deficient [9–11].

In our work, we create HaLP 2.0, a remodelled design of HaLP that is more extensible and scalable to large-scale experiments. We perform a large-scale analysis of the capacity

for LLMs to effectively engage in lifelong learning and characterize model behavior when presented with novel real-world tasks. Additionally, we provide new module implementations and evaluate the resultant improvement in system performance.

Chapter 3

HaLP 2.0 System Architecture

In this chapter we present the system architecture for HaLP 2.0. HaLP 2.0 is a system that equips robotic manipulation frameworks with the knowledge of large language models such as GPT-4, so that high-level natural language commands can be decomposed into individual manipulation skills. This allows complex tasks to be autonomously executed on a robotic system. We additionally provide the LLM with an interface for requesting new skills when the existing skillset is deficient; the system can then acquire these skills via imitation learning from human demonstrations. The system accumulates a growing skillset over time, and by evaluating how effectively those skills are used we can demonstrate the system's ability for lifelong learning.

HaLP 2.0 is designed to be suitable for large-scale research on robotic manipulation and lifelong learning tasks. Therefore, our system design is optimized for several objectives:

- **Modularity.** The full pipeline for executing a high-level natural language command consists of several components, including the large language model, scene perception, low-level manipulation, among many others. As detailed in Section 3.1, there are many suitable implementations for each of these modules, each of which may be of interest. We therefore seek to maximize extensibility to accommodate these alternatives, as well as to enable ablation studies.

- **Reproducibility.** Certain behaviors in the system may be stochastic. For ease of post-hoc analysis, we aim to make experiment results easily reproducible. This allows alternative algorithms to be reliably compared on the same scenes.

- **Scalability**. We envision that HaLP 2.0 may be used to conduct large-scale robotics experiments. To do so, we require the ability to record data while multiple users concurrently interact with the system.

- **Usability.** Large-scale robotics experiments involve collecting robot interaction data with non-expert users. This requires a user-friendly interface so that non-experts can have a streamlined experience, yielding higher quality data.

HaLP 2.0 is designed with these objectives in mind; the details of its architecture are explained in Section 3.1. We then further equip it with a web-based user-interface suitable for experiments with non-expert users, as detailed in Section 3.2.

3.1 Language-powered robot manipulation

In this section we describe the system architecture for robotic manipulation with a LLM-based task planner. This is the core functionality of HaLP 2.0, and can be commanded by a human via a simple terminal-based interface.

HaLP 2.0 consists of multiple components that interact with the various foundational models that comprise the entire system. These components are isolated into individual modules that interface with each other; some components may support alternative implementations as needed. Figure 3.1 shows the relationships between these various modules, which are outlined below.

- The **world** module (Section 3.1.1) is the physics backend in which the robot and the scene are loaded. All robot actions are executed in the world.

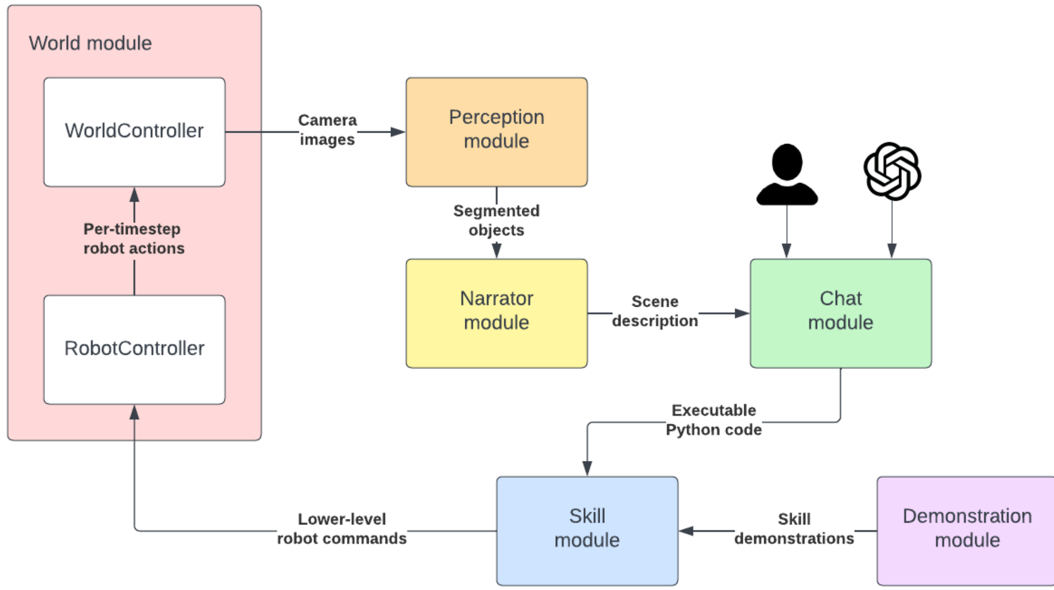


Figure 3.1: System architecture diagram for HaLP 2.0.

- The **perception** module (Section 3.1.2) detects, segments and labels objects in the scene, using cameras in the world.
- The **narrator** module (Section 3.1.3) writes scene descriptions in natural language. These annotations can be used as context for text-based language models.
- The **chat** module (Section 3.1.4) interacts with the LLM and prompts it for text and executable code.
- The **skill** module (Section 3.1.5) implements various actions the robot can perform.
- The **demonstration** module (Section 3.1.6) stores and retrieves skill demonstrations in a canonical format.
- The **recorder** module (Section 3.1.7) saves and replays system interaction logs.

The following sections describe each of these modules in more detail. We finally summarize how they are combined in Section 3.1.8.

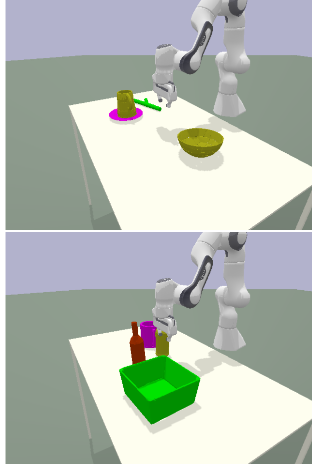
3.1.1 Connection to physics backend

The **world** module connects the system to a physics backend. It exposes an interface for the system to execute low-level robot commands and to obtain observations from the camera system. Instantiations of the world module can be a physics simulator, or the option to link to a real-world robotic system, such as via Polymetis [23].

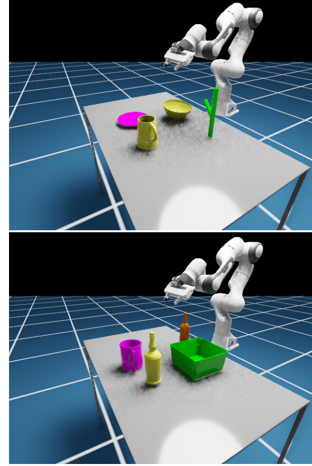
The predecessor HaLP was implemented purely for the PyBullet simulator [24]. We continue support for PyBullet using the AIRobot library for robot control [25] and the ShapeNet universe of objects [26]. On the other hand, we additionally provide an implementation for NVIDIA Isaac Sim with the Orbit framework [27, 28]. We choose to provide Isaac Sim because we believe its scene rendering looks more realistic; we include some sample visualizations in Fig. 3.2. This is a useful property because most image segmentation models are trained to work on real-world images; better rendering quality may lead to better system performance. We refer to Section 4.1 for a further discussion and analysis of this hypothesis.

Each world implementation conforms to the same interface, allowing the same algorithms to easily be deployed in different environments. In particular, it contains a `WorldController` that allows scene-level operations, and a `RobotController` that allows low-level robot manipulations. Their methods are described in Table 3.1. `RobotController` also provides default implementations for some slightly higher-level methods such as grasping given a gripper pose and placing given an object pose. These are stateful compositions of the interface methods that store the relative pose between the robot gripper and the grasped object.

We provide two different implementations of the `RobotController.goto` method. First, the Orbit framework provides a differential inverse kinematics controller for commanding the robot to target poses. This is a relatively simple controller; it does not consider any obstacles in the scene. As a result, motion plans from this controller may cause the robot to collide with objects in the scene, perhaps even hitting out of reach an object intended to be grasped.



(a) Sample scenes from PyBullet.



(b) Sample scenes from Isaac Sim.

Figure 3.2: Sample images of scene renderings from PyBullet and Isaac Sim.

Method	Return type and description
`WorldController`	
`.generate_scene(n: int)`	*None* Reset the scene to a new procedurally-generated configuration, consisting of n objects on a table.
`.get_camera_images()`	`CameraImage[]` Capture images from each camera in the scene. Each camera returns a RGBD image, the unprojected point-cloud, and any available ground-truth scene segmentation (if any).
`.get_objects()`	`ObjectGroundTruthInfo[]` Return the ground truth information about the objects in the scene. Each object returns its ground-truth semantic category, and its ID in the world.
`.get_robot()`	`RobotController` Return the robot controller for this world.
`RobotController`	
`.goto(pose: Pose)`	*None* Move the robot end-effector to the given pose.
`.go_home()`	*None* Return the robot home by resetting the joints to their home state.
`.set_gripper(open: bool)`	*None* Open or close the robot gripper.

Table 3.1: Interface methods for classes in the world module.

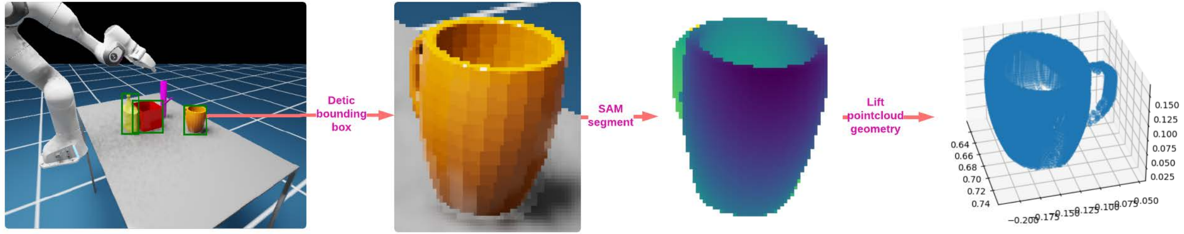


Figure 3.3: Detic and Segment Anything pipeline for object perception and segmentation.

Our second implementation uses cuRobo, a framework for collision-free motion generation [29]. While cuRobo has a specialized interface for Isaac Sim environments, it is also extensible to other physics backends. We posit that using a collision-free motion planner will improve system performance, by mitigating the failure mode of the robot arm pushing an object out of reach while attempting to grasp it. We evaluate this hypothesis in Section 4.2.

3.1.2 Scene perception and object segmentation

The **perception** module receives camera images from the *world* module and returns object segmentation information. This data includes image crops of segmented objects, pointcloud geometries lifted from a depth map, and semantic object class labels.

