

PARTNR: A BENCHMARK FOR PLANNING AND REASONING IN EMBODIED MULTI-AGENT TASKS

Anonymous authors

Paper under double-blind review

ABSTRACT

We present a benchmark for Planning And Reasoning Tasks in human-Robot collaboration (**PARTNR**) designed to study human-robot coordination in household activities. **PARTNR** tasks exhibit characteristics of everyday tasks, such as spatial, temporal, and heterogeneous agent capability constraints. We employ a semi-automated task generation pipeline using Large Language Models (LLMs), incorporating simulation-in-the-loop for the grounding and verification. **PARTNR** stands as the largest benchmark of its kind, comprising 100,000 natural language tasks, spanning 60 houses and 5,819 unique objects. We analyze state-of-the-art LLMs on **PARTNR** tasks, across the axes of planning, perception and skill execution. The analysis reveals significant limitations in SoTA models, such as poor coordination and failures in task tracking and recovery from errors. When LLMs are paired with *real* humans, they require 1.5x as many steps as two humans collaborating and 1.1x more steps than a single human, underscoring the potential for improvement in these models. We further show that fine-tuning smaller LLMs with planning data can achieve performance on par with models 9 times larger, while being 8.6x faster at inference. Overall, **PARTNR** highlights significant challenges facing collaborative embodied agents and aims to drive research in this direction.

1 INTRODUCTION

Imagine a domestic robot that collaborates with humans in daily activities using natural language, akin to human-to-human interactions. This scenario requires two key features: the dynamic collaboration between the robot and the human, and the use of natural language for interaction. Current benchmarks in embodied AI typically satisfy one or the other condition; either robots operate in isolation (Shridhar et al., 2020; Zhu et al., 2023; Krantz et al., 2020; Majumdar et al., 2024), or tasks are not specified in natural language (Yenamandra et al., 2023; Puig et al., 2024; Szot et al., 2023; Jain et al., 2020). Despite significant progress in the field of embodied AI, there remains a gap in realistic benchmarks that evaluate robots in collaborative settings. To bridge this gap, we introduce Planning And Reasoning Tasks in human-Robot collaboration (**PARTNR**), a novel benchmark that evaluates the ability of embodied AI agents to collaborate with humans across a range of household activities in simulated indoor environments (Figure 1).

PARTNR consists of 100,000 natural language instructions paired with tailored evaluation functions, focusing on four task types: (1) constraint-free, where sub-tasks can be completed in any manner by either agent, (2) spatial tasks that contain spatial constraints, (3) temporal tasks that require ordered execution, and (4) heterogeneous tasks that include actions that cannot be completed by one of the agents. Beyond the conventional challenges of long-horizon planning, novel partially observed environments, and large state and action spaces, **PARTNR** emphasizes the need for effective collaboration dynamics, such as task division and tracking partner’s progress.

Curating such a benchmark of large-scale, natural language tasks with tailored evaluation functions presents significant challenges. Current benchmarks typically rely on either templated tasks (Shridhar et al., 2020; Zhang et al., 2024a) or tasks and evaluations crafted by humans (Mandi et al., 2024; Li et al., 2023a), which can restrict the diversity or the scale of the datasets. To overcome this, we propose a semi-automated generation method using Large Language Models (LLMs) with simulation-in-the-loop grounding. First, a Large Language Model (LLM) generates task and evaluation functions, which are grounded in the objects and furniture of a simulated house. Next, we employ simulation-in-

054
055
056
057
058
059
060
061
062
063
064
065
066
067
068
069
070
071
072
073
074
075
076
077
078
079
080
081
082
083
084
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104
105
106
107

PARTNR: A collaboration dataset for planning and reasoning in multi-agent tasks

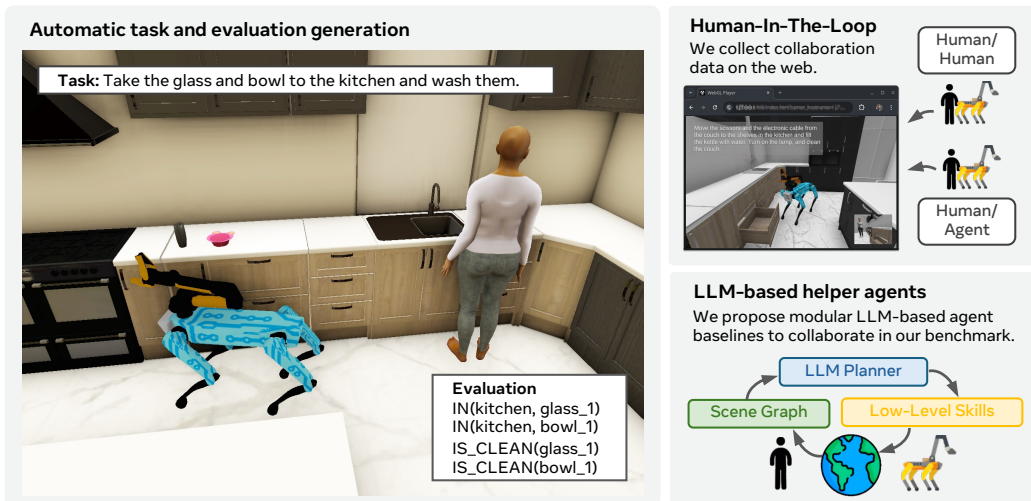


Figure 1: We present **PARTNR**, a benchmark for planning and reasoning in embodied multi-agent tasks, featuring 100,000 everyday tasks and evaluation functions generated semi-automatically, spanning 60 houses and 5,819 unique objects. We analyze LLM-based planning agents and also provide a human-in-the-loop tool to evaluate how agents collaborate with real humans.

the-loop to filter out hallucinations and infeasible instructions, complemented by human annotation to enhance diversity and accuracy. Subsequently, a set of 1,000 verified instructions and evaluation functions, along with diverse simulation houses, are utilized to guide an LLM through in-context prompting to create 100,000 tasks.

As **PARTNR** consists of natural language tasks and LLMs have shown strong results in planning (Yao et al., 2023; Ahn et al., 2022; Huang et al., 2022), we explore prompting and fine-tuning LLMs, to assess their effectiveness in collaborative scenarios. We study the effect of observability of the environment (i.e., full or partial), centralized vs. decentralized multi-agent control, learned or privileged oracle robot skills, and different ways of grounding the 3D world information for LLM-based planning. Beyond these automated evaluations with *synthetic* human partners, we also perform evaluations with *real* humans-in-the-loop, with people performing the task alone, with a human partner, or with an LLM-guided robot partner. Overall, we find that LLMs struggle at coordination, task tracking and dealing with perception and skill errors. While humans are able to solve 93% of **PARTNR** tasks, SoTA LLMs can only successfully complete 30% under non-privileged conditions. Moreover, in decentralized multi-agent settings, task completion takes 1.3x more steps than single-agent, due to poor tracking of partner actions, resulting in extraneous actions. In contrast, human pairs outperform single humans, in our human-in-the-loop experiments, highlighting potential for improving LLM collaboration strategies. LLMs also struggle to recover from skill failures and perception grounding errors, resulting in lower performance when privileged skills and privileged perception are removed. When comparing model sizes, we observe that a smaller fine-tuned Llama3.1-8B achieves a similar performance to a Llama3.1-70B without finetuning, while being 8.6x faster. This faster inference plays an important role when evaluated with real humans-in-the-loop, where the finetuned model takes fewer steps and offloads more tasks from the human.

In summary, **PARTNR** enables reproducible, large-scale, and systematic evaluations of embodied agents in a wide variety of collaborative scenarios. Through systematic evaluation, we reveal critical insights into the current limitations of LLM-based planners, opening interesting future research directions. All code, datasets, and human demonstrations on **PARTNR** tasks will be open-sourced.

2 RELATED WORK

Language-based benchmarks in Embodied AI. A large body of work on language benchmarks in Embodied AI has focused on navigation (Anderson et al., 2018; Krantz et al., 2020; Chen et al., 2019) or Embodied Question Answering (Das et al., 2018; Majumdar et al., 2024) which involve navigation

| | Environment | Multi-Agent | Language | Action Space | Task Types | Num tasks |
|-----------------------------------|-------------|-------------|----------|--------------|------------|-----------|
| Overcooked (Carroll et al., 2019) | 2D | ✓ | | HL | C | 4 |
| RoboGen (Wang et al., 2024) | 3D | | ✓ | LL+HL | CST | 106 |
| GenSim (Katara et al., 2023) | 3D | | ✓ | LL | CS | 100 |
| RoCo (Mandi et al., 2024) | 3D | ✓ | ✓ | LL | CS | 6 |
| FurnMove (Jain et al., 2020) | 3D-S | ✓ | | LL | C | 30 |
| RoboCasa (Nasiriany et al., 2024) | 3D-S | ✓ | ✓ | LL | CST | 100 |
| ALFRED (Shridhar et al., 2020) | 3D-S | | ✓ | HL | CST | 25,743 |
| BEHAVIOR-1K (Li et al., 2023a) | 3D-M | | | LL+HL | CST | 1,000 |
| WAH (Puig et al., 2021) | 3D-M | ✓ | | HL | C | 1,211 |
| Co-ELA (Zhang et al., 2024a) | 3D-M | ✓ | ✓ | HL | C | 44 |
| PARTNR | 3D-M | ✓ | ✓ | LL+HL | CSTH | 100,000 |

Table 1: **Comparison to similar embodied benchmarks.** We compare **PARTNR** to embodied AI benchmarks, focusing on natural language and multi-agent collaboration tasks. Comparison axes are – **Environment:** Household single room (S), household multi-room (M). **Action Space:** High-Level Actions (HL), Low-level Actions (LL). **Task Types:** Constraint-free (C), Spatial (S), Temporal (T), Heterogeneous (H) **Num tasks:** We measure tasks as the number of unique scene-goal pairs.

and information gathering but do not require agents to modify their environments. Closer to our work are instruction-following benchmarks (Shridhar et al., 2020; 2021; Puig et al., 2018; Wang et al., 2024; James et al., 2020; Gong et al., 2023), where agents interact with environments to complete tasks described via language, though the diversity of tasks is limited. In contrast, we leverage LLMs to generate diverse task definitions and scene initializations, and extend them to *multi-agent* settings. The idea of scaling up task generation using LLMs has been explored in a few recent works (Katara et al., 2023; Wang et al., 2024; Xian et al., 2023; Nasiriany et al., 2024). However, these works tend to focus on single-agent tasks that span relatively short horizons, while we consider long-horizon, multi-agent problems. Table 1 compares relevant benchmarks with **PARTNR**.

Embodied multi-agent benchmarks. Multiple works have proposed embodied multi-agent benchmarks (Puig et al., 2023; Agashe et al., 2023; Zhang et al., 2024a; Jain et al., 2019; Suarez et al., 2019). Many of these benchmarks focus on coordination in simple 2D environments, limiting their applicability to real world settings (Agashe et al., 2023; Carroll et al., 2019). Recent works have developed benchmarks studying collaboration in more realistic environments and activities (Puig et al., 2021; Zhang et al., 2024a; Jain et al., 2019; Puig et al., 2024; Szot et al., 2023), focusing on rearranging objects or furniture in large, partially observable 3D environments (Puig et al., 2021; 2024; Jain et al., 2019; Szot et al., 2023), or manipulating objects in a counter-top space (Mandi et al., 2024). However, these benchmarks are typically limited to a predefined and reduced set of tasks, often not described in natural language and primarily involving object rearrangement. In contrast, **PARTNR** covers an open set of tasks, each described in natural language, requiring agents to rearrange objects with spatial and temporal constraints, as well as requiring heterogeneous actions that can only be done by the human agent, (e.g., washing dishes or turning on the oven).

LLMs for decision making. Several works use LLMs as interactive policies, highlighting challenges in grounding them with observations and actions (Huang et al., 2022; Yao et al., 2023; Ahn et al., 2022; Huang et al., 2023; Zeng et al., 2022; Zheng et al., 2023; Guo et al., 2024; Liu et al., 2024). Some approaches improve grounding by prompting LLMs with demonstrations and task-specific constraints (Huang et al., 2022; Yao et al., 2023), or by integrating LLMs with external modules for multi-modal reasoning (Ahn et al., 2022; Huang et al., 2023; Zeng et al., 2022). Toolformer (Schick et al., 2023) allows LLMs to call APIs for information retrieval or environmental interaction. For instance, APIs can be used to call low-level policies (Driess et al., 2023), to leverage VLMs for obtaining the state of the world (Huang et al., 2023; Zhang et al., 2024b), or to another LLM serving as a world model (Zhao et al., 2024). SayPlan (Rana et al., 2023) maintains a persistent graph of the current world-state (Gu et al., 2024; Werby et al., 2024), enabling detailed semantic and geometric queries. Our work synthesizes these ideas by encoding the environment into a graph, using tools to extract relevant information, and executing tasks through motor skills. Another line of work fine-tunes LLMs with data from the target environments, by learning input and output adaptors (Li et al., 2022; Szot et al., 2024; Xiang et al., 2023). We explore low-rank adaptation of LLMs with multi-agent data to enhance coordination and efficiency. While fewer studies focus on LLMs in multi-agent collaboration (Zhang et al., 2024a; Park et al., 2023; Li et al., 2023b; Zhou et al., 2024), one notable example is CoELA (Zhang et al., 2024a) that design collaborative agents, though limited

162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215

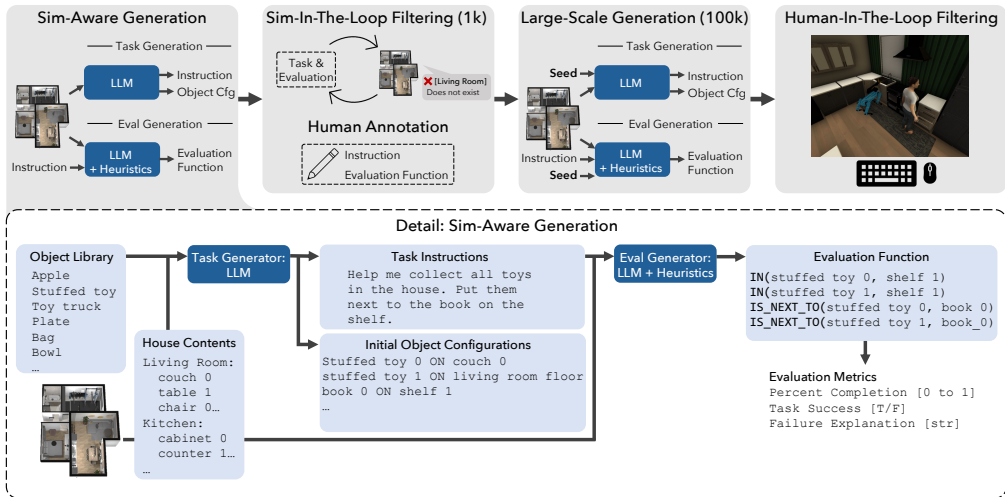


Figure 2: **The PARTNR generation pipeline.** Task and evaluation generators produce episodes, which are filtered and annotated for correctness. These episodes are then treated as seeds to achieve 100k-scale. Finally, episodes are vetted during human-in-the-loop collection.

in task diversity. Our work addresses a broader range of tasks and considers agents with varying capabilities, pushing the boundaries of multi-agent system collaboration in more complex scenarios.

3 BENCHMARK GENERATION

We introduce **PARTNR**, a benchmark aimed at training and evaluating robots at solving natural language tasks in collaboration with humans. **PARTNR** covers four types of tasks: (1) Constraint-free tasks, where sub-tasks can be completed in any manner by either agent. For example, *"Let's move all dirty plates to the sink."* (2) Spatial tasks that require reasoning about the spatial positioning of objects. For instance, *"Let's place the books on the shelf next to each other."* (3) Temporal tasks, where the sequence in which sub-tasks are executed is important. For example, *"Let's remove the candles from the dining table before bringing the plates to the table."* (4) Heterogeneous tasks, involving actions that are beyond the robot's capabilities. For example, *"Let's wash the dishes before putting them in shelves."* In scenarios where the robot's skills do not support washing, completing this task requires reasoning about agent capabilities. Our benchmark consists of natural language instructions and corresponding evaluation functions, both of which are generated at-scale using LLMs. Specifically, we generate 1,000 human-verified instructions and corresponding evaluation functions and use them as in-prompt examples to scale to 100,000 tasks in other scenes with different layouts and objects. A unique aspect of our automatic generation is the integration of an embodied simulator within the generation loop, which significantly reduces LLM errors such as hallucinations and infeasible actions.

Task Instruction

"Let's tidy up the family room. The toys go in the toy box. After that, set the plants on the shelf next to each other."

Evaluation Function

```

Propositions:
0 is_inside(["toy_fire_truck_0"], ["toy_box_0"])
1 is_inside(["toy_food_0"], ["toy_box_0"])
2 is_on_top(["plant_0"], ["shelf_0", "shelf_1"])
3 is_on_top(["plant_1"], ["shelf_0", "shelf_1"])
4 is_next_to(["plant_0"], ["plant_1"])

Dependencies:
0 WhileSatisfied([4], depends_on=[2,3])

Constraints:
0 TemporalConstraint(
  [(0,2), (1,2), (0,3), (1,3), (0,4), (1,4)]
)
1 TerminalSatisfactionConstraint([0,1,2,3,4])
    
```

Verify "is_next_to" when the plants are placed

The toys must be rearranged first

All placements are terminal

Figure 3: **Task and evaluation example.** Language tasks have inherent complexity and ambiguity; both of which are supported by the structures of our evaluation functions.

3.1 SIMULATION-IN-THE-LOOP TASK INSTRUCTION GENERATION

While LLM-based task generation has been studied in literature before (Katara et al., 2023; Wang et al., 2024; Xian et al., 2023; Nasiriany et al., 2024), these generations are not grounded beyond

216 user-created in-context prompts. In **PARTNR**, we use a simulation-in-the-loop generation technique
 217 to ground the LLM in the environment, agents and available actions. Specifically, we instantiate a
 218 simulation environment in the Habitat 3.0 simulator (Puig et al., 2024), populated with the HSSD
 219 dataset (Khanna et al., 2024), consisting of 60 unique houses and 5,819 OVMM objects (Yenamandra
 220 et al., 2023). The simulated house is parsed into a list of rooms and available furniture, and passed
 221 to an LLM, along with all available objects. Using this information, the LLM is asked to generate
 222 free-form, viable tasks in the scene, along with an initial scene state description. For example, if the
 223 generated task is "Clear dishes from the living room", the LLM should generate an initial scene with
 224 multiple dishes in the living room. At this stage, additional objects are also added to the scene to
 225 create clutter in the environment. Once generated, the tasks, initial states, and clutter are instantiated
 226 in the simulator, and infeasible instructions are filtered. For example, if the house does not have a
 227 living room, "Clear dishes from the living room" is invalid. Similarly, if the generated task requires
 228 actions not supported by the simulator, such as `folding`, the task is filtered. Generally, the rate of
 229 hallucinations is high, leading to a significant number of episodes being discarded. We observe that
 230 after filtering for infeasible instructions, the diversity in generated instructions is typically limited. For
 231 example, most of the instructions use the same objects (e.g., dishes) or similar rooms (e.g., kitchen or
 232 dining room). To increase diversity of the generated tasks, we manually annotate them to ensure task
 233 and object diversity, such as maintaining a balanced distribution of constraint-free, spatial, temporal,
 234 and heterogeneous tasks by modifying the instructions to elicit specific characteristics. This process
 results in 1,000 human annotated and simulation-verified tasks (Appendix A.5.2).

235 Such manual annotation is not practical for large-scale generation. Instead,
 236 we leverage the human-annotated 1,000 instructions to scale generation
 237 by using them as in-prompt examples. We prompt the LLM with both a house
 238 description and an example task, and instruct it to modify the task to fit
 239 the new house. For example a task like "Clear all dishes from the living
 240 room" is modified to "Clear all toys from the bedroom." This allows us to
 241 maintain the diversity of the original annotated instruction set, while ensuring
 242 a high likelihood of successful instantiation in the simulator. Qualitatively, we filtered or edited
 243 ~90% of free-form generated instructions and only ~10% of scaled instructions. We use *LLama3-70B-Instruct*
 244 (Dubey et al., 2024) for all instruction generation. Finally, all tasks go through a
 245 human-in-the-loop filtering. In this step, humans attempt to solve the tasks using our human-in-the-
 246 loop tool (Appendix A.12) and eliminate physically infeasible instructions that are difficult to detect,
 247 such as requiring an object to be at two locations. Figure 2 provides an overview of our pipeline.
 248 Details on the generation process can be found in Appendix A.3 and prompts in Appendix A.13.

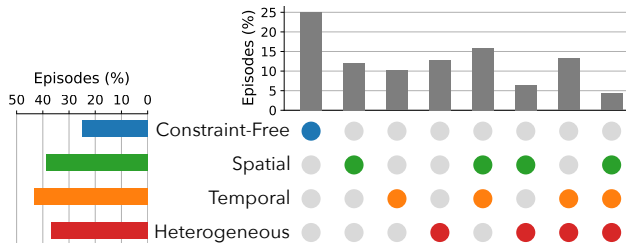


Figure 4: **Distribution of task types in PARTNR.** The left plot displays the percentage of tasks with each characteristic. Constraint-free tasks by definition exclude the other types. The top right bars correspond to the dot combination below.

3.2 EVALUATION FUNCTION GENERATION

257 To determine if an agent successfully completed the instruction "Clear all dishes from the living
 258 room", we need an evaluation function that can validate the removal of all spoons, forks, and other
 259 dishes from any of the living rooms. However, manually annotating all necessary rearrangements and
 260 state changes of a task is time intensive and since all tasks are unique, impractical at scale. Similar
 261 to instruction generation, we employ an LLM to create an evaluation function that assesses task
 262 completion without requiring any manual annotations. Specifically, we leverage the ability of LLMs to
 263 generate predicate-based Python programs using three types of APIs: a list of **propositions** indicating
 264 *what* relations between entities must be satisfied, a set of **dependencies** indicating *when* propositions
 265 should be queried, and a set of **constraints** indicating *how* propositions must be satisfied. We define an
 266 expressive vocabulary of each of these components to afford evaluation of all tasks in the benchmark
 267 (e.g., Figure 3). Closely related evaluation systems include defining tasks in PDDL (Ghallab et al.,
 268 1998) or BDDL (Srivastava et al., 2022). We choose to build a new Python-based evaluation system
 269 since neither have the expressivity to evaluate **PARTNR** tasks while maintaining human and LLM
 interpretability; for instance, BDDL does not support time-varying evaluation. Since **PARTNR** tasks
 have temporal dependencies (e.g. multi-step rearrangement), the input to the evaluation function is

the complete sequence of simulator states during task execution. The evaluation function returns 3 metrics: (1) Percent Complete ($PC \in [0, 1]$), the % of propositions that are satisfied w.r.t. constraints, (2) Success ($S \in \{\text{True}, \text{False}\}$), measuring if a task was successfully completed, defined as $S := (PC = 1)$, and (3) Failure Explanation (FE), a human and LLM interpretable language description of why the agents failed to accomplish the task. See Appendix A.4 for details.

We use *CodeLLama-70B-instruct* (Roziere et al., 2023) for evaluation function generation. Exemplified in Figure 3, producing perfect evaluation functions is non-trivial. The LLM must correctly classify the entire space of possible actions against natural language instructions and the specific simulation environment, which can be quite complex. For example, in Figure 3, the instruction "set the plants on the shelf" refers to "the shelf", but two shelves exist in the room. The evaluation function must allow either shelf while requiring placement of all plants, and finally account for a next-to relation. Any error or missing value in either a proposition or constraint invalidates the evaluation function. Consequently, we observe a large error rate in LLM generation, particularly pertaining to incorrect propositions and temporal sequencing constraints.

To alleviate these inaccuracies, we follow a similar semi-automated procedure to instruction generation. We first generate evaluation functions for the 1,000 human-annotated instructions and perform manual annotation to correct them (Appendix A.5.3). This results in a dataset of 1,000 human-verified instruction and evaluation pairs. Next, we generate evaluations for the scaled 100,000 instruction set. Recall that the scaled instructions are generated by prompting the LLM with an example instruction from the annotated set. We retrieve the corresponding annotated evaluation function and prompt the LLM with it. This is similar to approaches such as retrieval-augmented generation (Lewis et al., 2020) and improves the accuracy of evaluation function generation from 50% to 92% as measured through manual inspection (Appendix A.5.2). As a final step, we ask human users to solve all **PARTNR** tasks using our human-in-the-loop evaluation tool (Appendix A.12). All tasks that cannot be solved by humans over 6 retries (3 single-user, 3 multi-user tries) are deemed infeasible, and removed from the dataset. We find that about 90% of instructions, and 92% of evaluation functions from automated generation are accurate, resulting in a combined generation accuracy of $90 \times 92 = 83\%$ from automated generation, and 98% after human annotation and filtering.

3.3 THE **PARTNR** DATASET

The **PARTNR** dataset comprises of 100,000 episodes in 37 train scenes, 1,000 episodes in 13 validation scenes, and 1,000 episodes in 10 test scenes from the HSSD dataset (Khanna et al., 2024). After scaled generation, all validation and test set episodes are human annotated for correctness, as well as a 2,000-episode subset of train. See Appendix A.5.1 for correctness analysis of scale-generated episodes. Below, we analyze the characteristics and diversity of this dataset.

Characteristics: As described earlier, **PARTNR** focuses on four task types: constraint-free, spatial, temporal, and heterogeneous. We show the distribution of these task types in the test split in Figure 4; validation split is similar. **PARTNR** evaluates collaboration along these axes both independently and jointly. Secondary characteristics of interest include dependent rearranges (e.g., "Place them on the same table") and multi-step rearrangement of the same object (e.g. "Move the cup to the sink, wash it, then place it in the cabinet"). 7% of tasks include dependent rearranges and 6% include multi-step rearrangement. Tasks average 4.7 propositions to be satisfied (indicative of number of steps required to complete tasks). For analysis of linguistic phenomena and more characteristics, see Appendix A.2.

Diversity: The diversity of tasks in **PARTNR** is largely enabled by simulation-in-the-loop generation, which utilizes rich HSSD scenes, and the OVMM object set. Consequently, **PARTNR** tasks reference and require reasoning about 155 unique object types, 20 furniture classes and 13 room types. Note that each instruction, instantiated in each house, brings its own diversity. For example, "move the laptop to the office table", grounds *office* and *table* uniquely in each house, as well as different instances of *laptop* in different instructions. Further discussion can be found in Appendix A.2.

4 EXPERIMENTS AND ANALYSIS

We investigate how state-of-the-art planning and perception methods handle natural language tasks in new environments and coordinate with unseen partners using **PARTNR**. Since **PARTNR** consists of diverse spatio-temporal tasks specified in language, we primarily use LLMs in our baselines for planning, and study variants in (1) zero-shot prompting, retrieval-augmented generation or fine-tuning,

(2) centralized versus decentralized planning, (3) partially versus fully observed environment, (4) learned versus oracle low-level robot skills, and (5) privileged versus non-privileged perception.

Our experiments are conducted in the Habitat 3.0 simulator (Puig et al., 2024) with a simulated Spot robot (BostonDynamics; Yokoyama et al., 2023). We adopt a two-layer hierarchical control architecture, similar to (Puig et al., 2024; Szot et al., 2021), as illustrated in Figure 5, for the robot and simulated human. At the high level, a planner selects skills from a predefined skill library (e.g., navigate, pick, place, open, close). We also use a textual world graph with a three-layer hierarchy representing rooms, furniture, and movable objects. Each node in the graph stores a semantic category (e.g., kitchen, table or cup), 3D information (e.g., position or bounding box), and states (e.g., clean, powered on). See Appendix A.6 and Figure 10 for details.

4.1 BASELINES

We evaluate baselines along the following axes:

1. Variations of high-level planner:

- **Heuristic expert:** This approach utilizes expert-designed heuristics and privileged information about the task, environment and evaluation function to pre-plan all steps for human and robot based on their capabilities. For instance, both agents might rearrange objects, but only humans perform cleaning, filling, and toggling on/off tasks.
- **Zero-shot ReAct (ReAct):** We use ReAct (Yao et al., 2023) with an API library of functions or *tools* that enable the LLM to take actions. As observation, we provide the LLM with a concise, current world graph description plus a history of actions. The LLM uses this information to choose an action from [ExploreRoom, Navigate, OpenFurniture, CloseFurniture, PickObject, PlaceObject, Wait, Done] for the robot. See Appendix A.15 for prompts and Appendix A.8 for API details (human and robot).
- **ReAct with Retrieval-Augmented Generation (ReAct-RAG):** We also evaluate ReAct with RAG (Lewis et al., 2020) to investigate whether examples of planning on similar tasks improves the performance of ReAct. We construct a database of planning examples by collecting the successful traces from ReAct from a the 2,000 task training subset (see 3.3). During test time, the most relevant planning trace from the train dataset is selected based on sentence similarity and added to the LLM’s prompt (Pang et al., 2024; Madaan et al., 2024).
- **Finetuned LLMs (Finetuned):** We also investigate finetuning a smaller LLM (Llama3.1-8B) as our high-level planner, using successful traces from the ReAct baselines (Hsieh et al., 2023) that use Llama3.1-70B. Using the ReAct-RAG dataset, we split every episode into a sequence of high-level planning actions, filtering for only actions that were executed successfully. For every action, we build an input containing the world-graph and history of actions, similar to ReAct (see Appendix A.9 for more details). We then finetune an LLM to predict the action from the ReAct episode given this input, using a low-rank adapter (Hu et al., 2021). This model has reduced latency and computational demands, suitable for real world deployment.

All model generations are constrained to only output valid actions on observed objects using constrained generation (Geng et al., 2023). The constrained generation greatly reduces the hallucinations and ‘grammatical’ errors typical of LLMs. An episode is finished when both agents call Done or reach maximum simulation steps or LLM calls. Refer to Appendix A.8 for details.

2. **Centralized versus decentralized planning:** To study the overhead of coordination in multi-agent PARTNR tasks, we compare centralized and decentralized planners. In centralized, a single LLM decides actions for both agents, with complete information about both agent’s states, effectively removing any need for coordination between the agents. In decentralized, each agent is controlled by a different LLM, and each LLM needs to reason about the other agent’s actions.
3. **Partial versus full observability:** To evaluate if SoTA language models can explore new environments and identify task-relevant objects, we consider a partially observed setting where the planner knows the house’s layout but not the object locations, requiring exploration. This is in contrast to a fully observed setting, where all object locations are known in advance.
4. **Learned versus oracle low-level robot skills:** We examine the impact of learned neural-network skills versus oracle skills (with privileged simulation information) on overall performance in PARTNR tasks. We create a library of learned skills for pick, place, navigate, open and close actions (Appendix A.7 provides more details) and compare performance with oracle skills.
5. **Privileged versus non-privileged perception:** To study perception challenges such as inaccurate detection and approximate localization, we used a non-privileged world graph with modified

ConceptGraphs (Gu et al., 2024), built from agents’ RGBD observations only. As agents explore and take actions, this world graph is updated using onboard sensing (details in Appendix A.6). In contrast, with privileged perception, this information is available from the simulation.

4.2 RESULTS AND ANALYSIS

Metrics. We evaluate performance across different settings using four key metrics. First, the *simulation steps* metric measures the number of steps required for agents to complete the task within the simulation environment, serving as an indicator of efficiency. Second, the *success rate* reflects the completion of the task i.e. whether ‘all’ task constraints are satisfied. Given the complexity and long-horizon nature of **PARTNR** tasks, agents often partially complete the task. To account for this, we also report *percent complete*, which quantifies the ratio of completed task ‘propositions’ (percent complete = 1 for successful tasks). Lastly, we assess the reasoning efficiency of the planners through the *planning cycles* metric, which counts the number of high-level LLM calls each planner makes in the course of an episode. We cap the maximum planner calls at 50 in all experiments.

4.2.1 TASK PERFORMANCE ANALYSIS

Table 2 presents a comprehensive evaluation of the planning approaches defined in Section 4.1 using the *Llama3.1-70B-Instruct* model (Dubey et al., 2024) for ReAct baselines, and a finetuned *Llama3.1-8B* base model for the Finetuned baseline. Since **PARTNR** tasks are multi-agent, we also need a simulated human partner, which we control with a ReAct approach, using *Llama3.1-70B-Instruct*. Our main findings are detailed below.

PARTNR tasks are challenging for LLM-based planners. LLM-based baselines across all observability and controllability conditions perform worse than the privileged heuristic expert, due to errors in task tracking (not completing all steps, performing them in the wrong order, or undoing completed steps), and semantic errors (placing objects on the wrong furniture, or moving the wrong object), indicating a gap in LLM task planning.

LLMs struggle with coordination in decentralized settings. Decentralized ReAct baselines which do not have privileged access to partner’s intent are significantly slower at task completion than centralized ReAct (3295 steps with decentralized-partial in row(e) versus 2298 with centralized-partial in row(d)). This shows that reasoning about the partner e.g., knowing or inferring partner’s intent can improve task efficiency in **PARTNR** tasks, and current SoTA LLMs perform poorly at this. Moreover, decentralized ReAct with two agents is even *slower* than ReAct with a single-agent (3295 steps with multi-agent in row(e) versus 2519 with single-agent in row(a)), indicating that LLMs suffer from a significant coordination “burden”. This co-ordination burden is further highlighted in our analysis on *extraneous effort* in Section 4.2.2, where we find that agents end up repeating parts of the task or performing irrelevant actions with much higher frequency in decentralized settings.

LLMs are unable to recover from learned skill failures. When replacing oracle skills with learned skills, the success rate decreases from 0.73 to 0.57 (row(e) vs. row (h)). This decline can be attributed to the higher failure rate and increased number of simulation steps required by learned skills compared to privileged oracle skills. The LLMs struggle to recover from skill errors like failing to pick up an object or performing incomplete exploration, resulting in a lower success rate. Future research could investigate training large models with low-level skills in the loop, enabling them to learn recovery and replanning strategies in the face of such failures.

LLMs exhibit a high degree of sensitivity to errors in perception. When we replace privileged perception with off-the-shelf perception modules, success rate significantly declines (from 0.57 with a privileged, partial world graph in row(h) to 0.30 with Concept-Graphs (Gu et al., 2024) in row(i)). LLMs rely heavily on accurate world descriptions provided by the world graph and struggle to correct

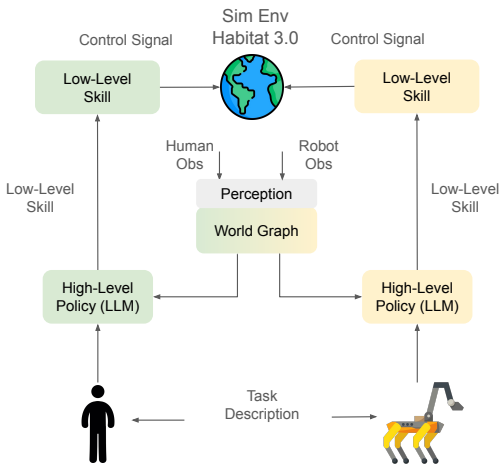


Figure 5: **Decentralized architecture.** The human and robot agents use a 2-layer hierarchical architecture, with high-level LLM planners that call low-level skills. Both agents build a world graph, updated using observations and actions.

| Method | Controllability | Skills | Observability | Sim Steps ↓ | Success Rate ↑ | Percent Complete ↑ | Planning Cycles ↓ |
|----------------------|-----------------|---------|---------------|-------------------|----------------|--------------------|-------------------|
| (a) ReAct-Single | Single Agent | Oracle | Partial | 2519.02 ± 57.48 | 0.73 ± 0.01 | 0.85 ± 0.01 | 18.68 ± 0.33 |
| (b) Heuristic-Expert | Centralized | Oracle | Full | 1260.88 ± 26.97 | 0.84 ± 0.01 | 0.94 ± 0.01 | N/A |
| (c) ReAct | Centralized | Oracle | Full | 1347.43 ± 33.80 | 0.74 ± 0.01 | 0.88 ± 0.01 | 17.49 ± 0.34 |
| (d) ReAct | Centralized | Oracle | Partial | 2298.13 ± 61.39 | 0.74 ± 0.01 | 0.85 ± 0.01 | 20.73 ± 0.51 |
| (e) ReAct | Decentralized | Oracle | Partial | 3295.20 ± 76.27 | 0.73 ± 0.01 | 0.86 ± 0.01 | 15.24 ± 0.31 |
| (f) ReAct + RAG | Decentralized | Oracle | Partial | 3467.47 ± 82.39 | 0.71 ± 0.01 | 0.84 ± 0.01 | 14.75 ± 0.31 |
| (g) Finetuned | Decentralized | Oracle | Partial | 3228.96 ± 75.14 | 0.70 ± 0.01 | 0.84 ± 0.01 | 12.85 ± 0.24 |
| (h) ReAct | Decentralized | Learned | Partial | 6494.88 ± 181.52 | 0.57 ± 0.02 | 0.76 ± 0.01 | 22.72 ± 0.58 |
| (i) ReAct | Decentralized | Learned | ConceptGraph | 12490.80 ± 208.90 | 0.30 ± 0.01 | 0.56 ± 0.01 | 23.84 ± 0.45 |

Table 2: **Analysis of planner baselines in various settings.** We compare performance using simulation steps, success rate and percent complete on the tasks, and the average number of planning cycles used by the baselines (described in Section 3).

errors such as misclassification (e.g., shelves misidentified as tables) or incorrect room assignments (e.g., a table in the living room mislabeled as being in the bedroom). Multi-modal models like VLMs might be stronger at recovering from such failures, which we leave for future work.