We provide an implementation using a similar pipeline as in HaLP of Detic [30] and Segment Anything (SAM) [31]. This pipeline is illustrated in Fig. 3.3. Given camera images from the world module, we first use Detic to detect objects and return their bounding boxes and category labels. Then, an image crop is generated and an exact segmentation is inferred using SAM. The pointcloud geometry can then be lifted from the segmentation. For Detic, we choose to use a closed vocabulary consisting only of the object categories we are working with. We find that an open vocabulary leads to many spurious objects being output.

When evaluating the performance of other modules, it is desirable to have perfect segmentation performance so that results are not impacted by perception failure. We therefore provide an alternative implementation. As simulators in the world module provide camera images with ground-truth information, the alternative implementation directly returns the

ground-truth segmentation, ready for use in such studies.

3.1.3 Natural language scene annotation

The **narrator** module receives segmented object data from the *perception* module and writes a natural language annotation of the scene. We provide an implementation based on the algorithm from HaLP, and refer to the original paper for details [9]. Objects are assigned graph vertices with edges representing geometric relations between them; the scene description consists of a natural language transcription of each edge in depth-first traversal order.

3.1.4 Large language model connection

The **chat** module facilitates communication with a large language model API, taking its name from OpenAI's ChatGPT [32]. Each implementation of the module may use a different underlying transport method for handling queries, allowing a choice of a different language models or APIs. For example, we allow GPT-4 [2] and GPT-4V [3] via API, and ChatGPT via web interface. On top of the transport "layer" is a collection of methods that provide higher-level abstractions; they are always available, as outlined in Table 3.2.

We use the chat module to ask the language model for task plans and executable code; code is executed on the robot. The specific interactions are outlined in Section 3.1.8.

Our default implementation of the chat module uses the OpenAI chat completion API to interact with models online. We predominantly use the GPT-4 [2] and the GPT-4V models [3]. We also provide support for users who wish to run HaLP 2.0 but do not have access to the OpenAI API. In this case, we allow them to work directly with ChatGPT in their browser, such as the OpenAI GPT-3 model [1]. This alternative implementation directly outputs prompts to the standard output stream for users to copy-paste into ChatGPT; responses from the LLM are then read back into the system via file input.

As such, this implementation provides a convenience for users who prefer a simpler setup procedure. However, we recommend using the online API for a more streamlined workflow.

Method	Return type and description
`ChatInterface`	
`.add_feedback(feedback: str)`	*None* Records a piece of feedback. This feedback is prepended to the prompt during the next query to the large language model.
`.get_text(prompt: str)`	`str` Send the prompt with any undelivered feedback to the implementation-specific transport, returning the large language model's text response.
`.get_code(prompt: str)`	`str` Perform the same query as `get_text`, but return only executable code snippets found in the response. Code snippets are found via a regex search for triple-backtick delimiters (```) as we assume the large language model provides a response typeset in Markdown.
`.clear_context()`	*None* Clears all chat history so that subsequent requests start with an empty context.

Table 3.2: Higher-level abstractions for instantiations of the chat module.

3.1.5 Skill classes

The **skill** module provides implementations for each of the skills we expect the robot system may require. Each skill can be called by code written by the *chat* module, and is executed in the *world* where its effects can then be observed.

The system is initialized with a "base" skillset consisting of some simple primitives. These enable it to perform the most basic actions such as grasping and placing. These are included in our skill implementations, in addition to imitation-learning frameworks that allow new skills to be acquired via human demonstrations.

The base skillset also includes methods that perform only perception-related tasks that observe the scene. These initial skills are based on those described in [9, 10], and we summarize them here.

- `find(object_name: str, visual_desc: str, place_desc: str): int`. This skill finds an object that best matches the given search terms. We support searching given at least one of an category name, a visual description, or a description of where the object is located.

  To handle visual descriptions, we use CLIP [17] to embed image crops for each object, and determine which object has the highest correspondence with the text search term. For object names and place descriptions, we generate a natural-language sentence describing each object, and match the search terms using sentence similarity scores from BERT [33].

- `get_location(obj_id: int): [x,y,z]`. This method returns the location of the specified object. It is inferred as the mean coordinate of the object's pointcloud.

- `pick(obj_id: int): None`. This skill navigates the robot manipulator to the specified object and carefully grasps it, lifting the object off the table. To find a valid grasp, we use a pre-trained Contact-GraspNet model [13].

- `get_place_position(obj_id: int, reference_id: int, desc: str): [x,y,z]`. This method determines a placement position of an object *relative* to a given reference object, matching the given description. For example, it can be used to determine the coordinates for placing an apple inside a bowl; in this case, the refernce object is the bowl, and the description is "inside".

- `place(obj_id: int, position: [x,y,z]): None`. This skill places the object in the given location. It is a simple implementation unaware of collisions: it moves the object above the given coordinate and releases it.

- `learn_skill(skill_name: str, docstring: str): Skill`. This method requests that a new skill is added to the robot's library. The skill is instantiated from an imitation-learning agent, with any human demonstrations collected as necessary.

For few-shot imitation learning, we provide Neural Descriptor Fields [14] for object grasping and placing, as well as NDF-based relational rearrangement [15]. These frameworks are used by `learn_skill`. Learnt skills are stored in the same Python namespace as the execution environment and can be directly called as the LLM sees fit.

We note that we decouple skill demonstration collection from skill acquisition requests. This allows demonstrations to be collected in advance, and reused as needed across multiple experiment trials. We detail saving demonstrations in Section 3.1.6 and the collection workflow in Section 3.2.2.

3.1.6 Saving demonstrations

The **demonstration** module contains protocols for reading and writing serialized human expert demonstrations as required by imitation-learning skills.

All demonstrations are saved as a collection of files; the main specification file describes the type of model the demonstration is for, along with other key information. At present, we only collect demonstrations for Neural Descriptor Field models, but the framework is easily extensible to support other varieties.

Each demonstration for an NDF model is saved as a pair of the two objects in the relation. For example, grasping skills involve the robot gripper and an object; placement skills involve an object and a tabletop. We save a "`.obj`" mesh for each object in a canonical pose; this allows pointclouds to be extracted necessary. We also save the geometric transforms that place both objects in the final configuration, to complete the demonstration. These transforms are the ones used by NDFs.

3.1.7 Recording interactions for further analysis

The **recorder** module is responsible for recording and replaying all system interactions. In order to perform post-hoc analysis on user interaction data, we equip the entire system with the ability to log and recreate all interaction sequences. Saving complete experiment

metadata allows us to make results reproducible.

We maintain modularity despite the each module having its own unique combination of data to be saved. Our main feature is a Python *decorator* that marks a method as recorded; as such, the recorder is minimally invasive to the other modules. Recorded methods generate log entries that can be reloaded; each log entry contains the fully-qualified name of the method being called, all function arguments, and the function outcome. Specifically:

- When running in *regular* mode, the recorder executes the method and saves its outcome to an experiment log. This outcome may be either a return value (which is saved), or an exception (which is saved and re-raised).

- When running in *replay* mode, the recorder seeks forward through the experiment log for a matching call. It returns the corresponding outcome (that is, it returns a saved value or raises a saved exception).

These log entries are saved to disk as a series of pickled log objects. We apply the recorder to save all human commands and feedback, as well as outputs from the *chat* and *perception*modules, and intermediate foundational model predictions from any *skills*.

We note that HaLP 2.0 is inherently stochastic: this is true not only of the models being used, but also of the physics simulator, in which individual steps may behave non-deterministically. However, reproducability requires being able to replay robot trajectories so that subsequent actions have consistent effects. To address this:

- The recorder saves the random seed, and resets it during replay.

- The recorder also allows arbitrary log lines to be written; the `RobotController` uses this feature to regularly save intermediate states along motion trajectories. During replay, all object poses are constrained to follow this sequence of waypoints and therefore have the same motion.

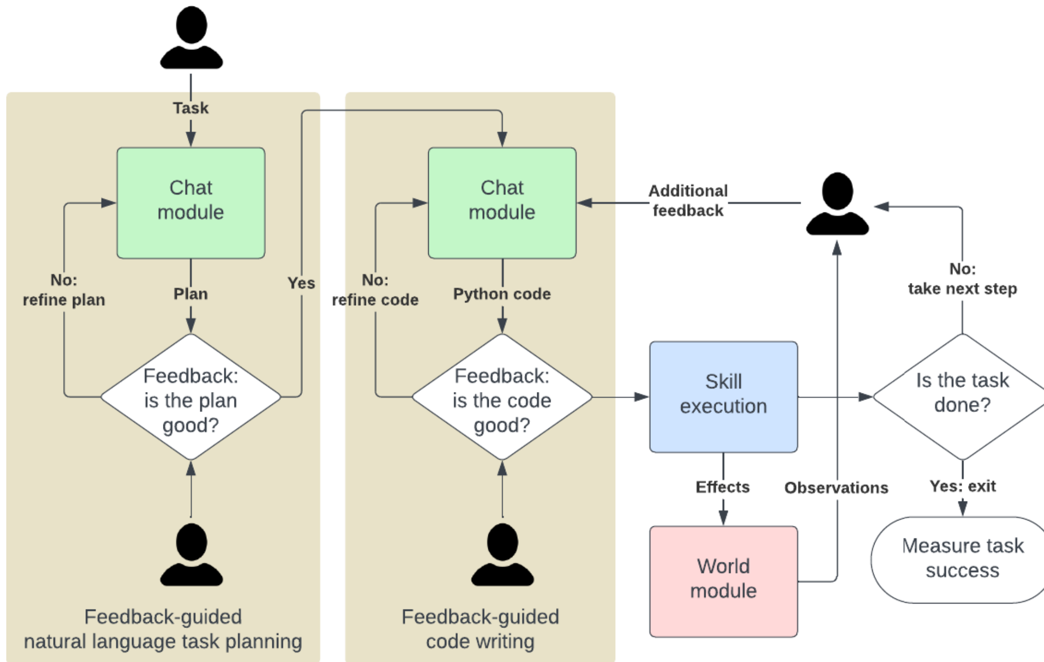


Figure 3.4: System execution pipeline diagram for HaLP 2.0.

3.1.8 Whole system pipeline

In this section we describe how all the individual sections are combined to form the full execution pipeline. This pipeline is the main entrypoint for the HaLP 2.0 and its experiments; we show it schematically in Fig. 3.4. Specifically, the user begins by creating a scene and selecting a task. Then:

1. We generate a natural-language plan for how the task is to be performed. The language model is given the scene description (from the *narrator*) and the task, and asked to provide a plan. If there is any human critique, the plan is iteratively improved by the language model; once approved, the planning step is complete.

2. We generate a Python code snippet for executing one step of the plan by querying the language model. If there is any human critique, the code is iteratively improved by the language model; once approved, the code is executed.

3. The human is asked to critique the outcome of the action, which is added to the language model's context before subsequent steps continue. This process repeats until the task is finished.