Finetuned 8B model performs on par with a ReAct with a 70B model, while being 8.6x faster.

We find that the finetuned planner with a small 8B model performs on par with ReAct, which uses a much larger 70B model (a 0.73 success rate with the 70B model in row(e), versus 0.70 with the finetuned 8B model in row(g)). At the same time, we find that the finetuned model is 8.6 times faster at inference. This indicates that the finetuning effectively distills task-relevant information from the training set and generalizes to new test tasks. When deployed with humans-in-the-loop, the finetuned model takes fewer steps and offloads more sub-tasks than the 70B model (see Table 3).

4.2.2 ANALYSIS OF COLLABORATIVE BEHAVIOR AND EFFICIENCY

Our analysis in Table 2 revealed challenges in LLM collaboration, prompting a deeper investigation into specific collaborative behaviors, explained below and detailed in Appendix A.11 and Table 12.

Robots offload up to 60% of tasks. We evaluate the robot’s ability to offload tasks from the human, measuring the ratio of sub-tasks performed by the robot to the total sub-tasks in successful **PARTNR** tasks. Despite similar success rates between single- and multi-agent (0.73 vs. 0.74), the robot offloads about 60% of sub-tasks in decentralized multi-agent, reducing human effort (Table 12).

Decentralized agents are prone to performing extraneous tasks. The agents sometimes end up performing sub-tasks that are not useful for the task such as rearranging an object that is not required by the task or repeating a sub-task already performed by the other agent. To capture such extraneous agent effort, we measure the portion of agent actions that did not increase the percent complete metric i.e., did not contribute to task progress, over the total number of successful agent actions in an episode. We find a 300% increase in extraneous effort in decentralized multi-agent settings compared to single-agent (Table 12), indicating a significant coordination burden.

Temporal and heterogeneous tasks are challenging for LLMs. LLMs struggle in temporal and heterogeneous tasks. Task success drops by 27% for temporal tasks and 20% for heterogeneous tasks compared to constraint-free tasks for ReAct (Table 13). This highlights the limitations of LLMs in reasoning about agent capabilities and following strict ordering constraints.

4.3 HUMAN-IN-THE-LOOP EVALUATION

We build on the human-in-the-loop infrastructure from Habitat 3.0 (Puig et al., 2024) and adapt it to a server-client architecture, with the server hosted on AWS capable of supporting multiple clients (see Appendix A.12). This allows us to run at-scale evaluation of our tasks with 129 non-expert human participants. We collect single-user and multi-user data on 1000 tasks from the validation and test set using this tool. In the single-user setting, a single participant completes the whole task, by driving the human agent in the simulator via keyboard/mouse controls Figure 14 in appendix shows our HITL interface. In multi-user, two participants complete the task together by controlling a human and a robot agent. The goal of these experiments is to study multi-user dynamics at **PARTNR** tasks, and see if multiple humans collaborating are more efficient than single human. Finally, we run a human-AI experiment where a human participant collaborates with a robot controlled by an

| Method | Success Rate \uparrow | Percent Complete \uparrow | Sim Steps \downarrow | Task Offloading \uparrow | Exploration Efficiency \downarrow | Extraneous Effort \downarrow |
|-----------------|-------------------------|-----------------------------|------------------------|----------------------------|-------------------------------------|--------------------------------|
| Single-user | 0.93 \pm 0.01 | 0.96 \pm 0.00 | 3046.99 \pm 80.79 | N/A | 2459.22 \pm 26.75 | 0.09 \pm 0.01 |
| Multi-user | 0.93 \pm 0.01 | 0.96 \pm 0.00 | 2369.55 \pm 49.33 | 0.59 \pm 0.01 | 1762.47 \pm 13.99 | 0.15 \pm 0.01 |
| Human-ReAct | 0.91 \pm 0.01 | 0.96 \pm 0.02 | 4267.71 \pm 83.40 | 0.16 \pm 0.01 | 2624.39 \pm 26.05 | 0.12 \pm 0.01 |
| Human-Finetuned | 0.92 \pm 0.01 | 0.96 \pm 0.00 | 3443.33 \pm 61.46 | 0.26 \pm 0.01 | 2164.94 \pm 21.31 | 0.13 \pm 0.01 |

Table 3: **Human-in-the-Loop Evaluation.** We evaluate the performance of a 2-person human team and human-LLM teams, comparing them to solo human performance on **PARTNR** tasks using metrics described in Section 4.1. Additional results and analysis in Appendix A.12.

LLM (using the ReAct and Finetuned models from Section 4.1). This experiment aims to evaluate LLM-controlled agents at collaborating with unseen, real humans. Table 3 shows the success rate (SR) and percent complete (PC) of tasks from the validation set in a single-user, multi-user, human-ReAct and human-Finetuned setting. Additionally, we measure the number of steps taken by each approach to complete the task, and the ratio of work completed by the robot i.e., task offloading. We also measure exploration efficiency in human-in-the-loop, by measuring the steps taken to pick the first object, and extraneous effort, indicating actions that were not useful for task completion. These results are summarized in Table 3. Some key findings are below (more results and analysis in A.12.4):

Humans are significantly better than LLMs at PARTNR tasks. In both single and multiple human environments, the success rate achieved is 0.93 on the **PARTNR** benchmark. In contrast, the ReAct model without any privileged information, achieves a significantly lower success rate of 0.30 (row (i) of Table 2). This highlights a significant gap in the performance of LLMs in planning tasks. Note that the LLM baselines like ReAct and Finetuned achieve a success rate of 0.92 and 0.91 when evaluated with *real* humans (Table 3), because humans are able to adapt to LLM mistakes. On the other hand, the simulated human in Table 2 is an LLM, which is unable to recover from partner mistakes.

Finetuned LLMs perform better than ReAct when coordinating with real humans. When deployed with real humans-in-the-loop, the finetuned model is faster than ReAct at task completion (3443 steps with finetuned versus 4267 with ReAct). It is also able to offload more tasks from humans than ReAct (26% with finetuned as compared to 16% with ReAct). This reflects that smaller models with faster inference can improve human experience in real-world deployment.

LLMs struggle at coordination, hampering human performance. Despite the Finetuned being faster than ReAct when collaborating with humans, both approaches are *slower* than the human doing the task alone. In contrast, two humans working together complete the task faster than a single human (2369 steps vs. 3046 with multi- and single-user respectively). This result is in line with the automated evaluation we observed in Table 1, where multi-agent LLMs are also *slower* than a single-agent LLM. This result further reinforces that LLMs suffer at coordination; while humans are able to coordinate and divide tasks between each other, decentralized LLMs are unable to do so.

LLMs are able to offload tasks from humans. Despite the aforementioned increase in the number of steps for task completion, robots guided by the finetuned model successfully offload 26% of tasks from humans. This indicates that LLMs can still offer assistance when collaborating with real human partners. Nonetheless, there remains significant potential for improvement.

5 CONCLUSION

We present **PARTNR**, a benchmark for reasoning and planning in multi-agent embodied tasks, featuring 100,000 natural language tasks instantiated in 60 simulated, multi-room houses with 5,819 unique objects. We use a semi-automated LLM-powered pipeline for large-scale instruction and evaluation function generation that uses simulation-in-the-loop grounding. **PARTNR** exhibits characteristics of everyday tasks, such as temporal and spatial constraints, and allows systematic evaluation of planning approaches. We find a significant gap between SoTA LLMs and human-level performance at **PARTNR** tasks. While our best LLM baseline only succeeds at 30% of tasks with no privileged information, humans are able to solve 93% of the tasks. Moreover, LLMs face challenges in coordinating with both LLM-based agents and real human partners. Human-in-the-loop evaluations, involving real humans collaborating with LLM-guided robots, reveal that LLM-guided partners decrease human efficiency compared to working solo. This suggests that LLM-based agents require significant improvements to become effective collaborative partners in embodied tasks. **PARTNR** serves as a challenging benchmark that highlights the substantial limitations of current models.

REFERENCES

- 540
541
542 Saaket Agashe, Yue Fan, Anthony Reyna, and Xin Eric Wang. Evaluating multi-agent coordination
543 abilities in large language models. *arXiv*, 2023. 3
- 544 Christopher Agia, Krishna Murthy Jatavallabhula, Mohamed Khodeir, Ondrej Miksik, Vibhav Vineet,
545 Mustafa Mukadam, Liam Paull, and Florian Shkurti. Taskography: Evaluating robot task planning
546 over large 3d scene graphs. In *CoRL*, 2022. 29
- 547 Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea
548 Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine
549 Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally
550 Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey
551 Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek
552 Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev,
553 Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do
554 as i can and not as i say: Grounding language in robotic affordances. In *CoRL*, 2022. 2, 3
- 555 Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid,
556 Stephen Gould, and Anton Van Den Hengel. Vision-and-language navigation: Interpreting visually-
557 grounded navigation instructions in real environments. In *CVPR*, 2018. 2
- 558 BostonDynamics. Spot robot. <https://www.bostondynamics.com/products/spot>.
559 Accessed: 2024-10-01. 7
- 560
561 Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca
562 Dragan. On the utility of learning about humans for human-ai coordination. In *NeurIPS*, 2019. 3
- 563
564 Howard Chen, Alane Suhr, Dipendra Misra, Noah Snavely, and Yoav Artzi. Touchdown: Natural
565 language navigation and spatial reasoning in visual street environments. In *CVPR*, 2019. 2, 17
- 566 Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. Embodied
567 question answering. In *CVPR*, 2018. 2
- 568
569 Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan
570 Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal
571 language model. In *ICML*, 2023. 3
- 572 Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha
573 Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models.
574 *arXiv*, 2024. 5, 8
- 575 Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding
576 for structured NLP tasks without finetuning. In *EMNLP*, 2023. 7, 33
- 577
578 Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman,
579 Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. Pddl - the
580 planning domain definition language. *Technical Report*, 1998. 5
- 581 Ran Gong, Jiangyong Huang, Yizhou Zhao, Haoran Geng, Xiaofeng Gao, Qingyang Wu, Wensi Ai,
582 Ziheng Zhou, Demetri Terzopoulos, Song-Chun Zhu, et al. Arnold: A benchmark for language-
583 grounded task learning with continuous states in realistic 3d scenes. In *ICCV*, 2023. 3
- 584 Qiao Gu, Alihusein Kuwajerwala, Sacha Morin, Krishna Murthy Jatavallabhula, Bipasha Sen, Aditya
585 Agarwal, Corban Rivera, William Paul, Kirsty Ellis, Rama Chellappa, Chuang Gan, Celso Miguel
586 de Melo, Joshua B. Tenenbaum, Antonio Torralba, Florian Shkurti, and Liam Paull. Conceptgraphs:
587 Open-vocabulary 3d scene graphs for perception and planning. In *ICRA*, 2024. 3, 8, 29
- 588
589 Yanjiang Guo, Yen-Jen Wang, Lihan Zha, and Jianyu Chen. Doremi: Grounding language model
590 by detecting and recovering from plan-execution misalignment, 2024. URL <https://arxiv.org/abs/2307.00329>. 3
- 591
592 Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner,
593 Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger
language models with less training data and smaller model sizes. *arXiv*, 2023. 7

- 594 Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu
595 Chen. Lora: Low-rank adaptation of large language models. In *ICLR*, 2021. 7, 35
596
- 597 Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot
598 planners: Extracting actionable knowledge for embodied agents. In *ICML*, 2022. 2, 3
599
- 600 Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan
601 Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda
602 Luu, Sergey Levine, Karol Hausman, and brian ichter. Inner monologue: Embodied reasoning
603 through planning with language models. In *CoRL*, 2023. 3
- 604 Unnat Jain, Luca Weihs, Eric Kolve, Mohammad Rastegari, Svetlana Lazebnik, Ali Farhadi, Alexander
605 G. Schwing, and Aniruddha Kembhavi. Two body problem: Collaborative visual task comple-
606 tion. In *CVPR*, 2019. first two authors contributed equally. 3
607
- 608 Unnat Jain, Luca Weihs, Eric Kolve, Ali Farhadi, Svetlana Lazebnik, Aniruddha Kembhavi, and
609 Alexander G. Schwing. A cordial sync: Going beyond marginal policies for multi-agent embodied
610 tasks. In *ECCV*, 2020. 1, 3
- 611 Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. Rlbench: The robot
612 learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 2020. 3
613
- 614 Pushkal Katara, Zhou Xian, and Katerina Fragkiadaki. Gen2sim: Scaling up robot learning in
615 simulation with generative models. *arXiv*, 2023. 3, 4
- 616 Mukul Khanna, Yongsen Mao, Hanxiao Jiang, Sanjay Haresh, Brennan Shacklett, Dhruv Batra,
617 Alexander Clegg, Eric Undersander, Angel X. Chang, and Manolis Savva. Habitat synthetic scenes
618 dataset (hssd-200): An analysis of 3d scene scale and realism tradeoffs for objectgoal navigation.
619 In *CVPR*, 2024. 5, 6, 16, 20
620
- 621 Jacob Krantz, Erik Wijmans, Arjun Majumdar, Dhruv Batra, and Stefan Lee. Beyond the nav-graph:
622 Vision-and-language navigation in continuous environments. In *ECCV*, 2020. 1, 2
623
- 624 Alexander Ku, Peter Anderson, Roma Patel, Eugene Ie, and Jason Baldridge. Room-across-room:
625 Multilingual vision-and-language navigation with dense spatiotemporal grounding. In *EMNLP*,
626 2020. 17
- 627 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal,
628 Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela.
629 Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*, 2020. 6, 7
630
- 631 Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-
632 Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A
633 benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *CoRL*, 2023a.
634 1, 3
- 635 Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem.
636 Camel: Communicative agents for "mind" exploration of large language model society. In *NeurIPS*,
637 2023b. 3
638
- 639 Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An
640 Huang, Ekin Akyürek, Anima Anandkumar, et al. Pre-trained language models for interactive
641 decision-making. In *NeurIPS*, 2022. 3
- 642 Xinzhu Liu, Di Guo, Xinyu Zhang, and Huaping Liu. Heterogeneous embodied multi-agent collabo-
643 ration. *IEEE Robotics and Automation Letters*, 9(6):5377–5384, 2024. doi: 10.1109/LRA.2024.
644 3390588. 3
645
- 646 Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri
647 Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement
with self-feedback. In *NeurIPS*, 2024. 7, 34

- 648 Arjun Majumdar, Anurag Ajay, Xiaohan Zhang, Pranav Putta, Sriram Yenamandra, Mikael Henaff,
649 Sneha Silwal, Paul Mcvay, Oleksandr Maksymets, Sergio Arnaud, et al. Openeqa: Embodied
650 question answering in the era of foundation models. In *CVPR*, 2024. 1, 2
651
- 652 Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large
653 language models. In *ICRA*, 2024. 1, 3
654
- 655 Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi,
656 Ajay Mandlekar, and Yuke Zhu. Robocasa: Large-scale simulation of everyday tasks for generalist
657 robots. In *RSS*, 2024. 3, 4
658
- 659 Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason
660 Weston. Iterative reasoning preference optimization. *arXiv*, 2024. 7, 34
661
- 662 Joon Sung Park, Joseph O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S
663 Bernstein. Generative agents: Interactive simulacra of human behavior. In *ACM Symposium on
664 User Interface Software and Technology*, 2023. 3
665
- 666 Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba.
667 Virtualhome: Simulating household activities via programs. In *CVPR*, 2018. 3
668
- 669 Xavier Puig, Tianmin Shu, Shuang Li, Zilin Wang, Yuan-Hong Liao, Joshua B. Tenenbaum, Sanja
670 Fidler, and Antonio Torralba. Watch-and-help: A challenge for social perception and human-`{ai}`
671 collaboration. In *ICLR*, 2021. 3
672
- 673 Xavier Puig, Tianmin Shu, Joshua B Tenenbaum, and Antonio Torralba. Nopa: Neurally-guided
674 online probabilistic assistance for building socially intelligent home assistants. In *ICRA*, 2023. 3
675
- 676 Xavier Puig, Eric Undersander, Andrew Szot, Mikael Dallaire Cote, Tsung-Yen Yang, Ruslan
677 Partsey, Ruta Desai, Alexander William Clegg, Michal Hlavac, So Yeon Min, Vladimír Vondruš,
678 Theophile Gervet, Vincent-Pierre Berges, John M. Turner, Oleksandr Maksymets, Zsolt Kira,
679 Mrinal Kalakrishnan, Jitendra Malik, Devendra Singh Chaplot, Unnat Jain, Dhruv Batra, Akshara
680 Rai, and Roozbeh Mottaghi. Habitat 3.0: A co-habitat for humans, avatars and robots. In *ICLR*,
681 2024. 1, 3, 5, 7, 9, 16, 20, 31, 38
682
- 683 Team PyTorch. Accelerating generative ai with pytorch ii: Gpt, fast. [https://pytorch.org/
684 blog/accelerating-generative-ai-2/](https://pytorch.org/blog/accelerating-generative-ai-2/), 2023. 32
685
- 686 Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf.
687 Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning.
688 In *CoRL*, 2023. 3, 29
689
- 690 Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks.
691 In *EMNLP*, 2019. 34
692
- 693 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi
694 Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code.
695 *arXiv*, 2023. 6, 24
696
- 697 Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke
698 Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach
699 themselves to use tools. In *NeurIPS*, 2023. 3
700
- 701 Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi,
702 Luke Zettlemoyer, and Dieter Fox. ALFRED: A Benchmark for Interpreting Grounded Instructions
703 for Everyday Tasks. In *CVPR*, 2020. 1, 3
704
- 705 Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew
706 Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In
707 *ICLR*, 2021. 3
708
- 709 Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott
710 Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, Silvio Savarese, Hyowon Gweon,
711 Jiajun Wu, and Li Fei-Fei. Behavior: Benchmark for everyday household activities in virtual,
712 interactive, and ecological environments. In *CoRL*, 2022. 5

- 702 Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent
703 game environment for training and evaluating intelligent agents. *arXiv*, 2019. 3
704
- 705 Andrew Szot, Alexander Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah
706 Maestre, Mustafa Mukadam, Devendra Singh Chaplot, Oleksandr Maksymets, et al. Habitat 2.0:
707 Training home assistants to rearrange their habitat. In *NeurIPS*, 2021. 7
708
- 709 Andrew Szot, Unnat Jain, Dhruv Batra, Zsolt Kira, Ruta Desai, and Akshara Rai. Adaptive coordina-
710 tion in social embodied rearrangement. In *ICML*, 2023. 1, 3
711
- 712 Andrew Szot, Max Schwarzer, Harsh Agrawal, Bogdan Mazouze, Walter Talbott, Katherine Metcalf,
713 Natalie Mackraz, Devon Hjelm, and Alexander Toshev. Large language models as generalizable
714 policies for embodied tasks. In *ICLR*, 2024. 3
715
- 716 Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory
717 Erickson, David Held, and Chuang Gan. Robogen: Towards unleashing infinite data for automated
718 robot learning via generative simulation. In *ICML*, 2024. 3, 4
719
- 720 Abdelrhman Werby, Chenguang Huang, Martin B uchner, Abhinav Valada, and Wolfram Burgard.
721 Hierarchical open-vocabulary 3d scene graphs for language-grounded robot navigation. In *RSS*,
722 2024. 3
723
- 724 Zhou Xian, Theophile Gervet, Zhenjia Xu, Yi-Ling Qiao, and Tsun-Hsuan Wang. Towards a
725 foundation model for generalist robots: Diverse skill learning at scale via automated task and scene
726 generation. *arXiv*, 2023. 3, 4
727
- 728 Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu.
729 Language models meet world models: Embodied experiences enhance language models. In
730 *NeurIPS*, 2023. 3
731
- 732 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.
733 React: Synergizing reasoning and acting in language models. In *ICLR*, 2023. 2, 3, 7
734
- 735 Sriram Yenamandra, Arun Ramachandran, Karmesh Yadav, Austin Wang, Mukul Khanna, Theophile
736 Gervet, Tsung-Yen Yang, Vidhi Jain, Alexander William Clegg, John Turner, et al. Homerobot:
737 Open-vocabulary mobile manipulation. In *CoRL*, 2023. 1, 5
738
- 739 Naoki Yokoyama, Alex Clegg, Joanne Truong, Eric Undersander, Tsung-Yen Yang, Sergio Arnaud,
740 Sehoon Ha, Dhruv Batra, and Akshara Rai. Asc: Adaptive skill coordination for robotic mobile
741 manipulation. *IEEE Robotics and Automation Letters*, 2023. 7
742
- 743 Andy Zeng, Maria Attarian, Krzysztof Marcin Choromanski, Adrian Wong, Stefan Welker, Federico
744 Tombari, Aveek Purohit, Michael S Ryoo, Vikas Sindhwani, Johnny Lee, et al. Socratic models:
745 Composing zero-shot multimodal reasoning with language. In *ICLR*, 2022. 3
746
- 747 Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B Tenenbaum, Tianmin
748 Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language
749 models. *ICLR*, 2024a. 1, 3
750
- 751 Hongxin Zhang, Zeyuan Wang, Qiushi Lyu, Zheyuan Zhang, Sunli Chen, Tianmin Shu, Yilun Du,
752 and Chuang Gan. Combo: Compositional world models for embodied multi-agent cooperation.
753 *arXiv*, 2024b. 3
754
- 755 Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for
756 large-scale task planning. In *NeurIPS*, 2024. 3
757
- 758 Sipeng Zheng, Jiazheng Liu, Yicheng Feng, and Zongqing Lu. Steve-eye: Equipping llm-based
759 embodied agents with visual perception in open worlds, 2023. URL <https://arxiv.org/abs/2310.13255>. 3
760
- 761 Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe
762 Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, et al. Sotopia: Interactive evaluation for
763 social intelligence in language agents. In *ICLR*, 2024. 3

756 Hao Zhu, Raghav Kapoor, So Yeon Min, Winson Han, Jiatai Li, Kaiwen Geng, Graham Neubig,
757 Yonatan Bisk, Aniruddha Kembhavi, and Luca Weihs. Excalibur: Encouraging and evaluating
758 embodied exploration. In *CVPR*, 2023. 1
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809

A APPENDIX

We present **PARTNR**, a benchmark for reasoning and planning in multi-agent embodied tasks, featuring 100,000 natural language, everyday tasks. We show at-scale generation of these tasks using LLMs with simulation in the loop for grounding and human in the loop for filtering. We also evaluate several LLM-based planning models on these tasks and highlight avenues for future work. This appendix provides additional details on these contributions and is organized as follows:

- A.1 Code and **PARTNR** benchmark open-sourcing
- A.2 Details and additional analysis on the **PARTNR** dataset
- A.3 Simulation features and prompts for the **PARTNR** task generation
- A.4 The **PARTNR** evaluation system
- A.5 Human annotation and quality assessment for the **PARTNR** tasks and evaluation functions
- A.6 World graph for perception in LLM agents
- A.7 Learned low-level robot skills
- A.8 Implementation details for ReAct agents
- A.9 Details on finetuning LLM agents for the **PARTNR** tasks
- A.10 Additional results
- A.11 Analysis of collaborative behavior and efficiency of LLM agents
- A.12 Human-in-the-loop (HITL) system and evaluation for the **PARTNR** tasks and LLM agents
- A.13 Prompts for task and evaluation generation
- A.15 Prompts for planner baselines
- A.16 [Analysis of planner failures](#)

A.1 OPEN-SOURCING **PARTNR** DATASET AND CODEBASE

Accompanying this paper, we will release the code and data necessary to reproduce our experiments. Released code includes our **PARTNR** benchmark tasks, metrics, baseline oracle skills, large planning model framework, and dataset generation utilities. Released data includes extensions of the Habitat Synthetic Scenes Dataset (HSSD) ([Khanna et al., 2024](#)), generated benchmark task episodes, and model weights for our trained neural network skills and fine-tuned large planning model.

The publicly released codebase accompanying **PARTNR** will be contained in a public github repository and depend on the most recent version of the AI Habitat platform (habitat-lab and habitat-sim (v0.3.2)) ([Puig et al., 2024](#)) which it extends to define collaboration tasks and skills.

For the purpose of this submission, the anonymized code is included in the supplementary zip file. To preserve anonymity and respect supplement size limits, the data and assets required to reproduce the results are not included in the initial upload, but will be released along with the public code repository at a later date.

In order to model the space of rich indoor collaboration tasks we propose with **PARTNR**, we extended HSSD with additional asset authoring and annotation. To enable more realistic indoor object manipulation, we added articulated 3D furniture models such as drawers, cabinets, and appliances. These models were converted from rigid source assets in HSSD and swapped into the scenes. We prepared 60 scenes divided into train, val, and test splits to support our experiments. Each scene is manually adjusted by a human to ensure simulation robustness and minimize potential issues. Furniture is annotated with a set of Receptacles (surfaces which support small object placement such as shelves and drawers) and can be opened and closed by the agents. Receptacles are further filtered contextually in each scene to ensure that the active set is accessible to the agents. Additional annotations include point or marker sets for each furniture, region annotations, and semantic classification of objects. The marker sets indicate either a spread of surface points (for distance/occlusion checking) or the location of key points of interest such as faucets (for cleaning/filling) and handles (for opening/closing) necessary for low-level skill training and oracle skill execution. Region annotations included per-scene region volumes (e.g., kitchen, living-room, bedroom, etc.) for checking and specifying the location of objects and furniture. Semantic annotations indicate the object category or class (e.g. table, chair, cup, toy) to support open language prompt grounding and semantically guided task generation.

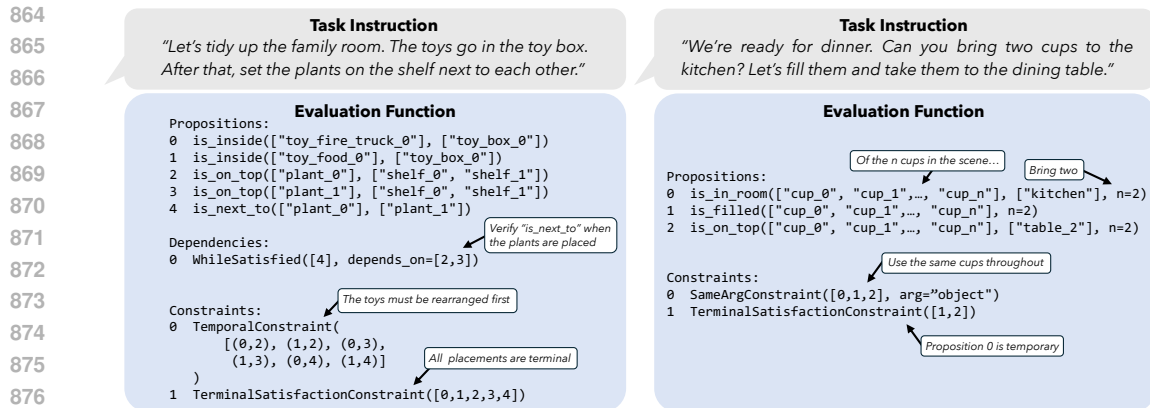


Figure 6: **Additional task and evaluation examples.** Expands on Figure 3 to show a task and evaluation containing subset counts and dependent rearranges (right).

A.2 DATASET DETAILS AND ADDITIONAL ANALYSIS

We expand on the details and analysis of the **PARTNR** dataset of Section 3.3, including analysis of linguistic phenomena, secondary task characteristics, and the distribution of sampled entities in generated tasks. See Figure 6 for additional task and evaluation examples.

Linguistic Phenomena. In Table 4, we present an analysis of linguistic phenomena manually annotated over 50 random episodes (similar to Ku et al. (2020) and Chen et al. (2019)). We analyze the following phenomena:

- **Class Reference:** refers to a semantic class of objects, furniture, or rooms in the scene. These references typically, but not necessarily, follow from the classes defined in OVMM.
- **Instance Reference:** refers to a unique object, furniture, or room; this disambiguation between entities of a class is typically achieved by visual description.
- **Co-Reference:** an expression that refers to an entity defined elsewhere in the instruction.
- **Passive Voice:** the instruction is phrased such that the object is receiving the rearrangement or state change; the request is typically asked instead of commanded.
- **Active Voice:** the instruction is phrased such that the rearrangement or state change is to be completed with an object; the request is typically commanded instead of asked.
- **High-Level Goal Spec:** this sets the operating context for the task before the particulars of rearrangement or state change are specified.
- **Agentic Reference:** a reference to one of the agents performing the task. Typically used to incite a suggested task division between the human and robot.

We observe that **PARTNR** tasks have a high rate of entity class references, such as *the table*, (6.38/episode), and common occurrences of instance references and co-references. This signals a need for capable natural language understanding, scene grounding, and co-reference resolution. Tasks and sub-tasks are predominantly issued using active voice (92%), but 14% of tasks include at least one occurrence of passive voice. Half of tasks involve a high-level goal specification, which commonly serves to reduce the search space. For example, a task starting with *Let's clean up all the toys in the playroom* constrains object search to that room and softly constrains the placement of those objects to locations in which toys would commonly be stored. Finally, agentic references are present in 14% of episodes.

Secondary Task Characteristics. In Table 5, We present an analysis of secondary task characteristics present in **PARTNR** as derived automatically from evaluation functions of all episodes in our dataset. We find rare but present occurrences of subset counts, where agents must reason about manipulating a subset of a set of objects (e.g. *bring two cups...* when more than two cups exist in the scene). We find that every episode contains at least one occurrence of resolvable ambiguity, where an instruction makes an object/furniture/room reference that may be resolved by more than one entity instance.

| Linguistic Phenomenon | p | μ | Example |
|-----------------------|-----|-------|-------------------------------|
| Class Reference | 100 | 6.38 | <i>The table</i> |
| Instance Reference | 12 | 0.14 | <i>The coffee table</i> |
| Co-Reference | 50 | 0.64 | <i>That, Those, it, ...</i> |
| Passive Voice | 14 | 0.18 | <i>Can you bring me...?</i> |
| Active Voice | 92 | 2.40 | <i>Set it on the stool.</i> |
| High-Level Goal Spec | 50 | 0.50 | <i>Let's tidy up. Move...</i> |
| Agentic Reference | 14 | 0.16 | <i>Do... While I...</i> |

Table 4: **Analysis of linguistic phenomena in the PARTNR dataset.** p is the % of instructions that contain the phenomenon while μ is the average number of times the phenomenon occurs within each instruction. A random sample of 50 Test episodes were included in manual annotation.

| Secondary Characteristics | p | μ | Example |
|---------------------------|-----|-------|--------------------------------------|
| Subset Count | 1 | 0.01 | <i>Bring two cups...</i> |
| Resolvable Ambiguity | 68 | 1.68 | <i>Move the pants to any chair.</i> |
| Dependent Rearrange | 7 | 0.09 | <i>Place them on the same table.</i> |
| Multi-Step Rearrange | 6 | 0.11 | <i>Cup to sink. Cup to cabinet.</i> |

Table 5: **Analysis of secondary task characteristics in the PARTNR dataset.** p is the % of instructions that contain the phenomenon while μ is the average number of times the phenomenon occurs within each instruction. All episodes in Test were included using automatic annotation.

Dependent rearrangements build on this task characteristic; when multiple objects/furniture/rooms can satisfy a sub-task, a task may require that same entity to solve another sub-task. An example of this is the same table being used to solve the placement of both a spoon and a bowl ("*bring the spoon and bowl to the same table in the living room.*"). This occurs in 31% of episodes and the resulting dynamic dependency is a challenging aspect of collaboration. Finally, 6% of tasks include multi-step rearrangement of the same entity. For example, moving a cup to the sink and then to the cabinet after it is washed. Tasks with multi-step rearrangement have two such rearrangements on average.

Distribution of Entities in PARTNR Tasks. In Figure 7, we examine the distribution of task-relevant objects, furniture, and rooms. We define task-relevant objects to be objects requiring rearrangement or a state change, task-relevant furniture to be target furniture for rearrangement, and task-relevant rooms to be target rooms for rearrangement. Object categories are shown in Figure 7a, furniture categories are shown in Figure 7b, and room categories are shown in Figure 7c. The PARTNR dataset contains a long tail of semantic object categories, and within those categories, a wide variety of objects. Thus, for agents to perform well in PARTNR tasks, they must display visual reasoning and collaboration behaviors that generalize across the semantic particulars of a task. The skew of distributions in Figure 7b and Figure 7c can be understood by the relative occurrences of rooms and objects in HSSD scenes, e.g. there are more tables than couches on average.

The episodes are completely unique not just through instruction, but also through randomization of object instances, initial object locations, count of clutter objects, and agent starting positions. So, the same low-level action trajectories would not transfer to a different task.

We run an analysis on the instructions, removing all nouns (as proxy for object ID), and checking for uniqueness. We find that 800 val-1k episodes and 5193 train-10k episodes retain their uniqueness, implying 80% uniqueness in val, and 52% uniqueness in scaled train episodes. Moreover, instructions are unique with a minimum edit distance of at least 5 to account for slight variation.

A.3 SIMULATION-IN-THE-LOOP LARGE-SCALE TASK GENERATION

In this section, we describe in detail the simulation-in-the-loop task generation pipeline. We follow a 4-step generation pipeline:

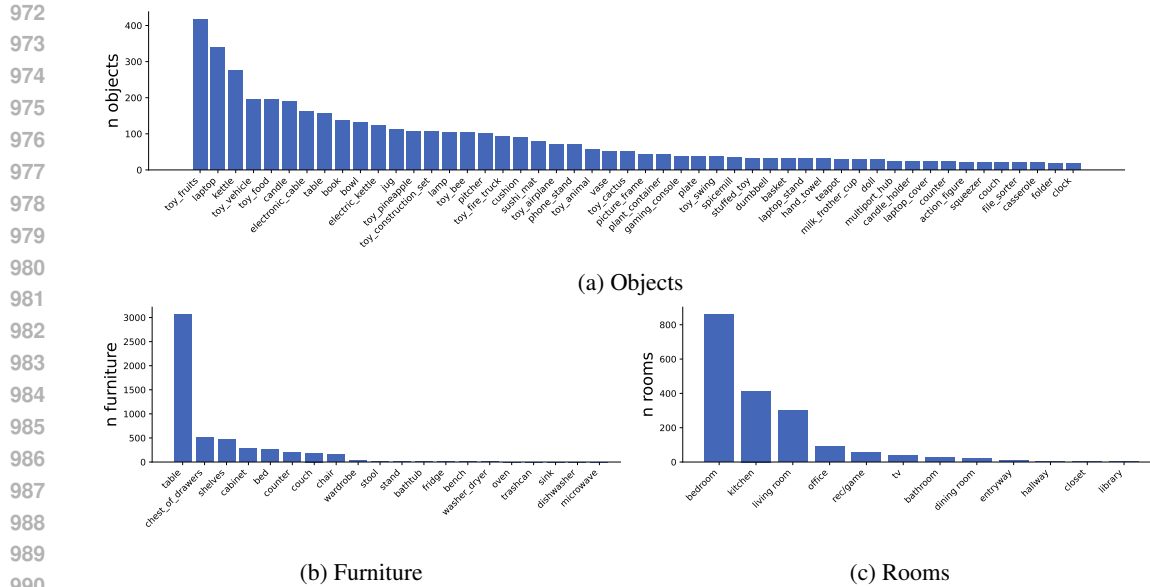


Figure 7: **The distribution of task-relevant object categories, furniture categories, and room categories in the PARTNR dataset.** The object distribution is truncated to the 50 most common entities, down from 155 total object categories. The count of entities is derived by the total number of occurrences of that entity in the Test split evaluation data.