At runtime, the user is able to select which implementation of each module they would like, by supplying an appropriate command-line flag. As a result, conducting ablation studies on these modules is very streamlined.

Additionally, we allow the pipeline itself to be customized with different implementations as needed for other styles of experiments. Therefore, it also functions as an extensible module, and we call it the **experiment** module.

3.2 Web-based user interaction interface

We aim to make the fruits of robotics research accessible to the general public. While hardware logistics limit the possibility of distributing physical robots, we can use a user-friendly web interface to enable simulated interaction. We hope that learnings from data collected through this interface will fuel rapid research development, that will eventually enable large-scale physical deployments.

To this end, we design and publish a user-friendly webpage that allows non-expert users to remotely interact with HaLP 2.0. Fig. 3.5 shows a screenshot of this interface. As previously discussed, our objectives for the interface prioritize system scalability and usability; in this section we discuss these design choices. Section 3.2.1 describes the user interface, which also supports a skill demonstration interface (Section 3.2.2). The interface connects to a Flask webserver [34] and application orchestrator, which manages multiple HaLP 2.0 instances running in parallel. It communicates with those instances via a message-passing protocol (Section 3.2.3), while keeping them in parallel processes to maximize throughput while subject to resource constraints (Section 3.2.4).

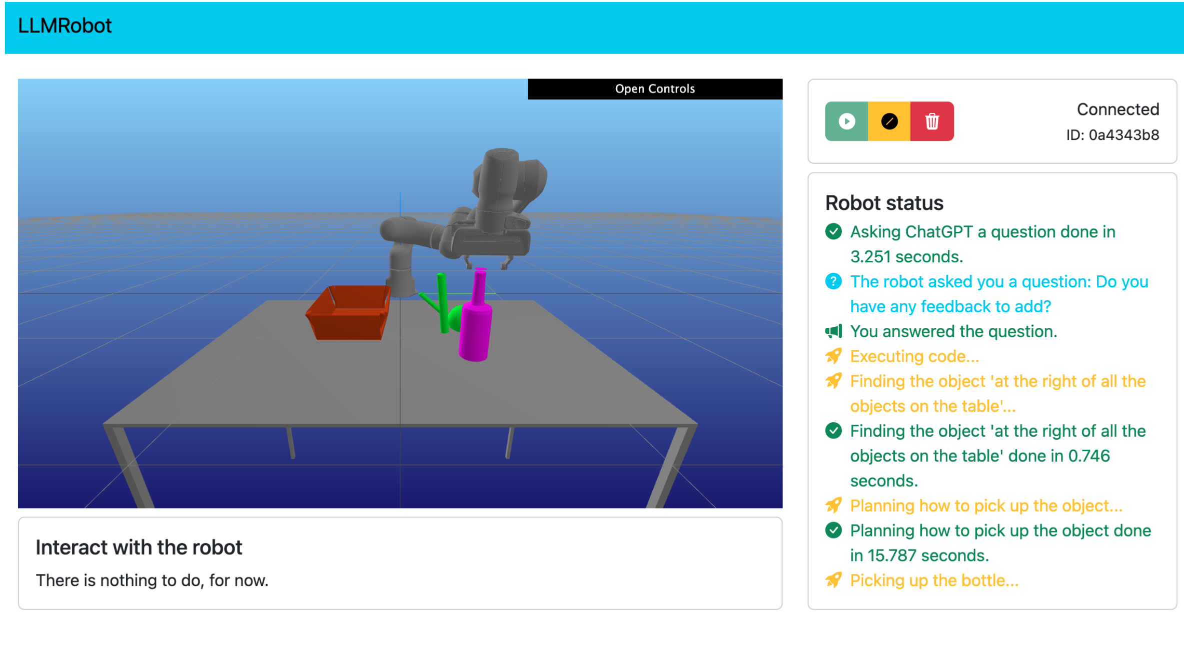


Figure 3.5: Sample screenshot of the web-based user interaction interface.

3.2.1 User interface design

The user interface arranges into a friendly layout the elements that would normally be accessed from a researcher's terminal prompt. These consist of a connection status indicator, a robot scene visualizer, an interaction panel, and a system status log. Together, they provide the user with a streamlined experience.

Connection status indicator. The connection status indicator provides buttons for connecting and disconnecting from HaLP 2.0. We aim to ensure active engagement before any resource-intensive physics environment is assigned.

When the webpage loads, the user is instructed to click on the green "connect" button. This button assigns the user to a simulated environment in which they can interact with a robot; details of robot assignment are discussed further in Section 3.2.4. Once assigned, the user will be able to see the robot in the visualizer and command it through the interaction

panel. They will remain connected until they have completed all their desired interactions.

At the conclusion of their interactions, disconnection is automatic upon window close, even if not explicitly requested through the red "disconnect" button. This design minimizes resource wastage due to users who are not actively engaged.

Robot scene visualizer. The robot scene visualizer is the main element of the page. This allows the user to inspect a live rendering of the scene and determine the most interesting tasks and the most useful feedback for the robot.

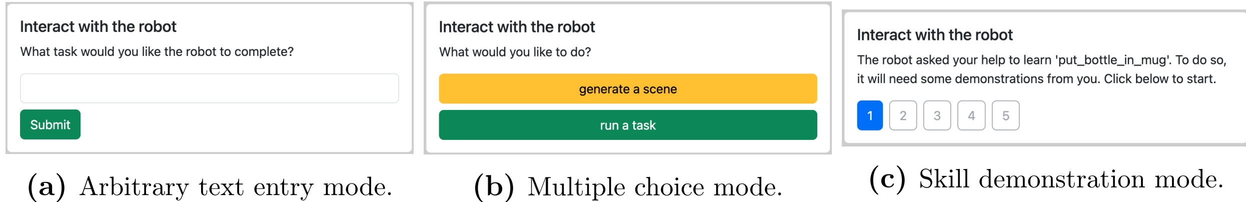
This visualizer is implemented using Meshcat, an open-source scene rendering webserver. Meshcat provides mouse controls to let the user zoom or pan the scene and accurately perceive relative object poses. On each timestep in the simulated physics world, the system tracks changes to object poses and renders any changes in Meshcat.

As each Meshcat server instance only displays one scene, each HaLP 2.0 instance automatically creates a new Meshcat server for itself. Each server uses a different port, and only the relevant URL is provided to the interface. Finally, the user interface embeds the rendering into the webpage via an HTML `iframe`.

Interaction panel. The interaction panel prompts the user for robot instructions and feedback. It is activated whenever the system seeks human input.

To respond, the user only needs to complete the form in the panel. Whereas a terminal screen can typically only support text input, a web-based form is much more diverse. As shown in Fig. 3.6, this panel will show either a free-response text field or a multiple-choice selection depending on the nature of the requested feedback, which is specified in the message protocol (Section 3.2.3).

Additionally, the interaction panel launches the skill demonstration interface when required for the system to learn a new skill (Section 3.2.2). This dialog is depicted in Fig. 3.6c: the system can prompt the user for a collection demonstrations, which the user then provides through the pop-up demonstration interface.



(a) Arbitrary text entry mode. **(b)** Multiple choice mode. **(c)** Skill demonstration mode.

Figure 3.6: Different input modes supported by the web-based user interface.

As such, the interaction panel initiates all interaction between the human user and the robot system. To ensure the user is aware a response is being requested, the entire webpage darkens with the exception of this panel, drawing the user's attention.

System status log. The system status log gives the user live updates on HaLP 2.0's activity. Some models in the system may take considerable time to initialize or run. Therefore, feedback is important, not only for researchers to determine that the system hasn't entered an infinite loop or other fatal bug, but also for end users whose attention span may be more limited. Providing these updates gives a reassurance that the system is functioning as normal.

The status log is populated with milestone events such as the start and end of running each foundational model. These events are published via the message protocol described in Section 3.2.3 so that they may be displayed to the user.

Layout. These various components are arranged on the webpage using Bootstrap [35], a common open-source CSS framework for responsive web design. Responsiveness allows users on mobile devices to also have a pleasant experience interacting with the system. While we currently expect the current predominant use case to involve researchers and crowd-sourced users on desktops or laptops, we design this interface with the vision of it being readily accessible to users on a variety of platforms.

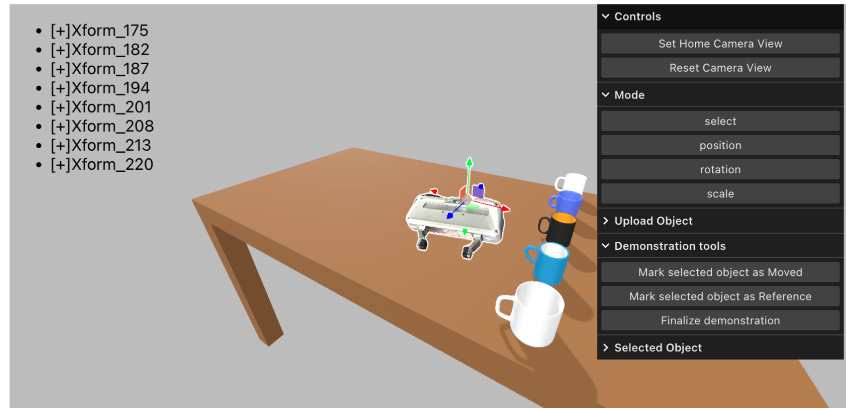


Figure 3.7: Sample screenshot of the skill demonstration interface.

3.2.2 Skill demonstration interface

The skill demonstration interface is a separate web application for demonstrating object rearrangement skills. Its primary purpose is to make it easy for users to move, rotate, and align objects in arbitrary poses, so that those objects and poses may be recorded for an imitation learning algorithm. It is versatile, running directly out of a web browser; we show a screenshot in Fig. 3.7.

The interface is based on a ThreeJS application developed for scene construction in the Improbable AI lab. We load a set of objects from the Objaverse collection [36] and allow the user to manipulate objects as they wish, without being hindered by any simulated gravity or collisions as we do not provide a physics server.

Once the user is happy with the orientation of their objects, they select the two objects involved in the manipulation; for example, the robot gripper and the mug being grasped. The object meshes and their poses are sent to the orchestrator via HTTP POST to be saved by the *demonstration* module. To respect size limits on HTTP requests, the mesh data is compressed using GZIP before being sent.

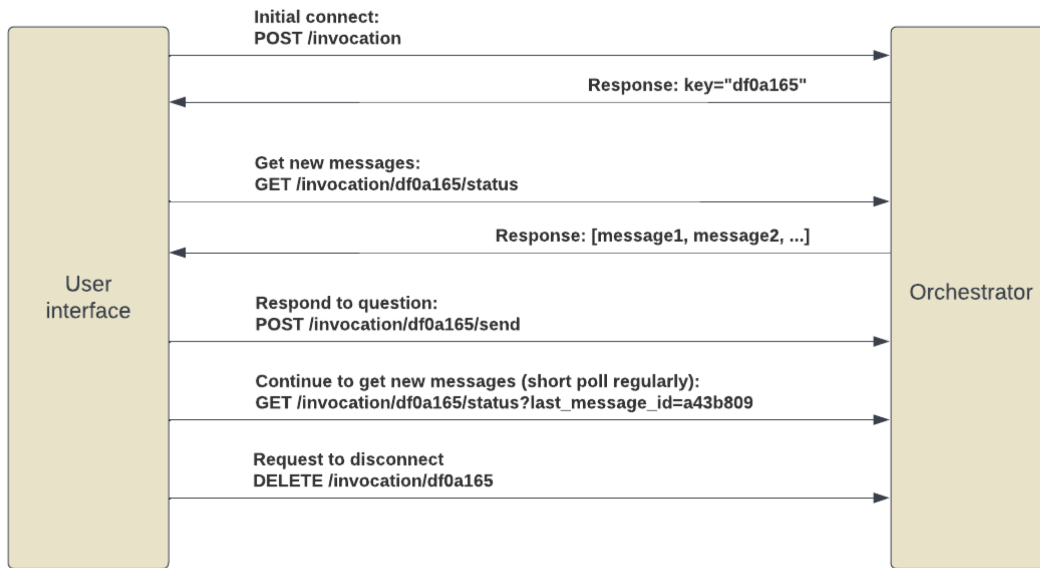


Figure 3.8: HTTP protocol between the user interface and the orchestrator.

3.2.3 Message protocol between orchestrator and robot system

Communication between the orchestrator and each HaLP 2.0 instance is facilitated by inter-process message-passing. The user interface initiates then communicates with the orchestrator over HTTP, where each request either sends a user action to the robot system, or asks for a feed of the latest events that have occurred.

More specifically, the web interface sends a POST request each time the user takes an action. Meanwhile, it also uses short polling with GET requests to retrieve events from the robot system. This protocol is illustrated in Fig. 3.8. In these requests, each event is described as a *message*; a message is a typed data structure consisting of multiple fields that fully describe the event. There are different *classes* of messages, corresponding to different types of events. These include prompting the user to answer a question, or notifying the interface that the Meshcat visualizer is ready.

The same message datatype is used to communicate between the orchestrator and HaLP 2.0; the orchestrator acts as an intermediary, and as a buffer storing the latest messages from HaLP 2.0 until they are requested by the interface. All messages include a timestamp, a class

<table>
<tr><th>Class</th><th>Direction</th><th>Description</th></tr>
<tr><td><code>visualizer</code></td><td>HaLP 2.0 to interface</td><td>The Meshcat robot scene visualizer is ready.</td></tr>
<tr><td colspan="3">URL: the visualizer URL that should be embedded into the interface webpage.</td></tr>
<tr><td><code>question</code></td><td>HaLP 2.0 to interface</td><td>System has a question for the user.</td></tr>
<tr><td colspan="3">Prompt: a string containing the question being asked.
Choices: an optional field containing multiple-choice options. Each choice has a display color, a text description, and a unique identifier.
Allow Empty: a boolean field indicating whether a blank response is valid.</td></tr>
<tr><td><code>response</code></td><td>Interface to HaLP 2.0</td><td>User has responded to the question.</td></tr>
<tr><td colspan="3">Question ID: the question to which this response corresponds.
Data: a string containing the user's response. Must match one of the response identifiers if the question was multiple-choice.</td></tr>
<tr><td><code>stage-begin</code></td><td>HaLP 2.0 to interface</td><td>A new milestone stage has started.</td></tr>
<tr><td colspan="3">Stage: a text description of the milestone, for display in the system status log.</td></tr>
<tr><td><code>stage-end</code></td><td>HaLP 2.0 to interface</td><td>The previous milestone stage has ended.</td></tr>
<tr><td colspan="3">Stage: a text description of the milestone, for display in the system status log.
Elapsed: the running time that elapsed during this stage, in seconds.</td></tr>
<tr><td><code>skill-request</code></td><td>HaLP 2.0 to interface</td><td>System requires a new skill to be taught.</td></tr>
<tr><td colspan="3">Skill Name: the name of the skill being requested.
Num Keys: the quantity of demonstrations being requested.</td></tr>
<tr><td><code>skill-demo</code></td><td>Interface to HaLP 2.0</td><td>User has provided a set of demonstrations.</td></tr>
<tr><td colspan="3">Request ID: the skill request to which this demonstration corresponds.
Skill Name: the skill being demonstrated.
Keys: the list of keys returned by the demonstration interface.</td></tr>
<tr><td><code>poison</code></td><td>Interface to HaLP 2.0</td><td>User has requested to disconnect.</td></tr>
</table>

Table 3.3: Protocol of messages and their fields, used to communicate between the robot system and the user web interface.

field, and an ID field, which is a unique identifier for the message. The full specifications for the message classes and their fields is presented in Table 3.3. For each message, its class is checked to route it to the appropriate interface component for handling. To receive up-to-date messages, the interface issues a GET request with the ID of the latest message it has received. The orchestrator responds with an ordered list of all subsequent messages.

3.2.4 Concurrency management

As individual users should be able to interact with distinct environments, we use a pool of processes running separate instances of HaLP 2.0. These processes communicate with the orchestrator via the previously-defined message protocol.

However, the simulator and the neural network models are all very resource intensive. To concurrently run models such as Neural Descriptor Fields, GraspNet, and BERT, in addition to the IsaacSim physics simulator, relies on access to a large amount of GPU memory. Empirically, running the full system requires over 14 GiB of GPU RAM, more than one-fourth the capacity of a NVIDIA RTX A6000 GPU.

Therefore, it is imperative to limit concurrent use. We achieve this by programmatically throttling the system to an allotted quota of simultaneous users. Moreover, as system initialization can be lengthy to load the various models, we pre-emptively start as many processes as possible and aim to assign new users to instances that are already fully initialized. Specifically:

- Whenever there are fewer instances running than the maximum capacity, a new one is started. Its state is recorded as “uninitialized”. If there is any user waiting for an instance, they are immediately assigned this instance and it enters the “assigned” state.
- Otherwise, it enters the “initialized” state once it is ready to take human input. Once a user clicks the “start” button on the web interface, they can receive this instance and it enters the “assigned” state.

- Instances in the “assigned” state will continue running until killed by the user either manually or by timeout, which occurs after 10 seconds without any short polling request. At that time, the instance is sent the `poison` message and allowed to terminate gracefully.

Correspondingly, the full lifecycle of a user session is as follows.

- Users who navigate to the web interface are ignored by the orchestrator until they click the “start” button. At that time, they enter the “waiting” state.
- If an “initialized” instance is available, they immediately receive that one. As a second option, they receive an “uninitialized” instance if one is available. The user then enters the “running” state.
- The user will stay in this state until they disconnect, at which point they will enter the “exited” state. The user’s session will eventually be garbage-collected.

Together, these mechanisms try to ensure that as many instances are available as possible; instances are pre-emptively initialized ahead of time before users arrive so that they do not waste time waiting for the startup sequence. As such, the system is able to maximize its throughput.

Chapter 4

Evaluation of System Modules

In this chapter we study the performance of our system as a result of its new designs. For our system, we provide three main dimensions of analysis: the performance of the scene perception algorithms (Section 4.1), the performance of the new motion planner (Section 4.2), and the capacity for large learning modules to perform lifelong learning (Section 4.3).

4.1 Semantic segmentation performance

Most vision models are trained to solve perception tasks in real-world scenes. While our ultimate goal is to see HaLP 2.0 thrive in the real-world, large-scale experiments based in simulations may only be reliable if vision models perform well on simulated scenes too.

HaLP 2.0 supports NVIDIA Isaac Sim as its main physics simulator. It was hypothesized that its higher-quality renderings lead to improved semantic segmentation performance, when compared to the PyBullet simulator used in the predecessor. In this section we evaluate this choice by comparing the accuracy of vision models using renderers from each simulator.

To do so, we quantify metrics for the two main types of failure modes in perception:

1. **Misclassified object.** The semantic labeler assigned the incorrect object category.

2. **Missing object.** The segmentation model did not detect the object at all.

To address the first failure mode, we define a concept we call *confusion*. Consider two object categories: a ground-truth category x of interest to a planner, and a distractor category y distinct from x. Suppose a scene contains objects of both categories. We define the confusion of x as y to be the probability that, when the planner asks for x, the semantic labeler erroneously returns the object of category y. Informally, confusion is a measure of how likely category x can suffer from an "imposter" of category y.

Naturally, a simulator that minimizes confusion has better system performance. We empirically measure confusion by generating sequences of random scenes containing precisely one object from each of the two categories, taking image crops from ground-truth segmentations, and observing the CLIP misclassification rate in the `find` method. We perform 100 trials each for all pairs of categories out of bottles, bowls, containers and mugs.

Table 4.1 presents the pairwise confusion scores across a set of categories. We note that in some cases, the objects may return a low similarity score on all object classes below the detection threshold, and therefore return no match at all. We also consider that a failure of the detection system. The results show that for this object classification task, scenes rendered by Isaac Sim may generally lead to improved performance than PyBullet, especially in the case of mug distractor objects.

We also evaluate the second failure mode for Detic object *detection*, reporting results in Fig. 4.1. We find that segmentation performs comparably on Isaac Sim in cases where it is already good on PyBullet (Fig. 4.2), and can also outperform in certain categories such as bottles. We examine some failure cases, showing representative examples in Fig. 4.2. Specifically, in these cases objects nevertheless appear out-of-distribution with respect to real-world items. The ShapeNet object itself may be unrealistic (Fig. 4.2a); the default Isaac Sim material may also look unrealistic (Fig. 4.2b). We note that custom textures are an Isaac Sim feature that we have not yet availed. Therefore, these results show only a baseline level of performance; with additional enhancements to textures and lighting, we expect further improvements are well within reach.

Ground truth category x	Distractor category y								Not found	
	bottle		bowl		container		mug			
	Isaac	PyB	Isaac	PyB	Isaac	PyB	Isaac	PyB	Isaac	PyB
bottle	-	-	**1%**	3%	**6%**	14%	**11%**	44%	**7%**	10%
bowl	**1%**	2%	-	-	**1%**	3%	**10%**	16%	**12%**	29%
container	**0%**	**0%**	**0%**	2%	-	-	**5%**	23%	24%	**19%**
mug	**1%**	3%	**1%**	9%	**0%**	1%	-	-	**2%**	12%

Table 4.1: Pairwise confusion scores for semantic labeling under different simulator renderings.