1. Simulation-in-the-loop small-scale free-form LLM generation:

In **PARTNR**, we initiate the process by setting up a simulation environment using the Habitat 3.0 simulator, populated with the HSSD dataset which includes 60 unique houses and 5,819 OVMM objects. This simulated environment is parsed to identify a list of rooms and available furniture. This information, along with a list of all available objects, is then passed to a Language Model (LLM). The LLM utilizes this data to generate free-form, viable tasks within the scene, accompanied by an initial scene state description. For instance, if the task is "Clear dishes from the living room," the LLM would generate an initial scene depicting multiple dishes in the living room. To add complexity, additional objects are introduced to create clutter.

2. Simulation-in-the-loop filtering and annotation.

The tasks and initial states generated by the LLM are instantiated within the simulator. At this stage, tasks that are infeasible due to the layout of the house or the capabilities of the simulator are filtered out. For example, a task like "Clear dishes from the living room" would be discarded if the simulated house lacks a living room. Similarly, tasks requiring unsupported actions, such as "folding," are also filtered out. This filtering process is crucial as it significantly reduces the number of unrealistic or impossible tasks, although it also tends to limit the diversity of the generated instructions. The full list of allowed actions in the simulator are: [open, close, power on, power off, move, clean, fill, pour, pick, place] and related actions. Next, we use manual annotation to diversity of the resulting tasks. To counter the limited diversity resulting from automated filtering, we manually annotate the tasks to ensure a balanced distribution of various types of tasks, such as constraint-free, spatial, temporal, and heterogeneous tasks. This manual intervention involves modifying the instructions to incorporate different objects and settings, resulting in a curated set of 1,000 human-annotated and simulation-verified tasks. This step is essential for maintaining quality and diversity but is impractical for scaling up due to its labor-intensive nature.

3. Large-scale generation.

To scale the generation of diverse tasks without extensive manual effort, we leverage the 1,000 human-annotated instructions as examples in the LLM prompts. By providing the LLM with both a house description and an example task, we instruct it to adapt the task

to fit the new setting, such as changing "Clear all dishes from the living room" to "Clear all toys from the bedroom." This approach helps preserve the diversity of the tasks while enhancing the likelihood of successful instantiation in the simulator.

4. Human-in-the-loop filtering.

The scaled tasks undergo a final human-in-the-loop filtering where physically infeasible instructions are eliminated, ensuring the practicality and realism of the tasks. This ensures that tasks such as "Move 4 cups to the dining table", where there are only 2 cups in the scene are removed. Or "First move the cup from the kitchen to the living room, then place a jug in the kitchen, next to the cup", which consists of a physically infeasible instruction.

A.3.1 HABITAT 3.0 AND HSSD EXTENSION

We generate **PARTNR** using modified HSSD scenes [Khanna et al. \(2024\)](#) and the Habitat 3.0 [Puig et al. \(2024\)](#) simulator due to its humanoid simulation capabilities and availability of features which support modeling of collaborative tasks as discussed in Appendix A.1.

Our extensions to the Habitat platform include a set of features targeting: object state manipulation (e.g., clean/dirty, powered on/off and filled/un-filled), evaluation of object relative relationships (e.g., next-to, above, within, on-top, on-floor, in-region, etc), and procedural clutter generation utilities enabling generation of valid initial scene contents pre-conditioned on the output of LLM-generated requirements from open-language prompts (see Section 3). For example, in order to evaluate the task of rearranging a tea set from furniture A to furniture B, we must first generate a scene with both types of furniture in accessible locations and a tea set already sitting on or inside of furniture A.

A.4 THE **PARTNR** EVALUATION SYSTEM

In this section, we formalize the components of the evaluation system, define the resulting metrics, and present the details and prompts used for LLM-based evaluation generation.

A.4.1 EVALUATION PREDICATES

We use logical predicates to query the state of objects, furniture, and rooms at the current timestep in the simulator. The evaluation system operates on the resulting binary state values. The details of all logical predicates are in Table 6.

A.4.2 PROPOSITIONS

The primary component of a task evaluation function is a list of propositions. We define a proposition as an evaluation predicate instantiated with argument values. Propositions additionally enable the evaluation of instructions with ambiguous references ("*on a table*" — which table?) and subset counts ("*two spoons*" — any two of the n total spoons). Ambiguity is enabled by extending the predicate arguments to lists. Subset counts are enabled by optional arguments `number`, which defines the subset size, and `arg_match`, which is a boolean indicating whether all entities in the subset must be satisfied with the same second argument. Suppose we want to evaluate the task "*Bring a spoon to the table.*". If we have a single spoon and a single table, the proposition is straightforward:

```
is_on_top([spoon_1], [table_1]).
```

If we have multiple spoons and need just one (an ambiguous instruction), the following proposition is used:

```
is_on_top([spoon_1, spoon_2, spoon_3], [table_1]).
```

In the above case, a list is treated as a OR of entities. The same holds for multiple tables:

```
is_on_top([spoon_1, spoon_2, spoon_3], [table_1, table_2]),
```

in which any spoon may be placed on any table. If the instruction specifies bringing two spoons to the table, the `number=2` argument is added:

```
is_on_top([spoon_1, spoon_2, spoon_3],
          [table_1, table_2], number=2).
```

1080
 1081
 1082
 1083
 1084
 1085
 1086
 1087
 1088
 1089
 1090
 1091
 1092
 1093
 1094
 1095
 1096
 1097
 1098
 1099
 1100
 1101
 1102
 1103
 1104
 1105
 1106
 1107
 1108
 1109
 1110
 1111
 1112
 1113
 1114
 1115
 1116
 1117
 1118
 1119
 1120
 1121
 1122
 1123
 1124
 1125
 1126
 1127
 1128
 1129
 1130
 1131
 1132
 1133

| Predicate Name | Category | Description |
|--|-----------|--|
| <code>is_on_top(o₁, o₂)</code> | Rearrange | o_1 is considered on top of o_2 if a downward raycast from any of the o_1 bounding box corners intersects with o_2 . |
| <code>is_inside(o₁, o₂)</code> | Rearrange | o_1 is considered inside o_2 if a threshold number of opposing raycasts hit the same object. Ray casts are 1.00m and the threshold number is 2. |
| <code>is_in_room(o₁, r₁)</code> | Rearrange | o_1 is considered to be in room r_1 if at least 25% of keypoints (bounding box corners and center) are contained in the 3D region. |
| <code>is_on_floor(o₁)</code> | Rearrange | o_1 must be within 0.04m vertically of the navigation mesh. |
| <code>is_next_to(o₁, o₂)</code> | Spatial | The bounding box of o_1 must overlap vertically with the bounding box of o_2 and the horizontal L2 distance between bounding boxes is less than or equal to 0.50m. |
| <code>is_clustered(o₁, ..., o_n)</code> | Spatial | Each o_i satisfies <code>is_next_to(o_i, o_j)</code> for some $j \neq i$. |
| <code>is_clean(o₁)</code> | State | o_1 has affordance <code>cleanable</code> and the state machine indicates that o_1 is clean. |
| <code>is_dirty(o₁)</code> | State | o_1 has affordance <code>cleanable</code> and the state machine indicates that o_1 is not clean. |
| <code>is_filled(o₁)</code> | State | o_1 has affordance <code>fillable</code> and the state machine indicates that o_1 is filled. |
| <code>is_empty(o₁)</code> | State | o_1 has affordance <code>fillable</code> and the state machine indicates that o_1 is not filled. |
| <code>is_powered_on(o₁)</code> | State | o_1 has affordance <code>powerable</code> and the state machine indicates that o_1 is powered on. |
| <code>is_powered_off(o₁)</code> | State | o_1 has affordance <code>powerable</code> and the state machine indicates that o_1 is powered off. |

Table 6: **Logical predicates to evaluate the relations of objects, furniture, and rooms in the PARTNR.** Predicates exist for measuring rearrangement of objects (category=Rearrange), spatial placements relative to objects or furniture (category=Spatial), and states of objects or furniture (Category=State). For all predicates, $o_i \in \{\text{object, furniture}\}$ and r_i is a room.

1134 Finally, the instruction may specify that two spoons should be brought to the *same* table.
 1135 `arg_match=True` enables this:

```
1136         is_on_top([spoon_1, spoon_2, spoon_3],
1137                 [table_1, table_2], number=2, arg_match=True).
```

1139 Propositions are represented as JSON within dataset files and as Python function calls (as shown
 1140 above) during human annotation for interpretability.

1143 A.4.3 DEPENDENCIES

1144 Evaluation functions operate over a sequence of simulation states. By default, all propositions in
 1145 the list of evaluation propositions will be evaluated at every time step. However, many tasks in
 1146 the **PARTNR** benchmark involve a required temporal order during execution. Take for example the
 1147 instruction “*Move the cup from the table to the sink. Then, return the cup to the table.*” This consists
 1148 of a multi-step rearrangement where the proposition checking that the cup is on the table is *dependent*
 1149 on a different proposition first being satisfied. We define a proposition dependency as the following:

```
1150     PropositionDependency(  
1151         proposition_indices, depends_on, relation_type),
```

1153 where the argument `proposition_indices` is a list of integers indicating the dependent propo-
 1154 sitions, the argument `depends_on` is a list of integers indicating the head propositions, and the
 1155 argument `relation_type` indicates the condition that the `depends_on` propositions must take
 1156 in order for `proposition_indices` to be evaluated for satisfaction. The following relation types
 1157 are supported:

- 1158 • **after_satisfied**: evaluate propositions in `proposition_indices` after the propositions
 1159 in `depends_on` have been satisfied.
- 1161 • **after_unsatisfied**: evaluate propositions in `proposition_indices` after propositions
 1162 in `depends_on` have been satisfied at some point in the past but are no longer satisfied.
- 1163 • **while_satisfied**: evaluate propositions in `proposition_indices` when propositions in
 1164 `depends_on` are currently satisfied.

1165 As a concrete example, suppose the instruction is “*Move the ball and bat to the kitchen and set them
 1166 next to each other. Then, move them to the closet.*” The propositions for this task would be

```
1167     0 is_in_room([ball], [kitchen])
1168     1 is_in_room([bat], [kitchen])
1169     2 is_next_to([ball], [bat])
1170     3 is_in_room([ball], [closet])
1171     4 is_in_room([bat], [closet]).
```

1174 and the dependencies would be

```
1175     0 PropositionDependency([2], [0, 1], "while_satisfied")
1176     1 PropositionDependency([3, 4], [0, 1], "after_satisfied").
```

1178 The first ensures that the `is_next_to` predicate is only queried during kitchen placements and
 1179 the second ensures that placing the ball and bat in the closet is not checked at the start of the task
 1180 execution, which would be inadvertently satisfied upon scene initialization. Each evaluation function
 1181 has a (possibly empty) list of such proposition dependencies.

1183 A.4.4 CONSTRAINTS

1184 **PARTNR** tasks often require constraining *how* tasks are completed. Examples of this include complet-
 1185 ing evaluation propositions in temporal order (“*Do [x] before doing [y]*”) and enforcing links between
 1186 ambiguities (“*Fill one of the cups, then put that cup on the table*”). We define a set of constraint
 1187 types that enables evaluating such task complexities:

- **TemporalConstraint:** A directed acyclic graph (DAG) over the indices of propositions that defines the temporal requirement of when propositions should be satisfied. If propositions are satisfied out of order, then the task is marked unsuccessful. The temporal constraint is defined as

`TemporalConstraint (graph_edges),`

where `graph_edges` is a set of pairwise temporal constraints. For example, an edge “0 → 1” indicates that the proposition at index 0 must be satisfied at an earlier time step than the proposition at index 1.

- **SameArgConstraint:** Requires the argument used to satisfy a proposition to be the same within a pre-determined set of propositions. The same argument constraint is defined as

`SameArgConstraint (proposition_indices, arg_names),`

where `proposition_indices` is a list of proposition indices to link and `arg_names` is a list specifying which component of each proposition to link.

- **DifferentArgConstraint:** Requires the argument used to satisfy a proposition to be unique within a pre-determined set of propositions. The different argument constraint is defined as

`DifferentArgConstraint (proposition_indices, arg_names),`

where `proposition_indices` is a list of proposition indices to link and `arg_names` is a list specifying which component of each proposition to link.

- **TerminalSatisfactionConstraint:** Some propositions, once satisfied, should remain satisfied at the end of the episode. Others are expected to become unsatisfied, such as in multi-step rearrangements or object state changes of a single object. The terminal satisfaction constraint is defined as

`TerminalSatisfactionConstraint (proposition_indices)`

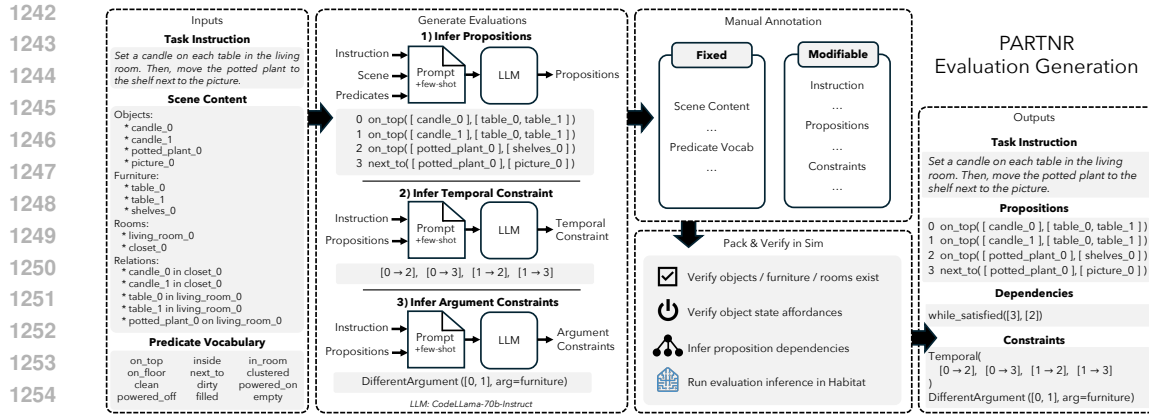
where `proposition_indices` is a list of proposition indices that should be satisfied at time $t = T$ for an episode rollout of duration T .

A.4.5 EVALUATION METRICS

A task evaluation function is constituted of propositions, dependencies, and constraints as defined above. From this data, the task evaluation function serves to determine what percentage of the task is complete for a given human-robot collaboration rollout. From a sequence of simulation states, we evaluate the truth values of each proposition with respect to both the dependencies (*when* the propositions must be evaluated) and the constraints (*how* the propositions must be satisfied). The Percent Completion (PC) is defined as the ratio of satisfied propositions to the total number of propositions. Success is defined as $S := (PC = 100)$. Both Percent Completion and Success metrics solely evaluate task completion and thus are agent-agnostic. Given that **PARTNR** is designed to evaluate multi-agent collaboration, we note that these metrics can be combined with duration-based metrics (e.g. simulation steps or time) to measure efficiency; multi-agent aspects like task division and partnered exploration serve to optimize task efficiency. The benefit of an agent-agnostic metric is flexibility to evaluate any number of agents performing the task with respect to the high-level goal.

A.4.6 EVALUATION FUNCTION GENERATION

This section provides detail on evaluation function generation beyond the overview provided in Section 3. In particular, Figure 8 shows the first two steps of Figure 2 (sim-in-the-loop generation and filtering + annotation) in greater detail as they pertain to evaluation generation. Notably, we take a three step process to generating evaluation functions; first, an LLM generates a list of evaluation propositions which are parsed into a usable format. Then, an LLM infers the temporal constraint over these propositions by predicting topological generations of a temporal graph. For example, the prediction $[[0, 1], [2, 3]]$ implies that propositions at indices 0, 1 can be completed in either order, and both must be completed before propositions at indices 2, 3. This prediction is stored and evaluated more generally as edges of a directed acyclic graph (DAG). We found that the assumption of topological generations sufficiently expresses the tasks in **PARTNR** while being simpler for



1256 **Figure 8: Evaluation function generation pipeline overview.** A three-step LLM-generation process
1257 produces propositions and constraints during *Generate Evaluations*. The evaluation function then
1258 is saved to a file for optional human annotation during *Manual Annotation*. Finally, the evaluation
1259 function is packed and verified in simulation during *Pack & Verify in Sim* to ensure that all entities
1260 and affordances exist.

1261
1262
1263

| Task | Correctness (%) | |
|-----------------|-----------------|----------|
| | Evaluation | Combined |
| Constraint-Free | 89 | 97 |
| Spatial | 87 | 85 |
| Temporal | 91 | 93 |
| Heterogeneous | 93 | 91 |
| Average | 90 | 92 |

1264
1265
1266
1267
1268

1269
1270 **Table 7: Manually-annotated generation accuracy of 100k-scale PARTNR tasks and evaluation**
1271 **functions.** Altogether, we find that 83% of episodes are generated without any task or evaluation
1272 function errors. Analysis performed on 100 sampled episodes of each task type via PrediViz.

1273
1274 an LLM to generate than a DAG. The third step is predicting argument constraints, in which
1275 the LLM is provided the instruction and propositions list and must predict a list of constraints,
1276 either `SameArgConstraint` or `DifferentArgConstraint`. All LLM queries are performed
1277 against CodeLlama70b-Instruct (Roziere et al., 2023). All prompts used for evaluation function
1278 generation are included in Appendix A.14.