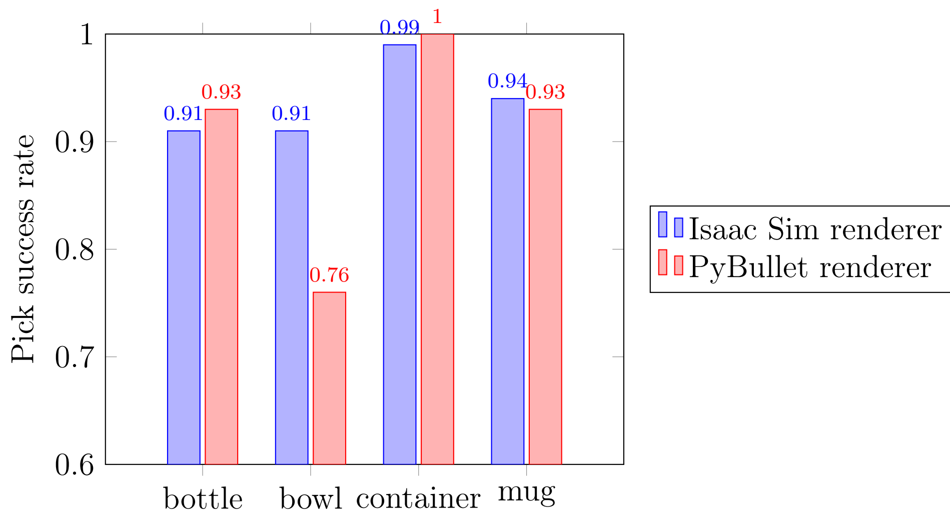
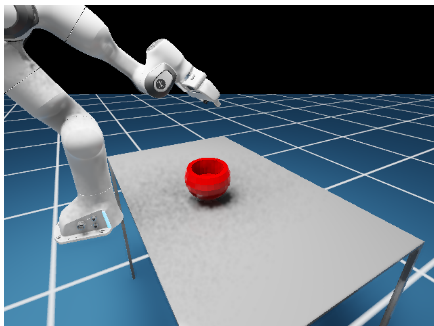
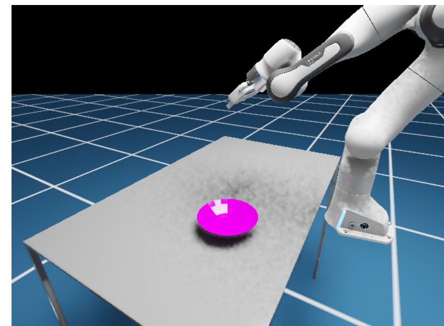


Figure 4.1: Object detection success rates under different simulator renderings.



(a) ShapeNet bowl that doesn't look like a bowl.



(b) Bowl with unrealistically shiny texture.

Figure 4.2: Sample Isaac Sim images where Detic fails to detect the object.

4.2 Collision-free motion planning

Another new addition to the system is cuRobo [29], a collision-free robot motion generator. HaLP 2.0 provides cuRobo as an alternative to other motion planning algorithms such as differential inverse kinematics, which is provided by the Orbit library [28]. Qualitatively, cuRobo provides several advantages. These include:

- **Collision-free in cluttered scenes.** As the primary purpose of cuRobo is to provide collision-free motion plans, we can rely on it to control a robot arm in a cluttered scene. Simpler algorithms for inverse kinematics may disregard obstacles and return a path that collides with the clutter.

- **Waypoint-free.** For our collision-prone controllers, we use a sequence of manually-designed waypoints to help guide the robot gripper into grasping and placing poses. These waypoints have generally been successful at avoiding collisions with the object of interest. However, hardcoded waypoints are naturally not robust. Collision-free planners such as cuRobo allow us to eliminate the need for waypoints.

- **Execution-free pose feasibility.** The cuRobo algorithm detects whether the target pose is infeasible instead of returning an incomplete and unsuccessful plan. This is unlike the differential inverse kinematics controller, which will continuously output robot actions. Detecting feasibility ahead of time is helpful for avoiding unnecessary collisions, and may also be used to help in pose selection when multiple target poses are valid. We use this attribute with Contact GraspNet [13], which generates several suitable grasp poses.

To measure the degree of improvement, we compare cuRobo and the Orbit differential inverse kinematics controller in a sequence of grasping tasks. In each task, we generate a scene containing a single object, and use each planner with Contact GraspNet to pick up the object. We run trials over the bottle, bowl, container and mug object classes; Fig. 4.3 shows

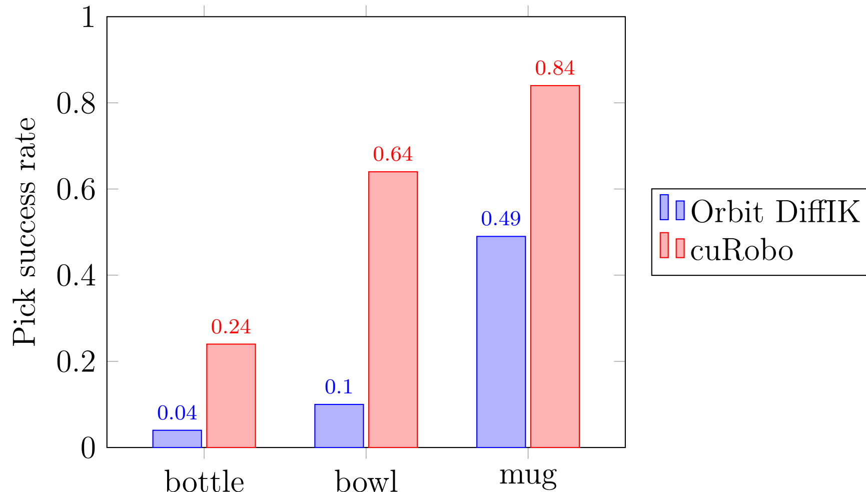


Figure 4.3: Pick object success rate using various motion planners.

the rate of success of each controller over 100 trials. It is evident that cuRobo's performance is far above that of the differential inverse kinematics controller, even in these uncluttered scenes. This improvement can be attributed to cuRobo's ability to select motion plans that avoid colliding into and disturbing the object to pick.

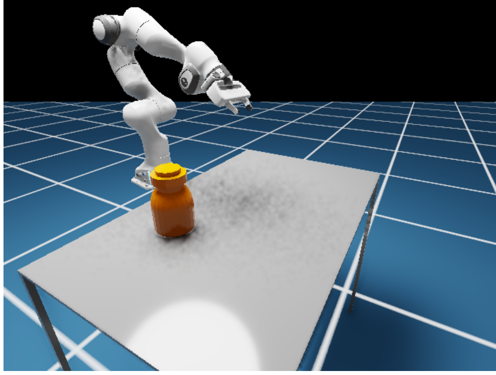
Nevertheless we note that cuRobo's success rate is low in some object categories. We attribute this to the following causes:

- Some ShapeNet objects do not admit valid grasps. For example, bottles may be too wide for the robot gripper (Fig. 4.4a). There are also many bowls that are "full"; that is, filled with a solidified "soup". These bowls also cannot be grasped by a robot gripper.

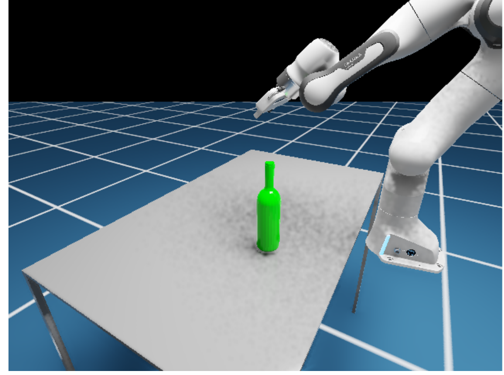
- Some objects can be grasped but difficult to lift without slipping. For example, bottles have narrow bottlenecks (Fig. 4.4b), and grasps on that bottleneck are difficult to maintain. If the bottle falls out of the grasp during the trial, it is counted as a failure.

4.3 LLMs as lifelong learners

In this section we present the design and results of the experiments used to evaluate large language models at lifelong learning. This is the primary function of HaLP 2.0: to facilitate



(a) A bottle too wide to be grasped.



(b) A bottle too narrow to be lifted.

Figure 4.4: Sample ShapeNet objects that induce failure modes in cuRobo grasping.

the accumulation of skills to create an increasingly capable agent over time. To evaluate this, we create progressions of tasks that form a curriculum for learning skills that can be reused in new task contexts, and demonstrate that large language models are capable of effective skill construction and reuse.

4.3.1 Experiment design

Although HaLP 2.0 is a full-stack pipeline for performing LLM-guided robotic manipulation tasks, a direct evaluation on this system does not reflect the performance of the language model planner as its components are prone to failure. We refer to [9] for the results of evaluating the HaLP pipeline and some of its modules. We wish to conduct a focused analysis of the language model's lifelong learning ability.

Yet this itself poses many challenges: whether a skill is *appropriate* is in itself an ill-defined concept. For instance, consider a scenario in which the robot is asked to pour out the contents of a mug. A seemingly reasonable skill request in this case may be "rotate mug". In another scenario where a robot is asked to place a mug upside down on a rack to dry, "rotate mug" sounds like a valid and wholly relevant skill. But the geometric configuration of the mug is entirely different in this case. Fundamentally, this skill is *underspecified* and is not teachable or actionable.

A plan consisting of valid tasks may also be underspecified as a whole. For the same pouring task, the robot may need to start with "grasp mug" as its first skill. This is seemingly correct until the robot holds the mug by the rim; only at that point do we realize we needed to specify a grasp by the handle.

These levels of detail are subtle, subjective, and difficult to evaluate for correctness. A level of detail that may seem appropriate in one case may seem absurd or problematic in another. We posit that evaluating the correctness of an effectively symbolic task plan is challenging. Instead of analyzing plan correctness, we focus instead on the system's effectiveness at skill reuse. We defer to works such as [6] for a discussion of the planning abilities of language models, opting to focus on the *learning* aspect of our system. The discussion of evaluation's limitations is continued in Section 5.1.

To do so, we crowd-source a large collection of natural scenes and tasks on the Prolific research platform [37]; we clarify that this study was approved by the Institutional Review Board. With this dataset, we perform the following evaluations.

A quantitative evaluation of how effectively skills are created and reused. This evaluation measures whether the large-language model can verify that the skills it proposes are useful.

In particular, we query the model with each scene-task pair twice. On the first run, we allow it to make arbitrary calls to `learn_skill`, effectively assembling an entire symbol universe for a symbolic task plan. On the second run, we add the new symbols to the skill library and provide a new language context. We expect that well-selected skills will all be reused in this re-run, and no additional skills should be requested.

A qualitative characterization of the types of skills being proposed. This evaluation considers longer task progressions that enable the system to accumulate and reuse skills over multiple tasks. We choose sequences of tasks of increasing difficulty, sharing common scene elements so that we can observe the system's behavior. We present some of these

Metric	Score
Task skill-sanity	88.5%
Task skill-completeness	86.2%
Task skill-efficiency	100.0%
Skill-wise sanity	92.4%
Skill-wise efficiency	100.0%

Table 4.2: Quantitative metrics for the evaluation of skill reuse in large language model lifelong learning.

sequences below.

4.3.2 Metrics for lifelong learning performance

Our crowd-sourcing efforts yield a total of 71 usable scenes with 213 corresponding tasks. The details of the survey prompts and a categorized breakdown of these scenes is included in Appendix A. We take a sample of these tasks to run the quantitative evaluation.

We specifically measure the following metrics in this evaluation:

- **Task skill-sanity.** The fraction of tasks in which all skills requested on the first run are actually called.
- **Task skill-completeness.** The fraction of tasks in which the second run does not require new skills absent in the first run.
- **Task skill-efficiency.** The fraction of tasks in which the second run uses all of the skills learned and used in the first run.
- **Skill-wise sanity.** The fraction of requested skills that are actually called.
- **Skill-wise efficiency.** The fraction of requested skills that are used in both runs.

Clearly an efficient system should score highly on all five metrics. We report the results of this evaluation in Table 4.2. Notably, the system achieves perfect skill-efficiency, indicating that all skills used in the first run are deemed necessary by the second run. For the other metrics,

the overall performance is over 85% at a per-task level, with skill-wise metrics scoring over 90%. Among those tasks where the system failed, we highlight some key findings.

Extraneous unused skills. Sometimes the language model is "perfectionist" and attempts to perform extraneous actions that are not required to minimally complete the task. These skills are "cleanup" actions that are recommended but which the model chooses not to perform. For example:

- In a task "turn on faucet", the model learns the skill `rotate_faucet_off`.
- In a task "wash dishes", the model learns the skill `put_away`.
- In a task "put clothes away", the model learns the skill `sort_clothes`.

Most skills that fail the skill-sanity metric have skills of this form. In these cases, it appears that the learning of additional skills is intentional; they are neither required nor used for the task. A similar trend explains the skill-completeness metric. As language model rollouts are stochastic, there are times when these cleanup actions are not suggested until the second run. This leads to an apparently lowered skill-completeness score.

Redundant skill-learning. Very occasionally, the language model learns skills that are redundant given the existing ones. This comes in two forms; first, GPT-4 emphasizes safety, and may choose to learn special skills for fragile or hazardous items such as mugs and knives. Context-engineering to highlight that built-in skills are already cautious do not successfully eliminate this behavior. Second, GPT-4 learns specializations, such as storing cookware instead of generic placing. These specializations are indeed redundant, but subject to context-engineering; we discuss mitigation further in our qualitative studies in Section 4.3.3.

Lastly, we summarize the types of skills learned in this evaluation. As skills function as imperative commands in a symbolic plan, the first word in each skill is a verb. We show the distribution of these verbs in Table 4.3.

Skill type	Verbs used
Concrete low-level action, learnable as a primitive	close, fold, grasp, hang, move, navigate, pour, press, put, remove, rotate, stack, tilt, turn
Higher-level action, a combination of primitive actions	add, dispense, dry, dump, dust, fill, fluff, rinse, scrub, smooth, spray, spread, straighten, submerge, throw, tuck, unplug, wipe, wring
Abstract action, requiring extensive planning	activate, adjust, apply, deactivate, drain, load, sort, start, store, tidy, use, vacuum, wash

Table 4.3: An approximate categorization of the types of skill verbs learned by the large language model.

4.3.3 Characterization of large language model behavior

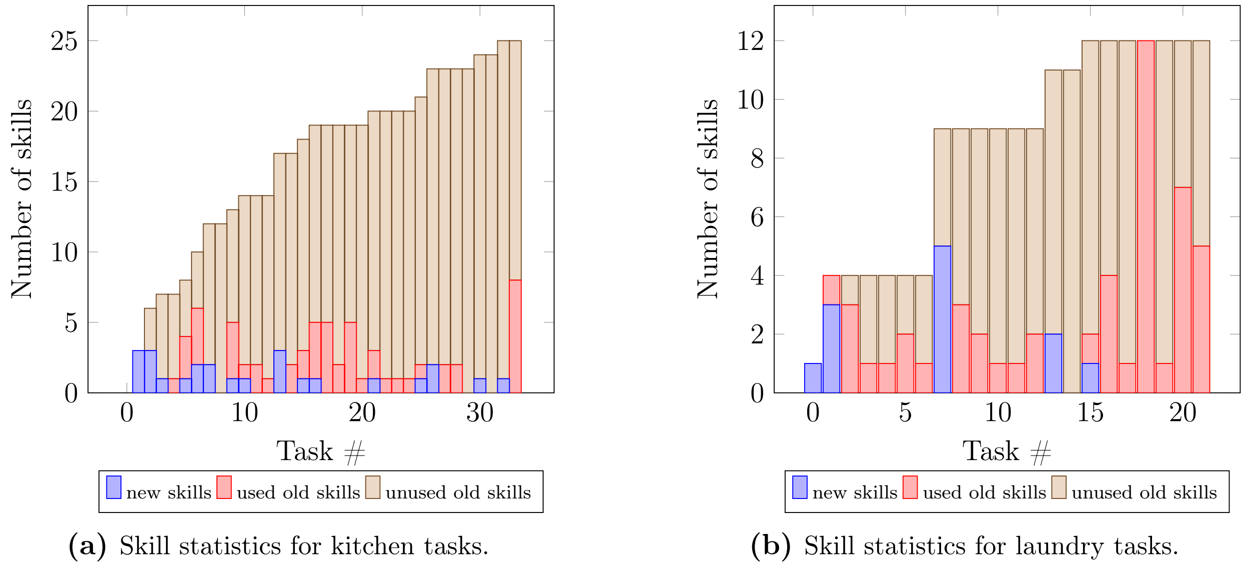
For the qualitative characterization, we create progressions by grouping scenes by category, such as kitchen or laundry scenes. Tasks are ordered in the suggested difficulty rating given by the crowd-source survey participants. Emergent behaviors in this experiment are surprising and demonstrate extraordinary attempts to use skills effectively. For example, on a kitchen task to load a dishwasher, the language model tries to repurpose a skill previously learned for opening a refrigerator—a similar kitchen appliance.

```
# Pick the dirty dish
pick(dish_id)
# Open the dishwasher - could repurpose the "open_refrigerator" skill
# for opening things
open_refrigerator()  # Repurposing this skill to open the dishwasher
```

Even more astoundingly, a kitchen task to "take stems off tomatoes" after previously learning how to open a milk bottle yielded the following code snippet:

```
# Assuming "open_milk_bottle" simulates the twisting/pulling motion
# needed to remove a stem.
detach_stem = open_milk_bottle  # Renaming the function for clarity
```

These examples show that the language model is capable of adapting skills to novel contexts



(a) Skill statistics for kitchen tasks.

(b) Skill statistics for laundry tasks.

Figure 4.5: Skill growth and learning frequency for tasks in sample task progressions.

outside of where they were first seen. We include some more examples of skill reuse interactions in Appendix B. Generally these interactions demonstrate that the system is effective at remembering and reusing skills even over long task progressions, appropriately selecting skills that are suitable for the task. We find that the system continually reuses skills as the size of the skill library increases, requiring fewer new ones as some tasks become repetitive. We summarize the use and growth of the skill library of these sequences in Fig. 4.5.

HaLP 2.0 allows for human-in-the-loop feedback to refine code before it is executed, and we remark that this is very necessary in order to obtain symbolic plans that are syntactically correct. We empirically observe that GPT-4V is susceptible to ignoring explicit docstring directions at times, leading to function calls that violate the prescribed specifications. We enumerate some of these below.

- Using the `find` skill to return a list of *all* objects matching a filter, even though we explicitly specify that it only returns a single object.

- Using `learn_skill` to obtain a function, only to immediately overwrite it with a `def` function-definition of the same name and an empty body. It appears that the model is attempting to provide a stub to prompt the human for an implementation, even

Type of learn_skill docstring	# skills	Proportion of unnecessary skills	
		GPT evaluation	Manual evaluation
Instruction to avoid call	26	19%	15%
No instruction to avoid calling	41	54%	27%

Table 4.4: Proportion of unnecessary skills learned under differently engineered docstrings.

though we explicitly tell it not to do so.

- Attempting to learn "skills" that observe the scene rather than manipulate it, such as `is_dirty` for cleaning tasks. Our interface specifies that skills should be used to command robot actions. The interface provides `find` which should be used instead to filter for the desired objects.

- Learning redundant (unnecessary) skills, such as `store_cookware` or `pick_knife` when a simple `pick` or `place` would suffice. We find that engineering the docstring for `learn_skill` has a noticeable effect on the proportion of redundant skills, by instructing the language model not to learn skills unless absolutely necessary; the docstring is included in Appendix B. In our post-hoc analysis, we ask GPT-4 to compare method docstrings and identify skills that are merely more specific versions of other skills. We also perform this evaluation manually as GPT-4 may be overaggressive as it is not familiar with our skill framework, finding redundancy in skills that sound similar but are actually very different, such as the pair `turn_on_faucet` and `turn_on_kettle`. We report the results in Table 4.4; they show that trials with engineered contexts yielded far fewer redundant skills, such as in the interaction presented in Table B.1. On the other hand, when not instructed to avoid skill-learning, the language is much more liberal, yielding much more skills that were redundant. We refer to the rollout in Table B.2 as an example.

We provide feedback whenever the language model writes code that violates any of these rules. This is only to ensure that the code being produced is syntactically valid and therefore

parseable by our verifier; we suggest that prompting models to write perfect code without human interaction is out of scope. We do not provide any feedback other than correcting syntax errors.

Nevertheless, we note that these failures are relatively rare, occurring in less than 10% of tasks, and that the vast majority of new skills are meaningful and productive. We believe our results show that language models are promising for more complex planning tasks and for lifelong learning, and that once we have the ability to train policies for these new skills, it is possible to create very capable robotic agents.

Chapter 5

Limitations and Further Work

Our studies of HaLP 2.0 provide insights to a number of questions about the performance of large language models in lifelong learning, as well as foundation models for perception and control in simulated environments. However, some questions yet remain unanswered, some being too complex to understand with the current framework. We also open a plethora of avenues for further exploration, some of which we discuss below.

5.1 Evaluation of skill and plan correctness

In Section 4.3 we evaluated lifelong learning based on the system's ability to construct a set of skills that, on an independent run, is classified as sufficient by a large language model. We remark that the results are highly dependent on the distribution of tasks used in evaluation: inevitably, there may be some tasks that the system will find easier or harder than others.

Accordingly, we do not base our evaluation on tasks and scenes of our own creation. We expect that crowd-sourced tasks will largely eliminate any bias in the distribution of the evaluation dataset; moreover, they will be more realistic in evaluating the potential for HaLP 2.0 to actually be useful in a real-world deployment. As such, we suggest that the results of this evaluation are indicative of the system's potential.

Nevertheless, we are limited to these symbolic evaluations: attempting to execute task

plans in all these crowd-sourced scenes is largely infeasible. It is not only time-consuming to train all the requisite policies and to collect the demonstration data, but also extremely error-prone in execution. As was discussed, evaluating the correctness of a symbolic plan can also be problematic due to the subjectivity of understanding the nuance of under- and over-specified skills. Our study focuses on the objective evaluation of skill reuse; we remark that a rigorous evaluation mechanism for skill correctness is challenging, but may be especially insightful.