1280 A.5 HUMAN ANNOTATION AND ACCURACY ASSESSMENT FOR PARTNR DATA

1282 A.5.1 GENERATION ACCURACY FOR TASKS AND EVALUATIONS

1283
1284 It is important for tasks in PARTNR to be solvable by collaboration agents and for the associated
1285 evaluation functions to accurately reflect the task being performed. In this section, we analyze the
1286 accuracy of 100k-scale PARTNR generation with respect to both of these criteria using the PrediViz
1287 tool (Appendix A.5.2). In Table 7, we demonstrate that the accuracy of task generation ranges from
1288 87-93% depending on task type and averages 90%. The accuracy of evaluation generation ranges
1289 from 85-93% depending on task type and averages 92%. Combining these numbers yields an overall
1290 joint accuracy of 83% for our 100k-scale dataset. In Table 8, we annotate the failure modes that lead
1291 to unsolvable tasks and incorrect evaluation functions. Common task-related failure modes include:

- 1292 • **Hallucination.** The produced instruction references objects, furniture, or rooms that do
1293 not exist in the scene the instruction was generated for. Example: "Move the clothes to the
1294 washing machine." produced for an environment that does not contain a washing machine.
- 1295 • **Unresolvable Ambiguity.** The produced instruction contains ambiguous directives that
cannot be reasonably resolved without further communication or a detailed understanding of

| | | Task Failures | | Evaluation Failures | |
|-----------------|--|------------------------|---|---|---|
| | | Mode | % | Mode | % |
| Constraint-Free | | Hallucination | 7 | Incorrect Ambiguity | 2 |
| | | Already Satisfied | 2 | Incorrect Furniture | 1 |
| | | Contradiction | 2 | - | - |
| Spatial | | Hallucination | 6 | Incorrect Temporal Grouping | 7 |
| | | Unresolvable Ambiguity | 5 | Incorrect Predicate (Other) | 5 |
| | | Already Satisfied | 2 | Incorrect Predicate (Room vs Furniture) | 3 |
| Temporal | | Hallucination | 4 | Incorrect Temporal Grouping | 3 |
| | | Contradiction | 3 | Incorrect Predicate (Other) | 3 |
| | | Already Satisfied | 2 | Incorrect Predicate (Room vs Furniture) | 1 |
| Heterogeneous | | Hallucination | 3 | Incorrect Predicate (Room vs Furniture) | 5 |
| | | Unresolvable Ambiguity | 2 | Incorrect Object/Furniture/Room | 2 |
| | | Contradiction | 1 | Incorrect Predicate (Other) | 2 |

Table 8: **Top three failure modes of 100k-scale task and evaluation generation reported for each task type.** Failures of task generation are led by the hallucination of non-existent entities, while failures of evaluation generation are led by incorrect temporal predictions and incorrect predicate functions. Analysis performed on 100 sampled episodes of each task type via PrediViz.

the task-issuer’s preferences. Example: *"Set the table for dinner"*; how many place settings are necessary? Should we set the formal dining table or nook table? What cutlery is needed?

- **Contradiction.** The produced instruction involves two or more sub-tasks that cannot simultaneously be satisfied. Example: *"Set the scissors on the coffee table. Set the bowl on the counter next to those scissors."*
- **Already Satisfied.** The produced instruction dictates sub-tasks that are all already satisfied at the start of the episode. Example: *"Move the laptop to the living room and turn it on"*, when the laptop is already powered on and in the living room.

According to Table 8, hallucinations are the most common failure mode for task generation. While simulation-in-the-loop filtering avoids this issue for evaluation generation, such filtering is inconsistent for tasks; language has a looser grounding to scene entities than statements of propositional logic. For example, a home might not have a formal dining room, but a table in the living room may serve the purpose of a dining table. Moving on to evaluation generation, common failures are as follows:

- **Incorrect Temporal Grouping.** The instruction implies a temporal order among sub-tasks (either explicitly via sequencing words, or implicitly via multi-step manipulations) and the predicted temporal constraint fails to reflect this order over the propositions. Example: allowing propositions to be satisfied in any order for the task *"First, return the plates to the kitchen. Then, tidy up the living room."*
- **Incorrect Predicate (Other).** The evaluation function uses the wrong predicate function to evaluate the task. Example: using `is_powered_on` instead of `is_filled` when the instruction asks to fill the kettle.
- **Incorrect Predicate (Room vs Furniture).** A task specifies that an object should be rearranged to another room, but the propositions overly-constrain the rearrangement to a target furniture for placement. This failure is separate from the one above because it is the most common, and it indicates the tendency for the LLM to produce single solution instances rather than reflect the full space of ambiguity. Example: producing a proposition like `is_on_top([electronic_cable], [bed])` for the task *"Move the electronic cable to the bedroom."*
- **Incorrect Object/Furniture/Room.** incorrect entities are selected for satisfying proposition. Example: the instruction calls for rearranging the cushion to a living room table, but `table_4` in the proposition `is_on_top([cushion_0], [table_4])` exist in bedrooms.
- **Incorrect Ambiguity.** For an instruction that can be satisfied n different ways, the evaluation affords m options for solution, where $m \neq n$. Example: *"Move a toy to the kid’s room"*,

where two or more toys exist in the toy bin but the evaluation function does not list out all possible toys for rearrangement.

According to Table 8, the primary failures modes of evaluation generation are incorrect temporal grouping and incorrect predicates. Regarding temporal grouping, we observe that many sub-task require multiple propositions to evaluate. Grouping these propositions consistently within the temporal constraint is a source of error. Take for example the instruction *"Set the shirt and pants next to each other on the counter. Then, move them to the dresser."* In this case, five propositions will exist; four for placements and one for the spatial relation. The temporal prediction may erroneously link the spatial relation with the dresser placements rather than the counter placements.

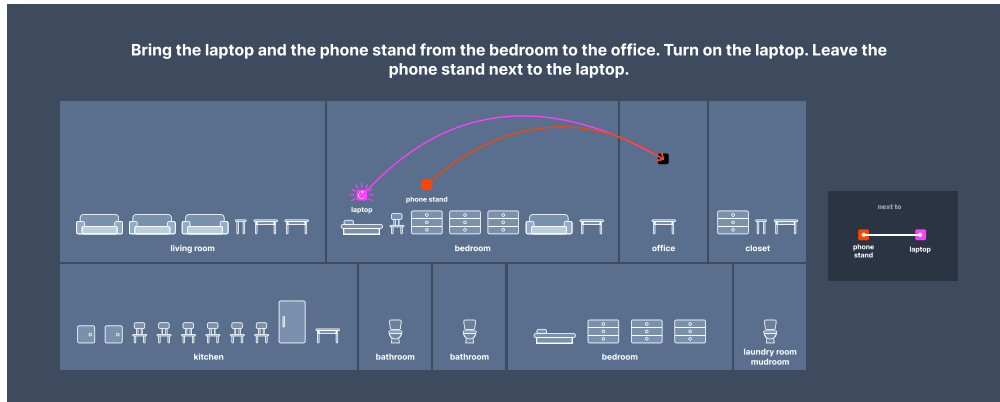
On average, this process takes 30 seconds per task. To further ensure task feasibility, we use human-in-the-loop to solve the tasks. This ensures that criteria such as object reachability and the absence of conflicting task constraints are met. This process is easily scalable on the web, as it does not require expert annotation, and takes 2 minutes 6 seconds per task on average. Our analysis indicates that automatic generation significantly reduces annotation time by a factor of 7x.

A.5.2 VISUALIZATION OF **PARTNR** TASKS AND EVALUATION FUNCTIONS

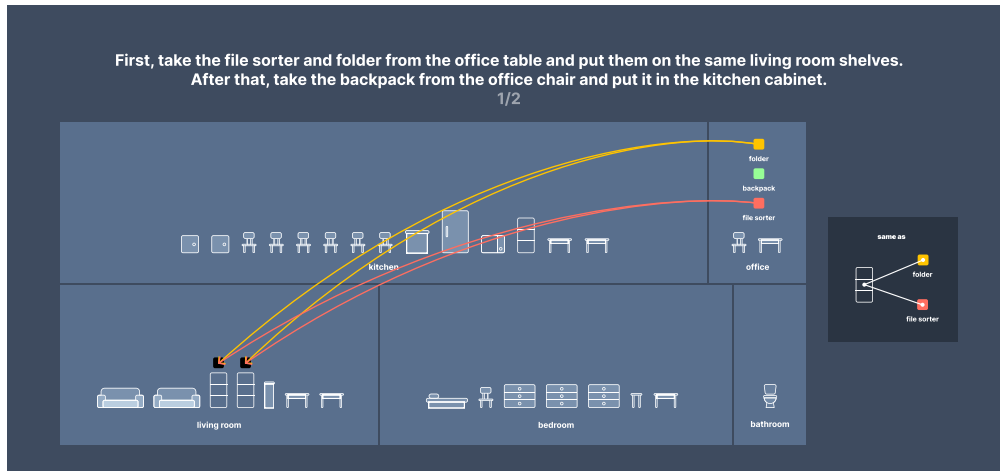
LLM-generated **PARTNR** episodes may contain errors so we evaluate their correctness with human annotators. This is a verification problem in which the generated instruction, evaluation function, and contents of the scene must be compared against each other. See Figure 6 for an example of this in code. To make this process faster, easier, and more accurate, we designed a visualization and annotation system (PrediViz) that illustrates the state of the world and the evaluation function relative to a task instruction (Figure 9). We chose a 2D illustration style to capture the relevant structure in the data:

- **Rooms.** Each room is drawn as a box with a name provided underneath. Names repeat if there are multiple instances of that room category in the scene. We model the rooms as a fully-connected graph of accessibility. We treat rooms as spatially independent and reorder or wrap them as needed.
- **Objects.** We visualize objects as boxes with category names underneath. Each proposition of the evaluation function is assigned a unique color to separate them from other propositions. For consistency, objects are colored by the color of the first proposition they appear in.
- **Receptacles.** Receptacles are the furniture in each room. We designed a bespoke set of icons for the 25 categories of furniture such that they can be easily subitized – you can tell a chair is a chair and a table is a table just by glancing at it.
- **States.** Both objects and receptacles have states that change over time (empty/filled, dirty/clean, powered on/off). These states are displayed using motifs for objects and textual labels for receptacles, when necessary.
- **Affordances.** Different furniture affords different object placements. For example, a cup can be placed *on top* of the fridge or *inside* it, but only *on top* of a table. These relationships are annotated by hand at the category-level. We visualize the target affordance as a dark box indicating where an object is supposed to go. The initial affordances are provided with the metadata to initialize the objects on corresponding receptacles or rooms.
- **Placements.** To represent the requested movement of an object, we draw an arrow from the object's initial position to the intended position. A simple placement is represented by a single arrow. Multiple allowable placement targets (e.g. multiple tables in the room) are represented by multiple arrows. We use solid arrows for AND placements (e.g. *"place the doll and the toy truck on the couch"*) and dotted arrows for OR placements (e.g. *"place the doll or the toy truck on the couch"*). We also have support for choosing k out of n objects for placement (e.g. *"place two out of three dolls on the couch"*), for which we use a numerical label on the dotted lines.
- **Temporal Constraints.** To visualize temporal constraints, we split the instruction into multiple frames. For example, if the instruction requires us to *"place the doll on the couch and the toy truck in the chair; then put the stuffed toy inside a chest of drawers,"* we create one frame to represent the first half of the statement (i.e. *"place the doll on the couch and*

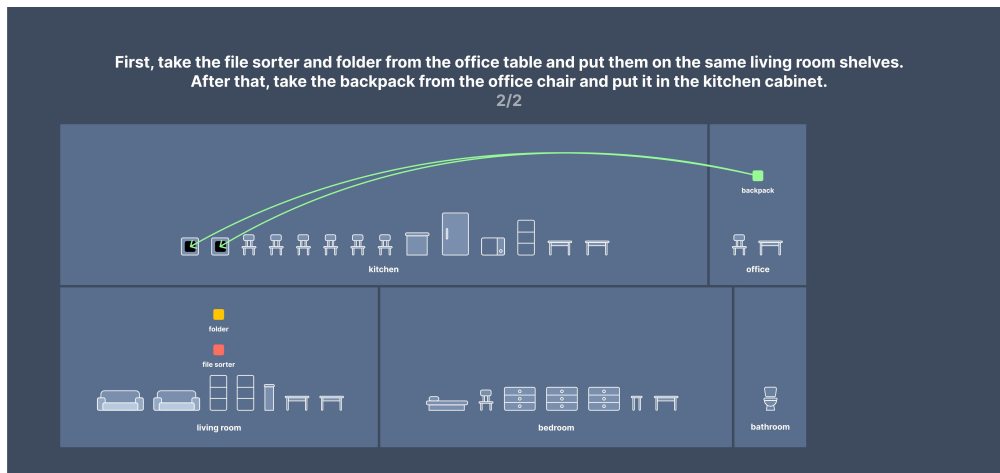
1404
 1405
 1406
 1407
 1408
 1409
 1410
 1411
 1412
 1413
 1414
 1415
 1416
 1417
 1418
 1419
 1420
 1421
 1422
 1423
 1424
 1425
 1426
 1427
 1428
 1429
 1430
 1431
 1432
 1433
 1434
 1435
 1436
 1437
 1438
 1439
 1440
 1441
 1442
 1443
 1444
 1445
 1446
 1447
 1448
 1449
 1450
 1451
 1452
 1453
 1454
 1455
 1456
 1457



(a) Example task #1



(b) Example task #2: first temporal frame



(c) Example task #2: second temporal frame

Figure 9: **PARTNR** tasks visualized in **PrediViz**. The design distills the task and scene to only the components necessary for verification. In example task #2, the split frames signify that the agents must first rearrange the file sorter and folder (Figure 9b), then rearrange the backpack (Figure 9c).

1458 *the toy truck in the chair") and another for the second half (i.e. "put the stuffed toy inside a*
 1459 *chest of drawers").*

1460 • **Special Relations.** We also illustrate special relations like `next_to`, `same_as`, and
 1461 `different_from` in the style of an informational legend shown on the side.
 1462

1463 Resulting visualizations are wrapped in a web-based annotation tool that affords binary verification
 1464 and failure model labeling. See Figure 8 for results derived from this tool. We ran a small-scale
 1465 experimental study (n=22) comparing human annotation using PrediViz to a text-based representation.
 1466 Using PrediViz, verification was 2.6 times faster, 8% more accurate, and perceived as 24% easier.
 1467

1468 A.5.3 HUMAN-ASSISTED CORRECTION ANNOTATION

1469 Below is an example task and evaluation function saved in plain text for human annotation. The
 1470 instruction, propositions, and constraints can be modified as necessary to ensure the task is feasible
 1471 and that the evaluation function reflects it. Annotators also have access to a file containing the objects,
 1472 furniture, rooms, and relations thereof to reference during this process.
 1473

```

1474                                     Dataset Correction Annotation Trial
1475
1476 # type: ignore
1477
1478 # -----
1479 # INSTRUCTION
1480 #   modify as necessary, but keep in mind the scene is fixed.
1481 # -----
1482 instruction = """
1483 Help me organize the entryway. First, place the phone, watch, and keychain on the table
1484 next to each other.
1485 """
1486
1487 # -----
1488 # PROPOSITIONS
1489 #   is_on_top(objects, receptacles, number=1, is_same_receptacle=False)
1490 #   is_inside(objects, receptacles, number=1, is_same_receptacle=False)
1491 #   is_in_room(objects, rooms, number=1, is_same_room=False)
1492 #   is_next_to(entities_a, entities_b, number=1, is_same_b=False, l2_threshold=0.5)
1493 #   is_on_floor(objects, number=1)
1494 #   Args:
1495 #     objects/receptacles/entities_*: OR of a list
1496 #     number: n objects/entities_a must satisfy
1497 #     is_same: the same entity must satisfy all n objects
1498 # -----
1499 propositions = [
1500     is_on_top(['cellphone_0'], ['table_4']),
1501     is_on_top(['watch_0'], ['table_4']),
1502     is_on_top(['keychain_0'], ['table_4']),
1503     is_next_to(['cellphone_0'], ['watch_0']),
1504     is_next_to(['watch_0'], ['keychain_0']),
1505 ]
1506
1507 # -----
1508 # TEMPORAL GROUPS
1509 #   Place propositions in groups s.t. one group must be satisfied before the next.
1510 #   Example:
1511 #     [ [0, 1], [2, 3] ] means props 0 & 1 must be satisfied before props 2 & 3.
1512 # -----
1513 temporal_groups = [
1514     [0, 1, 2, 3, 4],
1515 ]
1516
1517 # -----
1518 # TIE CONSTRAINTS
1519 #   options: SameArgConstraint, DifferentArgConstraint
1520 #   Args:
1521 #     proposition_indices: List[int]
1522 #     arg_indices: List[int]
1523 #   Example:
1524 #     SameArgConstraint([0, 2], [1, 1]). Means: Propositions 0 & 2 must
1525 #     match values on the argument at argument index 1 and 1, respectively.
1526 # -----
1527 tie_constraints = [
1528 ]

```

```

1512 # -----
1513 # TERMINAL SATISFACTION CONSTRAINT:
1514 # We assume all propositions must remain satisfied to the end of the episode.
1515 # if a proposition *should* become unsatisfied, add its index here.
1516 # -----
1517 exclude_final_constraint = []
1518 # -----
1519 # mark True if the task has a fatal issue
1520 # -----
1521 skip_episode = False
1522 reason = ""

```

A.6 WORLD GRAPH: THE PERCEPTION FRAMEWORK FOR LLM AGENTS

Scene-graph style hierarchical graphs have been shown to be effective for planning problems [Gu et al. \(2024\)](#); [Agia et al. \(2022\)](#); [Rana et al. \(2023\)](#). Inspired by such prior work, as illustrated in Figure 10, our world graph is a hierarchical multi-edge directed graph with $K = 3$ levels for representing the entities in the world. The nodes at first level correspond to rooms in the environment, followed by furniture at second, and objects and agents at third level. The root of this graph is an abstract house-node denoting the environment where tasks are taking place. Apart from the semantic information, each node also stores the 3D location of the entity and its affordance states e.g., clean/dirty, on/off, open/close, etc. The graph can then be serialized or accessed via specialized tools by ReAct policies. Prompts in Section A.15 provide an example of how the world graph is used by the LLMs in our baselines.

A.6.1 BUILDING AND UPDATING THE WORLD GRAPH

The initial graph is built by reading the the ground-truth room region annotations and furniture placements associated with a particular scene. The location of furniture and the region boundaries are used to associate each furniture to a specific room. This creates the initial two tiers, Room and Furniture, of the privileged world-graph. In partial-observability setting this is all the planner gets to start planning. However, in the full-observability setting, object-to-furniture assignments are read from each episode’s initialization information and a third Object tier is added to the world-graph.

Under partial-observability setting during execution, we use the panoptic sensors attached to both agents to detect all visible objects in the current-frame. Then ground-truth simulation information is used to extract the housing furniture or agent for each of these objects, as well as the location, and this new information is added to the maintained graph. For full-observability setting, each frame latest graph is mined from simulator using current information for objects and overwrites the previous world-graph.