5.2 System component extensions

The modularity of the HaLP 2.0 system makes it extremely easy to add new extensions. We believe the addition of new alternative implementations to these modules can continue to expand the system's capabilities. We enumerate some of these possibilities below.

New types of learnable skills. So far, we have used Neural Descriptor Fields for pose inference and object relational rearrangement with category-conditioned models [14, 15]. We additionally recommend Local Neural Descriptor Fields [38], a category-agnostic variant based on local "part" geometries, which may further reduce the volume of human interaction required to train the system. Diffusion-based policies may also be effective at handling situations where there are multiple valid task plans [39]. At the same time, skills are versatile enough to also allow the inclusion of closed-loop controllers such as reinforcement learning agents.

New types of model pipeline. HaLP 2.0 currently relies on human feedback between consecutive steps of an execution plan. It is feasible to draw inspiration from works such as Inner Monologue [8], which provide automated feedback mechanisms. This would allow HaLP 2.0 to become even more autonomous. Feedback mechanisms we may consider include updated scene descriptions in symbolic or natural language format, or in the case of vision-

language models, taking new camera images of the scene.

5.3 Even larger scale studies

HaLP 2.0 is designed for scale. As a full-stack system with a user-friendly interface and built-in support for concurrency, it is ready to be deployed to large crowd-sourcing data applications. This enables novel studies of the system's end-to-end performance on a large assortment of scenes, tasks, and skillsets. We believe that such a study would be even more illuminating about the lifelong learning capabilities of large language models, and the potential future applications of the system as a whole in human assistance settings.

At the same time, HaLP 2.0 can also be used for pure data collection to generate massive robot trajectory datasets. These datasets can consist of LLM-guided robot manipulation tasks, and be used downstream in end-to-end training of a new generation of behavior cloning and reinforcement learning agents that can perform increasingly complex tasks.

Chapter 6

Conclusions

HaLP 2.0 is a modular and scalable system for integrating robotic manipulation algorithms with the lifelong learning capabilities of large language models. We provide a user-friendly interface for running reproducible large-scale experiments and commanding robots through natural language.

Our evaluations show that HaLP 2.0 provides module implementations that improve system performance over its predecessor HaLP. We also run a focused study on large language models, showing that they have significant lifelong learning potential to be harnessed. But, the journey doesn't end here: we design HaLP 2.0 to be extensible, because the possibilities are many. Our system is ready to incorporate new skills, new pipelines, and even larger scales of crowd-sourced studies, to probe the limits of what large language models can do, however *improbable* those goals may be. Lifelong learning can enable our robots to acquire skills of ever-increasing complexity, and once ready for the real-world, to become the ultimate human assistant.

Appendix A

Scene and Task Crowd-sourcing Survey

Our crowd-sourcing survey for collecting a scene and task dataset was hosted on Prolific [37]. This study was approved by the Institutional Review Board of the Massachusetts Institute of Technology protocol E-5443. Below we include the exact survey text and some representative statistics about the types of scenes submitted.

A.1 Survey content

> *Teach our robots to be your assistants.*
>
> We are teaching some robots to be good human assistants. Imagine a robot that could help you around your house: in your kitchen, in your living room, in your laundry, in your yard! We need your help to tell us, what sorts of things you'd like your robot to do.
>
> For example, here's something that I'd like! Take a look at this kitchen. Three things I might like a robot to do are:
>
> - Put the small red bowl into the orange bowl on the shelf.
> - Wash the pink mug.
> - Tidy up everything in the top-right shelf.

Figure A.1: Sample kitchen image shown to survey-takers as an example of a robot scene.

So, what would you find helpful? Please:

- Give us a photo of a place you think a robot would be helpful. You can either find a photo online, or take a photo of somewhere around you.
- Look at your photo and imagine there's a robot there! Tell us three different tasks you think a robot could help you with.
- Please give your tasks in order of how hard you think they are—easiest first!

Please make sure your tasks only involve things you can see in the photo. The robot must be able to see all the tools it will need! That is:

- If your photo is in the living room, ***don't*** ask it to fetch a coke from the fridge. The robot can't see the fridge, and it can't see the coke.
- If your photo is in a completely empty kitchen, ***don't*** ask it to make you breakfast. The robot can't see the ingredients, and it can't see any frying pans.
- If you want the robot to clean something, **make sure it can see** cleaning supplies.

That's all we've got to say! Now it's your turn.

A.2 Scene characteristics

The collected dataset contains 100 responses, each consisting of an image and three tasks. We manually process those responses to filter out ones with obviously invalid tasks that are infeasible in the scene. Most commonly, this involved a cleaning task without any visible janitorial supplies.

Of those remaining, we categorize the tasks depending on the scene type, with the expectation that similar scenes may require similar skills. These statistics are shown in Table A.1. The most common tasks involved loading a dishwasher, putting clothes into a washing machine, and tidying an unorganized assortment of toys and tools.

Category	Number of scenes	Number of tasks
Bathroom	2	6
Bedroom	7	21
Floor only	1	3
Kitchen	21	63
Laundry	8	24
Living room	16	48
Puzzle	1	3
Outdoor	2	6
Shelf rearrangement	8	24
Study desk	5	15
Total valid	71	213
Total invalid	29	87

Table A.1: Number of scenes and tasks of each category in the crowd-sourced dataset.

Appendix B

LLMs in Long Task Progressions

Here we present some of the tasks evaluated in the longer task progressions along with the skills learned and used by the language model. We begin with tasks corresponding to scene belonging to the *kitchen* category in Table B.1. Due to confidentiality obligations we are unable to reproduce any of the submitted scene images. This dataset was obtained with the following docstring for `learn_skill` directing its use to be avoided.

> `learn_skill(skill_name: str,docstring: str): Callable`
>
> Adds a new skill to the current list of skills. Skills must be used to move objects. Do not use a skill to find objects. Attempts to find or observe an object will fail. Skills have no return value. **Only ever learn a new skill if it is impossible to achieve using existing ones. For example, if you already know how to pick up items, then do not additionally learn how to pick up a bottle.** You can assume that the skill implementation will be automatically provided. Do not provide any implementation of your own for skills that you learn. For example, to open a drawer, you can use:
>
> ```
> learn_skill(skill_name="open_drawer", docstring="Pulls a drawer open.")
> open_drawer(drawer_id)
> ```

We note the following insights about this dataset:

- Many new skills are learned within the first tasks. As the skill library accumulates, many skills are continually reused, such as `open_cabinet_door` (task 8) or `turn_on_faucet` (task 11). Skill learning can become less frequent in later tasks as the agent's capabilities grow.

- As discussed in Section 4.3.2, some extraneous skills are learned. An example of this is `store_cookware` (task 7), which we see to be an instance of unnecessary skill specialization being performed by the language model. However, comparing to the discussion in Table B.2, these extraneous skills are much rarer.

- Some other skills are similar and may be redundant with each other, although this is model-dependent. Examples include `open_fridge_door` (task 2), `open_cabinet_door` (task 8), and `open_oven_door` (task 14), which all involve opening doors.

#	Task	Skills used
1	put all dirty dishes in the sink	
2	take out bottle of milk and open it	`*close_fridge_door *open_fridge_door *open_milk_bottle`
3	wash dishes	`*fill_sink_with_water *rinse_dish *scrub_dish`
4	take stems off tomatoes	`*cut_stems`
5	put dishes in sink	`fill_sink_with_water`
6	clean the countertops	`*wipe_counter fill_sink_with_water rinse_dish scrub_dish`
7	clean up pots and pans	`*empty_contents *store_cookware fill_sink_with_water rinse_dish scrub_dish wipe_counter`
8	put salt in cabinet	`*close_cabinet_door *open_cabinet_door`
9	place the empty jug into the recycling bin	
10	wash dishes	`*pre_rinse_dish fill_sink_with_water rinse_dish scrub_dish store_cookware`
11	turn on water	`*turn_on_faucet pre_rinse_dish`

continued ...

#	Task	Skills used
12	move the glass cup from the counter to the shelf	close_cabinet_door open_cabinet_door
13	remove the pots and pans from the sink	store_cookware
14	take potatoes out of oven	*close_oven_door *open_oven_door *remove_from_oven
15	throw away all expired food in the refrigerator	close_fridge_door open_fridge_door
16	preheat the oven to 350 degrees	*set_oven_temperature close_oven_door open_oven_door
17	clean the knife and put it away	*turn_off_faucet pre_rinse_dish rinse_dish scrub_dish turn_on_faucet
18	put away clean dishes	close_cabinet_door open_cabinet_door
19	take a styrofoam cup	
20	rinse dishes in sink and load dishwasher	fill_sink_with_water pre_rinse_dish rinse_dish scrub_dish turn_off_faucet
21	dump out all the liquids in the cups before trashing them	empty_contents
22	dump out bowl of cherries from top shelf	*tilt_to_empty close_fridge_door open_fridge_door
23	clean counter	wipe_counter
24	pick up knife and cut up tomatoes	cut_stems
25	turn on faucet	turn_on_faucet
26	load the dishwasher	*close_dishwasher_door pre_rinse_dish
27	remove stove covers	*lift_stove_cover *move_cover_to_side
28	put olive oil in cabinet	close_cabinet_door open_cabinet_door
29	put the sauce bottle into the cabinet above	close_cabinet_door open_cabinet_door
30	put the dishes on the drying rack	
31	turn on kettle	*turn_on_kettle
32	remove one cup from the stack	
33	use the orange towel to dry off the plates in the sink	*dry_plate_with_towel
34	clean the plates with food on them	fill_sink_with_water pre_rinse_dish rinse_dish scrub_dish store_cookware turn_off_faucet turn_on_faucet

Table B.1: Skills learned (marked with asterisk) and used in crowd-sourced tasks proposed for kitchen scenes.

On the other hand, we compare with a version in which `learn_skill` does not instruct the LLM to avoid creating new skills. In particular, the bolded statements from the previous docstring was omitted. In this case, we see several more redundant skills in Table B.2, including `pick_knife`, `pick_cup`, `pick_plate`, as well as `store`-item counterparts for each of these skills.