It is not guaranteed in our setup that the images will pick up an object that was placed, filled or powered on/off in the previous step. Therefore, in partial-observability setting, we also add deterministic updates to the graph based on previous action and its result, e.g. successful placement triggers deletion of edge between agent who was holding the object and an addition of edge between placed object and the furniture it was placed on per action arguments.

A.6.2 NON-PRIVILEGED PERCEPTION USING CONCEPTGRAPHS

In order to study the dependence of the planner on the underlying world-graph, we also follow a modified version of ConceptGraphs (CG) pipeline [Gu et al. \(2024\)](#) to create the initial layout that is used to initialize our world-graph representation.

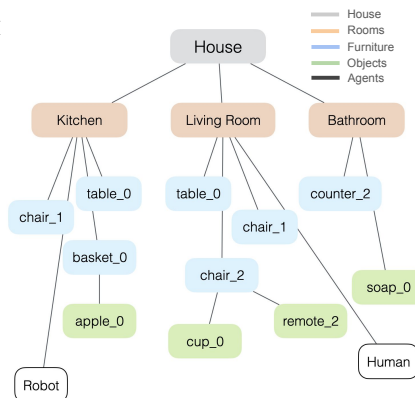


Figure 10: **World graph.** The high-level policies of the human and robot agents leverage a hierarchical world graph containing information about rooms, furniture, and objects in the environment. This graph is updated based on the observations and actions of both agents.

Modified Pipeline for Creating ConceptGraphs. We adapt the original CG pipeline to only use Meta-CLIP models for getting object name and category; using YOLO and SAM for object segmentation; and LLaMA3.1 for annotating room-label and inter-object spatiosemantic relations. Instead of using LLaVA and GPT for getting open-vocab descriptive object names, we use multi-perspective averaged CLIP embeddings of each object to classify its category given our closed object-vocabulary. For the required room-to-furniture relations, we extend CG pipeline by adding another prompt and query Llama3.1 model to assign room-labels to each entity given the categories of 10 closest entities to it.

Updating ConceptGraphs. Like the privileged world-graphs there are two modes of updating a ConceptGraphs-initialized non-privileged world-graph:

1. **Observations.** Using the same panoptic sensors we extract all the visible objects in current frame for both the agents. We use depth-sensors to extract the point-cloud associated with this object. We use this 3D location to first check if this is a redetection of a previously detected object. We use location, category and whether object-is-being-held-by-agent features to assess if this is a degenerate detection. If this is a new object detection then we use its location along-with bounding-boxes of existing furniture to check if this object is contained-within or on-top-of any of the recognized furniture-pieces. If it is then the world-graph is updated with this node and the edge.
2. **Actions.** Just like privileged world-graph, we can not guarantee sensors will pick up changed state of an object that is placed, powered, etc. Thus we add similar action based updates to this version as well. A special case is when non-privileged graph is updated by human agent’s action arguments which are grounded in privileged world-graph and may refer to same physical entity with different given names, e.g. `backpack_0` is `backpack_153` in non-privileged graph. We use a simple proximity and category matching heuristic to match human’s arguments to known entities in non-privileged graph, falling back on proximity based matching when no entity of same category are found.

Using ConceptGraphs with Simulated Skills Simulated skills require sim-handles of the placement furniture to snap objects onto them. The furniture in non-privileged graph do not have these sim-handles by design. Thus we come up with a simple proximity and category-name based matching to match a ground-truth furniture entity to a detected furniture-entity. If we can not find any ground-truth entity of the same category close to the detected furniture, we fallback on matching to the closest entity.

Prompts and Models used in ConceptGraphs Creation. CLIP model and pretrained backbone checkpoints: ViT-H-14-quickgelu, metaclip_fcc. Object detector: YOLO checkpoint yolov8x-worldv2. LLM: Llama3.1-70B-Instruct

Room Annotation Prompt for Modified CG

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>You are an expert on house
layouts. You will be given an input which will describe QUERY_OBJECT. This object will
be described by its name and the 10 pieces of furniture closest to it. You will assign a
```

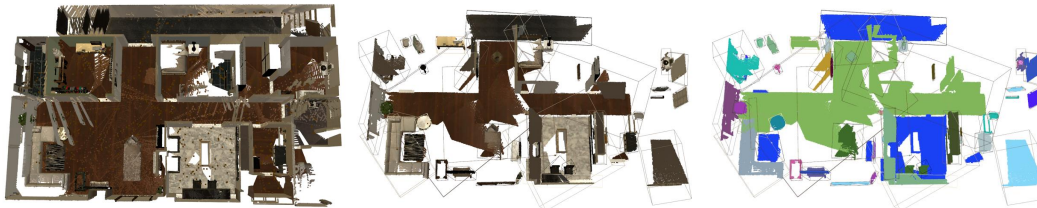


Figure 11: **ConceptGraph pipeline.** Left to right: Point-cloud of the scene built from a trajectory; All objects detected by ConceptGraphs pipeline after assigning a category-name based on CLIP; Semantic visualization of the apartment with color-coded class categories per furniture

1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673

```
ROOM_NAME to this object. The input will be a JSON with fields "QUERY_OBJECT_NAME" and
"CLOSEST_OBJECTS". Your output should also be a JSON consisting of key "ROOM_LABEL".
You should only output the JSON and nothing else.

You should only assign one of the following labels:
1. bedroom
2. living room
3. kitchen
4. dining room
5. hallway
6. bathroom
7. unknown: only when none of the above strings describe the object<|eot_id|>
```

A.7 LEARNED LOW-LEVEL ROBOT SKILLS

We provide the implementation details for training the learned low-level robot skills: [Explore, Navigate, OpenFurniture, CloseFurniture, PickObject, PlaceObject]. These skills are based on two types of basic skills: Navigate skill, which outputs robot base control velocity command given robot depth and sensor observations, and Manipulation skill, which outputs joint angles and base control velocity commands to reach the target location given robot depth and sensor observations. We then use these two skills to get the above robot skills.

The LLM receives a binary signal of successful or unsuccessful execution, as well as a simple explanation of the failure. For example, if navigation fails because the robot was not able to reach the target in the maximum allowed number of steps, then the LLM receives an ‘observation’ - “Unsuccessful execution - skill timed out on the way to target”. There are some mistakes such as the robot picking up the wrong object, the observation says “Successful execution!” (since the robot is holding an object). The LLM needs to take into account the new scene graph to recover from these mistakes, however, it is unable to do so.

Most salient failure cases per skill are as below:

- Navigation : Not able to reach the target in allowed steps (timing out)
- Pick : Picking the wrong object, being too far away from target object, unable to pick because of learned skill failure (object out of reach of arm, or requiring complex motion planning to reach)
- Place: Target place position too far, skill timing out, learned skill failure (target place location out of reach).

A.7.1 NAVIGATE SKILL

We follow the learned low-level robot skill from Puig et al. (2024) to get the skill. We briefly describe the details for completeness. The goal of Navigate skill is to navigate to the target object given the target object location. The observation space includes (1) an arm depth camera (224 × 171 with hFOV of 55), (2) the relative pose of the target object location in 2-dim polar coordinate system. The action space includes (1) the linear and (2) angular robot base velocities with the range of −10 and +10 m/s. The reward function is to encourage the robot to move forward while facing the target location with a correct orientation. An additional navigation success reward is given if the robot can navigate close enough to the target object, and a collision penalty is given if the robot collides with obstacles in the scene. Moreover, a slack reward is given to let the robot navigate to the target in as few steps as possible. Finally, the skill is trained with DD-PPO distributing training.

Below is the list of skills based on navigate skill.

- Navigate: As described above.
- Explore: It is a composed skill that involves calling Navigate skills sequentially given a sequence of navigation waypoints.

A.7.2 MANIPULATION SKILL

We follow the work from Puig et al. (2024) to train the skill. We briefly describe the details for completeness. The goal of Manipulation skill is to drive the robot’s arm and base to reach the

target object’s location. The observation space includes (1) an arm depth camera (224×171 with hFOV of 55), (2) the relative pose of the target object in a 3-dim Cartesian coordinate system, (3) the 7-dim arm joint angles, (4) a binary holding detector, and (5) the relative pose of arm end-effector to the target resting location in a 3-dim Cartesian coordinate system. The action space includes (1) the linear and (2) angular base velocities with the range of -10 and $+10$ m/s, (3) the delta arm joint angles applied to the arm with the range of -5×10^{-2} and $+5 \times 10^{-2}$ (7-dim), and (4) a binary command to grasp or release the object. The reward function is to encourage the arm to move toward the object. In addition, a success reward is given if the robot interacts with the right target object. Moreover, a slack reward is given to let the robot complete the task in as few steps as possible. Finally, the skill is trained with DD-PPO distributing training.

Below is the list of skills based on `manipulate` skill.

- `OpenFurniture`: Given the drawer handle location, `manipulate` skill drives the arm and the base to the target. The articulated furniture is opened if the gripper location is close enough to the handle location.
- `CloseFurniture`: Given the drawer handle location, `manipulate` skill drives the arm and the base to the target. The articulated furniture is closed if the gripper location is close enough to the handle location.
- `PickObject`: Given the target object location, `manipulate` skill drives the arm and the base to the target. The object is snapped to the gripper if the gripper location is close enough to the target object location.
- `PlaceObject`: Given the target place location, `manipulate` skill drives the arm and the base to the target. The object is desnapped from the gripper to the target place location if the gripper location is close enough to the target place location.

A.8 IMPLEMENTATION DETAILS FOR REACT AGENTS

For all experiments, LLM inference is performed on two Nvidia A100 GPUs using the `gpt-fast` inference engine [PyTorch \(2023\)](#). Inference on Llama-3.1-70B (using tensor parallelism over two A100s), resulted in an average generation speed of 11.43 tokens/s. Each planning step required an average of 52 tokens resulting in a latency of 4.55 seconds per planning step. The average wall time to complete an entire episode (planning steps for both agents and simulation time) was 36.0 minutes. The finetuned model based on Llama-3.1-8B required an average of 0.53s per planning step. For those experiments, simulation time and human agent inference time remained unchanged, giving a final wall time of 25.3 minutes per episode.

All decentralized baselines had a maximum timeout of 50 replanning calls, while centralized baselines had a maximum timeout of 100 replanning calls (to account for the fact that one planner would need to plan for both agents). Additionally all baselines had a maximum timeout of 20000 simulation steps.

A.8.1 SKILL API LIBRARY

Below is a list of the skills available across all baselines. Agents acting in the robot role do not have access to state-modifying actions (`Clean`, `Fill`, `Pour`, `PowerOff`, `PowerOn`). The ReAct agents considered in the main paper do not have access to perception tools (`FindAgentActionTool`, `FindObjectTool`, `FindReceptacleTool`, `FindRoomTool`). We additionally study ReAct agents that query the environment via those tools, which we name ReAct-Tools.

- **Clean** : Used for cleaning an object.
- **Close** : Used for closing an articulated entity.
- **Explore** : Search a specific room by visiting various receptacles or furnitures in that room.
- **Fill** : Used for filling an object.
- **Navigate** : Used for navigating to an entity.
- **Open** : Used for opening an articulated entity.
- **Pick** : Used for picking up an object. The agent cannot hold more than one object at a time.

- 1728 • **Place** : Used for placing an object on a target location. This requires the name of
1729 the object to be placed, the name of the furniture where it should be placed, spatial rela-
1730 tion (“on” or “within”) describing the relation between the object and furniture. The ob-
1731 ject to be placed must already be held by the agent (i.e. picked previously). Addition-
1732 ally, you can request to place the object near another object. For that you can optionally
1733 provide a spatial constraints (“next_to”) and the name of the reference object. To place
1734 next to an object, the reference object must already be on the target furniture. API tem-
1735 plate: Place[<object_to_be_placed>, <spatial_relation>, <furniture
1736 to be placed on>, <spatial_constraint>, <reference_object>]. spa-
1737 tial_constraint and reference_object should be set to “None” when necessary.
- 1738 • **Pour** : Used for pouring from one container to another. This skill will pour into the specified
1739 container from whichever container is currently held by the agent.
- 1740 • **PowerOff** : Used for turning off a powered object
- 1741 • **PowerOn** : Used for turning on a powered object
- 1742 • **Rearrange** : Used for moving an object from its current location to the target location. This
1743 requires the name of the object to be rearranged, the name of the furniture where it should be
1744 placed, spatial relation (“on” or “within”) describing the relation between the object and furniture.
1745 This skill will automatically pick the specified object and move to the target furniture and attempt
1746 to place it. Additionally, you can request to place the object near another object. For that you
1747 can optionally provide a spatial constraints (“next_to”) and the name of the reference object. To
1748 place next to an object, the reference object must already be on the target furniture. API tem-
1749 plate: Rearrange[<object_to_be_placed>, <spatial_relation>, <furniture
1750 to be placed on>, <spatial_constraint>, <reference_object>].
1751 spatial_constraint and reference_object should be set to “None” when necessary.
- 1752 • **Wait** : Used to make agent stay idle for some time
- 1753 • **FindPartnerAgentActionTool** : This tool will return a summary of the other agent’s actions.
- 1754 • **FindObjectTool** : Used to find the exact name/names of the object/objects of interest. An LLM
1755 will be used to distill relevant objects from the user query. Example (FindObjectTool[toys on the
1756 floor] or FindObjectTool[apples])
- 1757 • **FindReceptacleTool** : Used to know the exact name of a receptacle. An LLM will be used to
1758 distill relevant receptacles from the user query. Example (FindReceptacleTool[a kitchen counter])
- 1759 • **FindRoomTool** : Used to know the exact name of a room in the house. An LLM will be used
1760 to distill relevant rooms from the user query. Example (FindRoomTool[a room which might have
1761 something to eat])
- 1762 • **Done** : Used to indicate that the agent has finished the task.

1764 A.8.2 CONSTRAINED GENERATION

1766 We follow the procedure described in [Geng et al. \(2023\)](#), constraining token sampling to only select
1767 tokens that consistent with at least one accepting string in the specified grammar. For each call to the
1768 LLM we build a grammar which will only accept valid tool calls on observed entities. Below is the
1769 base grammar used tool calls for all experiments. For experiments utilizing a summary of the world
1770 representation (i.e. ReAct, Finetuned see Section 4.1) the perception tools (FindObjectTool,
1771 FindReceptacleTool, etc.) are omitted. The rules for object, furniture, and room are
1772 set dynamically based on the agent’s current world graph. This ensures skills are called only for
1773 entities that the agent knows exist.

1774 Tool Call Grammar

```
1775 root ::= Navigate | Pick | Place | Open | Close | Rearrange | PowerOn | PowerOff | Clean
1776 | Fill | Pour | Explore | Wait | FindReceptacleTool | FindObjectTool |
1777 FindAgentActionTool | FindRoomTool | Done
1778 Navigate ::= "Navigate[" nav_target "]"
1779 Pick ::= "Pick[" object "]"
1780 Place ::= "Place[" object ", " WS spatial_relation ", " WS furniture ", " WS
1781 ((spatial_constraint ", " WS obj_or_furniture )| ("none" | "None") WS ", " WS ("none" |
1782 "None")) "]"
1783 Open ::= "Open[" furniture "]"
```