#	Task	Skills used
1	put all dirty dishes in the sink	
2	take out bottle of milk and open it	`*close_refrigerator *open_milk_bottle *open_refrigerator`
3	wash dishes	`close_refrigerator open_refrigerator`
4	take stems off tomatoes	`*cut_stem *dispose_stem`
5	put dishes in sink	
6	clean the countertops	`*discard_trash *spray_cleaner *wipe_surface`
7	clean up pots and pans	`discard_trash spray_cleaner wipe_surface`
8	put salt in cabinet	`*close_cabinet *open_cabinet`
9	place the empty jug into the recycling bin	`close_cabinet open_cabinet`
10	wash dishes	`*apply_soap *dry_dish *rinse_dish *scrub_dish close_cabinet open_cabinet`
11	turn on water	`*turn_on_faucet`
12	move the glass cup from the counter to the shelf	`close_cabinet open_cabinet`
13	remove the pots and pans from the sink	
14	take potatoes out of oven	`*close_oven *open_oven *pick_baking_tray *place_baking_tray`
15	throw away all expired food in the refrigerator	`close_refrigerator discard_trash open_refrigerator`
16	preheat the oven to 350 degrees	`*set_oven_temperature close_oven open_oven`
17	clean the knife and put it away	`*dry_knife *pick_knife *store_knife apply_soap rinse_dish scrub_dish turn_on_faucet`
18	put away clean dishes	`*pick_cup *pick_plate *store_cup *store_plate close_cabinet dry_knife pick_cup pick_knife pick_plate store_cup store_knife store_plate`
19	take a styrofoam cup	`store_cup`

continued . . .

#	Task	Skills used
20	rinse dishes in sink and load dishwasher	*close_dishwasher *open_dishwasher *start_dishwasher *turn_off_faucet rinse_dish turn_on_faucet wipe_surface
21	dump out all the liquids in the cups before trashing them	*empty_cup discard_trash turn_off_faucet turn_on_faucet
22	dump out bowl of cherries from top shelf	*tilt_bowl close_refrigerator open_refrigerator
23	clean counter	spray_cleaner wipe_surface
24	pick up knife and cut up tomatoes	*cut_tomato *pick_tomato pick_knife
25	turn on faucet	turn_on_faucet
26	load the dishwasher	*add_detergent_to_dishwasher close_dishwasher open_dishwasher rinse_dish start_dishwasher turn_off_faucet turn_on_faucet
27	remove stove covers	*remove_stove_cover
28	put olive oil in cabinet	close_cabinet open_cabinet
29	put the sauce bottle into the cabinet above	*close_upper_cabinet *open_upper_cabinet
30	put the dishes on the drying rack	dry_dish
31	turn on kettle	*turn_on_kettle
32	remove one cup from the stack	
33	use the orange towel to dry off the plates in the sink	*dry_plate_with_towel *pick_orange_towel
34	clean the plates with food on them	apply_soap discard_trash dry_plate_with_towel pick_orange_towel rinse_dish scrub_dish store_plate turn_off_faucet turn_on_faucet

Table B.2: Skills learned (marked with asterisk) and used in crowdsourced tasks proposed for kitchen scenes.

Additionally, we show a sample interaction sequence in some laundry-room scenes in Table B.3. We note that once the system has accumulated sufficiently many skills, the instruction to "tidy up mess" causes the model to use every skill it has, enough to autonomously perform an entire laundry day.

#	Task	Skills used
1	fold the clothes in the front pile	*fold_clothes
2	put clothes in washer	*close_washing_machine *open_washing_machine *start_washing_machine fold_clothes
3	put clothes in washer or dryer	close_washing_machine open_washing_machine start_washing_machine
4	fold the sheets in the upper left corner of the shelf	fold_clothes
5	fold the clothes	fold_clothes
6	load clothes in the washer	close_washing_machine open_washing_machine
7	fold all the bun cloths	fold_clothes
8	iron the shirt and pants for work tomorrow	*hang_clothes *iron_clothes *plug_in_iron *set_up_ironing_board *unplug_iron
9	start the washer with the soaps in the top of the shelves	close_washing_machine open_washing_machine start_washing_machine
10	put clothes in dryer	close_washing_machine open_washing_machine
11	fold dry clothing	fold_clothes
12	organize everything on the top shelf	fold_clothes
13	transfer clothes from washer to dryer	close_washing_machine open_washing_machine
14	add detergent and softener from bottles on shelf	*pour_detergent *pour_softener
15	hang the bibs on the side of the cart	
16	vacuum the red carpet	*operate_vacuum fold_clothes
17	load clothes into the washer like the pile in the front of the picture	close_washing_machine open_washing_machine pour_detergent start_washing_machine
18	fold clothes	fold_clothes
19	tidy up mess	close_washing_machine fold_clothes hang_clothes iron_clothes open_washing_machine operate_vacuum plug_in_iron pour_detergent pour_softener set_up_ironing_board start_washing_machine unplug_iron
20	sort clothing by color, length, and garment type	fold_clothes

continued . . .

#	Task	Skills used
21	organize laundry room	close_washing_machine fold_clothes open_washing_machine operate_vacuum pour_detergent pour_softener start_washing_machine
22	start the washer	close_washing_machine open_washing_machine pour_detergent pour_softener start_washing_machine

Table B.3: Skills learned (marked with asterisk) and used in crowd-sourced tasks proposed for laundry scenes.

References

[1] T. Brown *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[2] OpenAI, *GPT-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[3] OpenAI, *GPT-4V(ision) system card*, https://cdn.openai.com/papers/GPTV_System_Card.pdf, 2021.

[4] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, “Voyager: An open-ended embodied agent with large language models,” in *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[5] B. Ichter *et al.*, “Do as I can, not as I say: Grounding language in robotic affordances,” in *6th Annual Conference on Robot Learning*, 2022.

[6] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi, “ChatGPT empowered long-step robot control in various environments: A case application,” *IEEE Access*, pp. 1–1, 2023.

[7] S. Li *et al.*, “Pre-trained language models for interactive decision-making,” in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 31 199–31 212.

[8] W. Huang *et al.*, “Inner monologue: Embodied reasoning through planning with language models,” in *6th Annual Conference on Robot Learning*, 2022.

[9] M. Parakh, A. Fong, A. Simeonov, T. Chen, A. Gupta, and P. Agrawal, “Lifelong robot learning with human assisted language planners,” 2023. arXiv: 2309.14321 [cs.RO].

[10] M. Parakh, “Building a language conditioned system for 6-DoF tabletop manipulation,” Master’s Thesis, Massachusetts Institute of Technology, 2023.

[11] A. Fong, “NDF-based API for human-assisted language planning (HaLP),” Master’s Thesis, Massachusetts Institute of Technology, 2023.

[12] A. Zeng *et al.*, “Transporter networks: Rearranging the visual world for robotic manipulation,” in *4th Annual Conference on Robot Learning*, 2020.

[13] M. Sundermeyer, A. Mousavian, R. Triebel, and D. Fox, “Contact-GraspNet: Efficient 6-DoF grasp generation in cluttered scenes,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 13 438–13 444.

[14] A. Simeonov, Y. Du, A. Tagliasacchi, J. B. Tenenbaum, A. Rodriguez, P. Agrawal, and V. Sitzmann, "Neural descriptor fields: SE(3)-equivariant object representations for manipulation," in *2022 International Conference on Robotics and Automation (ICRA)*, 2021, pp. 6394–6400.

[15] A. Simeonov, Y. Du, Y.-C. Lin, A. R. Garcia, L. P. Kaelbling, T. Lozano-Pérez, and P. Agrawal, "SE(3)-equivariant relational rearrangement with neural descriptor fields," in *6th Annual Conference on Robot Learning*, 2022.

[16] M. Shridhar, L. Manuelli, and D. Fox, "CLIPort: What and where pathways for robotic manipulation," in *5th Annual Conference on Robot Learning*, 2021.

[17] A. Radford *et al.*, "Learning transferable visual models from natural language supervision," in *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[18] M. Shridhar, L. Manuelli, and D. Fox, "Perceiver-actor: A multi-task transformer for robotic manipulation," in *6th Annual Conference on Robot Learning*, 2022.

[19] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, *Language models as zero-shot planners: Extracting actionable knowledge for embodied agents*, 2022. arXiv: 2201.07207 [cs.LG].

[20] Z. Zhao, W. S. Lee, and D. Hsu, "Large language models as commonsense knowledge for large-scale task planning," in *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[21] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei, "VoxPoser: Composable 3D value maps for robotic manipulation with language models," in *7th Annual Conference on Robot Learning*, 2023.

[22] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi, *GPT-4V(ision) for robotics: Multimodal task planning from human demonstration*, 2023. arXiv: 2311.12015 [cs.RO].

[23] Y. Lin, A. S. Wang, G. Sutanto, A. Rai, and F. Meier, *Polymetis*, https://facebookresearch.github.io/fairo/polymetis/, 2021.

[24] E. Coumans and Y. Bai, *PyBullet, a Python module for physics simulation for games, robotics and machine learning*, https://pybullet.org, 2021.

[25] T. Chen, A. Simeonov, and P. Agrawal, *AIRobot*, https://github.com/Improbable-AI/airobot, 2019.

[26] A. X. Chang *et al.*, "ShapeNet: An Information-Rich 3D Model Repository," Stanford University — Princeton University — Toyota Technological Institute at Chicago, Tech. Rep. arXiv:1512.03012 [cs.GR], 2015.

[27] NVIDIA, *NVIDIA Isaac Sim*, https://developer.nvidia.com/isaac-sim, 2022.

[28] M. Mittal *et al.*, "Orbit: A unified simulation framework for interactive robot learning environments," *IEEE Robotics and Automation Letters*, vol. 8, no. 6, pp. 3740–3747, 2023.

[29] B. Sundaralingam *et al.*, "CuRobo: Parallelized collision-free robot motion generation," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 8112–8119.

[30] X. Zhou, R. Girdhar, A. Joulin, P. Krähenbühl, and I. Misra, "Detecting twenty-thousand classes using image-level supervision," in *European Conference on Computer Vision*, 2022.

[31] A. Kirillov *et al.*, "Segment anything," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 4015–4026.

[32] OpenAI, *Introducing ChatGPT*, https://openai.com/blog/chatgpt, 2022.

[33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019.

[34] M. Grinberg, *Flask web development: developing web applications with Python*. O'Reilly Media, Inc., 2018.

[35] Bootstrap, *Build fast, responsive sites with Bootstrap*, https://getbootstrap.com, 2024.

[36] M. Deitke, D. Schwenk, J. Salvador, L. Weihs, O. Michel, E. VanderBilt, L. Schmidt, K. Ehsani, A. Kembhavi, and A. Farhadi, "Objaverse: A universe of annotated 3D objects," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 13 142–13 153.

[37] Prolific, *Easily find vetted research participants and AI taskers at scale*, https://www.prolific.com, 2014.

[38] E. Chun, Y. Du, A. Simeonov, T. Lozano-Perez, and L. Kaelbling, "Local neural descriptor fields: Locally conditioned object representations for manipulation," in *2023 International Conference on Robotics and Automation (ICRA)*, 2023.

[39] A. Simeonov, A. Goyal, L. Manuelli, Y.-C. Lin, A. Sarmiento, A. R. Garcia, P. Agrawal, and D. Fox, "Shelving, stacking, hanging: Relational pose diffusion for multi-modal rearrangement," in *7th Annual Conference on Robot Learning*, 2023.