```

1782 Close ::= "Close[" furniture "]"
1783 Rearrange ::= "Rearrange[" object "," WS spatial_relation "," WS furniture "," WS
1784 ((spatial_constraint "," WS obj_or_furniture )| (("none" | "None") WS "," WS ("none" |
1785 "None"))) "]"
1785 PowerOn ::= "PowerOn[" obj_or_furniture "]"
1786 PowerOff ::= "PowerOff[" obj_or_furniture "]"
1787 Clean ::= "Clean[" obj_or_furniture "]"
1788 Fill ::= "Fill[" object "]"
1789 Pour ::= "Pour[" object "]"
1790 Explore ::= "Explore[" room "]"
1791 Wait ::= "Wait[" "]"
1792 FindReceptacleTool ::= "FindReceptacleTool[" free_text "]"
1793 FindObjectTool ::= "FindObjectTool[" free_text "]"
1794 FindAgentActionTool ::= "FindAgentActionTool[" "]"
1795 FindRoomTool ::= "FindRoomTool[" free_text "]"
1796 Done ::= "Done[]"
1797 nav_target ::= (furniture | room | object)
1798 object ::= "object_1" | "object_2" | ...
1799 obj_or_furniture ::= (furniture | object)
1800 furniture ::= "furniture_1" | "furniture_2" | ...
1801 room ::= "room_1" | "room_2" | ...
1802 spatial_constraint ::= "next_to"
1803 spatial_relation ::= "on" | "within"
1804 free_text ::= [ "'.:;!a-zA-Z_0-9)*
1805 WS ::= [ ]*

```

1801 A.8.3 RETRIEVAL-AUGMENTED GENERATION FOR REACT AGENTS

1803 Retrieval-Augmented Generation (RAG) is the method of optimizing LLM text generation by querying
 1804 an external database. However, there are several challenges to applying RAG in our setup. First, when
 1805 implementing RAG, it is necessary to provide an external database from which LLMs can retrieve
 1806 information. However, it is unclear how to effectively generate such a dataset and determine which
 1807 content is most beneficial for solving the task. Second, once LLMs retrieve the information, it is
 1808 unknown where to ingest this information into the generation process. To solve the above challenges,
 1809 we develop an approach inspired by the recent literature that uses an LLM to generate its training
 1810 dataset to iteratively optimize performance (Pang et al., 2024; Madaan et al., 2024). As shown in
 1811 Fig. 12, we first construct the dataset by collecting the successful traces generated by LLMs for
 1812 solving the training tasks. Then, during test time, we select the most relevant trace by comparing the
 1813 sentence similarity between a test instruction and the ones in the dataset. The selected trace is passed
 1814 back to the LLM’s prompt as a successful example trace. This represents a refinement process that
 1815 uses its own past success experience to increase the chance of solving downstream tasks. Figure 12
 1816 illustrates the high-level idea.

1817 Specifically, we use the **PARTNR** train dataset to generate ReAct traces (where the human agent is
 1818 ReAct-Tools and the robot agent is ReAct). In total, we generate 925 traces that successfully solve
 1819 the tasks to form the RAG dataset. During evaluation time, we use sentence similarity computed by
 1820 all-mpnet-base-v2 Reimers & Gurevych (2019) to select the most similar instruction to the
 1821 instruction at hand in the dataset, followed by adding the trace into the ReAct prompt.

1822 A.9 FINETUNING LLM AGENTS

1823
 1824 In this section, we describe how we finetune an LLM to build our Finetuned baseline. We detail the
 1825 process to generate the data for finetuning our model and the training details below.

1827 A.9.1 DATA GENERATION FOR FINETUNING

1828
 1829 We train the model using successful traces from the ReAct baseline, which obtains the best decentral-
 1830 ized results. In particular, we run this baseline together with a ReAct-Tools human on the training
 1831 set, and keep the episodes with 1.0 success rate. If an episode reaches a 100% success rate some
 1832 time during the task, but fails at the end, we keep the actions up until the success step and replace
 1833 the last action with a Done[] action to finish the episode. This process results in 1,226 episodes. We
 1834 then split each episode into the sequence of robot actions, and filter out those that resulted in failure.
 1835 Our training set is constructed by building a prompt for each of the successful actions, as shown in
 Sec. A.15. This process results in 15,889 training samples. Note that each action prompt contains the

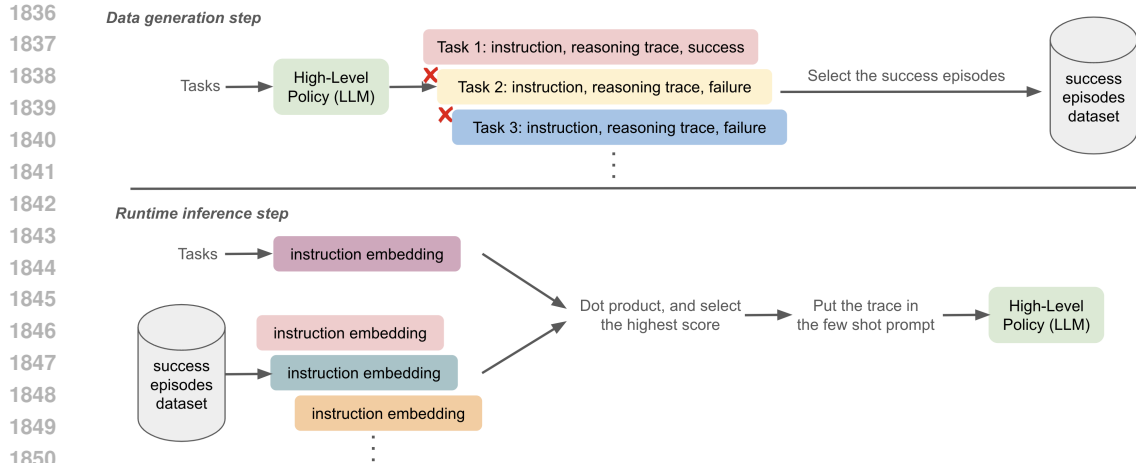


Figure 12: **Overview of ReAct-RAG.** RAG consists of two steps: (1) We first use a training dataset to obtain traces from LLMs and log traces with task instructions. (2) During runtime, we retrieve the traces by doing dot product on the instructions and obtaining the trace with the highest score, and finally put the trace in the prompt. We ensure that the dataset used for the first step is different from the one in the second step.

current state of the environment and the previous robot and human actions, but filters out the thought produced by the human or robot.

We also explored using the Heuristic-Expert baseline as a data source for finetuning, but we did not observe improvements in the resulting model. Given that this baseline plans using the ground truth evaluation function, we hypothesized it would help distill the natural language task into the right predicates. We followed the same process described above, obtaining 1250 episodes and 13939 training samples. We trained a model with ReAct, Heuristic-Expert and both data sources. We show the evaluation results for each model in Table 9, with the model trained with only ReAct data performing the best.

| Data source | Sim Steps ↓ | Success Rate ↑ | Percent Complete ↑ | Planning Cycles ↓ |
|--------------------------|-----------------|----------------|--------------------|-------------------|
| Heuristic-Expert | 3477.82 ± 78.19 | 0.63 ± 0.02 | 0.79 ± 0.01 | 15.50 ± 0.24 |
| ReAct | 3228.96 ± 75.14 | 0.70 ± 0.01 | 0.84 ± 0.01 | 12.85 ± 0.24 |
| ReAct + Heuristic-Expert | 3552.96 ± 61.95 | 0.69 ± 0.01 | 0.83 ± 0.01 | 14.47 ± 0.30 |

Table 9: **Performance of Finetuned model when using different data sources for finetuning.** We measure performance using simulation steps required to finish the episode, success rate and percent complete on the tasks, and the average number of planning cycles performed by the planner. Mean and standard error are reported over the validation set. Heuristic centralized expert has access to the task evaluation function.

A.9.2 IMPLEMENTATION DETAILS

We train the model to predict, for every example, the action taken by the agent, which corresponds to the text after the `<|reserved_special_token_0>` token.

We use a low rank adapter [Hu et al. \(2021\)](#) with $r = 132$, $\alpha = 128$, dropout=0.01, on top of the value and query projection layers W^V , W^Q . We train all models on 4 A100 GPUs, with a batch size of 2 per GPU. The models are trained for 40,000 steps, which takes around 24 hours.

A.10 ADDITIONAL RESULTS

To supplement the results in Table 2, we have included Table 10, containing results from additional baselines on the validation set, and Table 11 containing results on the **PARTNR** test set. ReAct rows

1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943

| Method | Controllability | Skills | Observability | Sim Steps ↓ | Success Rate ↑ | Completion Rate ↑ | Planning Cycles ↓ |
|------------------|-----------------|---------|---------------|-------------------|----------------|-------------------|-------------------|
| Heuristic-Expert | Centralized | Oracle | Full | 1260.88 ± 26.97 | 0.84 ± 0.01 | 0.94 ± 0.01 | N/A |
| ReAct | Centralized | Oracle | Full | 1347.43 ± 33.80 | 0.74 ± 0.01 | 0.88 ± 0.01 | 17.49 ± 0.34 |
| ReAct | Decentralized | Oracle | Full | 1915.63 ± 56.84 | 0.74 ± 0.01 | 0.86 ± 0.01 | 14.20 ± 0.34 |
| ReAct | Centralized | Oracle | Partial | 2298.13 ± 61.39 | 0.74 ± 0.01 | 0.85 ± 0.01 | 20.73 ± 0.51 |
| ReAct | Decentralized | Oracle | Partial | 3295.20 ± 76.27 | 0.73 ± 0.01 | 0.86 ± 0.01 | 15.24 ± 0.31 |
| ReAct-Tools | Decentralized | Oracle | Partial | 3622.52 ± 79.09 | 0.71 ± 0.01 | 0.85 ± 0.01 | 21.41 ± 0.34 |
| ReAct + RAG | Decentralized | Oracle | Partial | 3467.47 ± 82.39 | 0.71 ± 0.01 | 0.84 ± 0.01 | 14.75 ± 0.31 |
| Finetuned | Decentralized | Oracle | Partial | 3228.96 ± 75.14 | 0.70 ± 0.01 | 0.84 ± 0.01 | 12.85 ± 0.24 |
| ReAct | Decentralized | Learned | Partial | 6494.88 ± 181.52 | 0.57 ± 0.02 | 0.76 ± 0.01 | 22.72 ± 0.58 |
| ReAct | Decentralized | Learned | ConceptGraph | 12274.27 ± 212.65 | 0.25 ± 0.01 | 0.53 ± 0.01 | 26.74 ± 0.45 |
| ReAct-Single | Single Agent | Oracle | Partial | 2519.02 ± 57.48 | 0.73 ± 0.01 | 0.85 ± 0.01 | 18.68 ± 0.33 |
| ReAct-Single | Single Agent | Oracle | Full | 1590.20 ± 42.73 | 0.73 ± 0.01 | 0.85 ± 0.01 | 18.60 ± 0.38 |
| ReAct-8B | Decentralized | Oracle | Partial | 3699.45 ± 87.40 | 0.64 ± 0.02 | 0.80 ± 0.01 | 23.15 ± 0.47 |

Table 10: **Baseline results on PARTNR validation set.** We measure performance using simulation steps required to finish the episode, success rate and completion rate on the tasks, and the average number of planning cycles performed by the planner. Mean and standard error are reported over the validation set. Heuristic centralized expert has access to the task evaluation function. Collaboration enables higher task completion and success as compared to single agent task execution (shown in gray), at the expense of more simulation steps highlighting the coordination “burden”.

| Method | Controllability | Skills | Observability | Sim Steps ↓ | Success Rate ↑ | Completion Rate ↑ | Planning Cycles ↓ |
|------------------|-----------------|---------|---------------|------------------|----------------|-------------------|-------------------|
| Heuristic-Expert | Centralized | Oracle | Full | 1184.74 ± 22.88 | 0.69 ± 0.02 | 0.89 ± 0.01 | N/A |
| ReAct | Centralized | Oracle | Full | 1348.71 ± 53.34 | 0.67 ± 0.02 | 0.86 ± 0.01 | 21.14 ± 0.59 |
| ReAct | Centralized | Oracle | Partial | 2590.82 ± 90.71 | 0.56 ± 0.02 | 0.80 ± 0.01 | 25.57 ± 0.64 |
| ReAct | Decentralized | Oracle | Partial | 3353.33 ± 70.03 | 0.63 ± 0.02 | 0.84 ± 0.01 | 17.38 ± 0.33 |
| ReAct-Tools | Decentralized | Oracle | Partial | 3810.15 ± 86.52 | 0.61 ± 0.02 | 0.83 ± 0.01 | 25.79 ± 0.41 |
| ReAct + RAG | Decentralized | Oracle | Partial | 3489.18 ± 79.54 | 0.62 ± 0.02 | 0.83 ± 0.01 | 17.55 ± 0.38 |
| Finetuned | Decentralized | Oracle | Partial | 3460.60 ± 78.33 | 0.51 ± 0.02 | 0.78 ± 0.01 | 14.73 ± 0.25 |
| ReAct | Decentralized | Learned | Partial | 5905.88 ± 162.35 | 0.50 ± 0.02 | 0.76 ± 0.01 | 24.30 ± 0.60 |
| ReAct-Single | Single Agent | Oracle | Partial | 2632.30 ± 60.04 | 0.68 ± 0.01 | 0.85 ± 0.01 | 21.28 ± 0.37 |
| ReAct-Single | Single Agent | Oracle | Full | 1559.73 ± 36.02 | 0.73 ± 0.01 | 0.88 ± 0.01 | 21.06 ± 0.38 |
| ReAct-8B | Decentralized | Oracle | Partial | 4100.21 ± 98.97 | 0.51 ± 0.02 | 0.77 ± 0.01 | 27.65 ± 0.52 |

Table 11: **Baseline results on PARTNR test set.** We measure performance using simulation steps required to finish the episode, success rate and completion rate on the tasks, and the average number of planning cycles performed by the planner. Mean and standard error are reported over the test set. Heuristic centralized expert has access to the task evaluation function.

use the same summary prompting format as the baselines in 2. `ReAct-Tools` requires the agents to use perception tools to observe the environment instead. `ReAct-8B` uses `Llama-3.1-8B-Instruct` as the robot model for an equal capacity comparison with `Finetuned`.

We also present two additional baselines that use VLMs to extract visual information. We design two new baselines that combine the `ReAct` approach with a VLM to leverage vision during planning, as detailed below:

In the first baseline, we measure whether visual information can be combined with our graph representation to improve planning. For this, we still provide a privileged graph representation, but also add a caption of the agent’s current egocentric observation, generated using `Llama-3.2-11B-Vision-Instruct`. This provides strictly more information to the planner, as it uses both the visual information and accurate world graph. On the validation set, this baseline achieves a success rate of 0.72 (within the error margin of the baseline that does not use VLM, Table 2e), showing that adding the VLM output did not improve performance.

In the second baseline, we measure whether visual information alone can help perform better planning, when no graph is provided. For this, we use `Llama-3.2-11B-Vision-Instruct` to describe the visual observations seen by the agent so far. The planner uses these descriptions to generate an action. On the validation set, this baseline obtains a success rate of 0.69, lower than the baseline that uses a graph representation, Table 2 - row (e). Studying how to best leverage visual inputs to improve planning in `PARTNR` is an important direction that we leave for future work.

A.11 ANALYSIS OF COLLABORATIVE BEHAVIOR AND EFFICIENCY OF LLM AGENTS

Since task state success and percent complete metrics look at overall team performance for the human and robot agents in our tasks, we also evaluate metrics that allow us to look at different aspects of collaborative behavior of the agents. We measure percentage of sub-tasks done by the robot (task offloading), ratio of unnecessary rearrangements over total successful rearrangements done by both agents (extraneous effort), and number of exploration steps needed before first object is picked (exploration efficiency) to analyze agent behaviors (Table 12). Our main findings are below:

| Method | Sim Steps ↓ | Task Offloading↑ | Extraneous Effort↓ | Exploration Efficiency↓ |
|---------------|----------------|------------------|--------------------|-------------------------|
| Decentralized | 3295.20 ±76.27 | 0.596 ±0.01 | 0.21 ±0.01 | 994.88 ±24.890 |
| Centralized | 2298.13 ±61.39 | 0.49±0.01 | 0.04 ±0.004 | 684.06 ±21.71 |
| Single agent | 2519.02 ±57.48 | - | 0.047 ±0.01 | 1121.65 ±31.256 |

Table 12: **Analysis of collaboration characteristics for LLM agents.** Average and standard errors for task offloading, extraneous effort, and exploration efficiency are reported over the successful episodes from the validation set for LLM agents using `ReAct` approach in partially observable setting.

Agents are able to find objects faster when collaborating as compared to solo, but only when they successfully co-ordinate. The exploration efficiency increases i.e., agents are able to find task-relevant objects in fewer steps, in centralized and decentralized settings. By computing the average number of exploration steps taken before the first object is picked up for a task, we find that single agents require on average 127 steps more to locate objects compared to multi-agent. However, in centralized setting, where the co-ordination between agents is better owing to a single LLM co-ordinating the actions of both agents, shows higher gains in such efficiency as compared to decentralized settings. The challenge LLMs face in coordinating exploration in multi-agent settings also negatively impacts human-LLM team performance in our HITL experiments when paired with humans (Table 3).

Poor co-ordination also leads to wasted effort and more steps to complete the tasks than solo execution. Despite multiple agents working together, agents take longer to complete the tasks in decentralized settings as compared to solo execution owing to poor co-ordination. Poor co-ordination is further highlighted by extraneous effort, which increases by 300% in decentralized settings as compared to solo execution. The agents often repeat parts of the task – unsure of whether the other agent really executed that part, and sometimes even undo successfully completed tasks.

1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051

| Method | Success Rate per Task-type | | | |
|---------------|----------------------------|-----------|-----------|---------------|
| | Constraint-free | Spatial | Temporal | Object states |
| Decentralized | 0.82 ±0.02 | 0.82±0.02 | 0.60±0.03 | 0.66±0.03 |
| Centralized | 0.84 ±0.02 | 0.85±0.02 | 0.59±0.03 | 0.66±0.03 |
| Single agent | 0.85±0.02 | 0.81±0.02 | 0.58±0.03 | 0.68±0.03 |

Table 13: **Task performance per task type.** Average and standard errors of task success rate for episodes from the validation set categorized by task type. Performance is shown for LLM agents that use ReAct approach in partially observable setting.

The robot is able to offload more than half of the tasks from the human partner. The human-robot team takes longer to complete the task in decentralized setting, however, the robot offloads 60% tasks from the human partner, reducing their load of task execution. This highlights the potential of robots assisting humans more effectively as LLMs continue to advance in reasoning, coordination, and planning capabilities.

LLMs struggle to reason about temporal constraints and agent capabilities while planning PARTNR tasks. Constraint-free and spatial tasks in **PARTNR** require the LLMs to reason about only the final configuration and states of objects. Instead, the temporal tasks in **PARTNR** require tracking states of one or more objects over the entire episode, making them challenging (Table 13). Likewise, heterogeneous tasks necessitate reasoning about task distribution conditioned on each agent’s capabilities, which make them challenging.

A.12 HUMAN-IN-THE-LOOP EVALUATION FOR **PARTNR** TASKS

A.12.1 HITL INTERFACE AND WEB HOSTING

We adapt the existing human-in-the-loop (HITL) infrastructure from Habitat 3.0 (Puig et al., 2024) to support a more robust server-client architecture, with the server component hosted on AWS. Habitat3.0 HITL includes the ability to extend functionality to resource-constrained environments such as web browsers and VR devices, making the platform versatile for different user needs and experimental setups. Figure 13 shows our HITL system running on a web browser. Detailed interface is shown in Figure 14. Our adaptation to AWS hosting is crucial for handling multiple clients simultaneously, especially non-experts without access to powerful machines or large Habitat datasets. The server-client architecture not only enhances scalability but also ensures flexibility, allowing the system to accommodate a variety of operating systems and hardware platforms. Furthermore, the system includes a matchmaking service that enables pairing participants for multi-user sessions. When a participant requests a task, they are redirected to a "lobby screen" where they are instructed to wait until the next participant arrives.

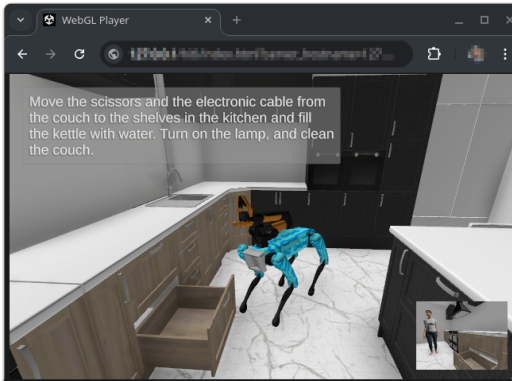


Figure 13: **HITL on Web-browser.** Our HITL system can be deployed on web browsers enabling large-scale collection.

A.12.2 PARTICIPANT RECRUITMENT AND QUALITY CONTROL

The study was performed through a 3rd party company specializing in large-scale annotations. The participants were recruited and compensated for their time by this company. The participants were English speakers, 18 years or older from the United States. For training them, we created a project and task overview video and guidelines. The participants were instructed to complete the tasks correctly and efficiently by themselves or with a partner. The participants also went through a tutorial where they performed some tasks to get acquainted with the interface before performing the main

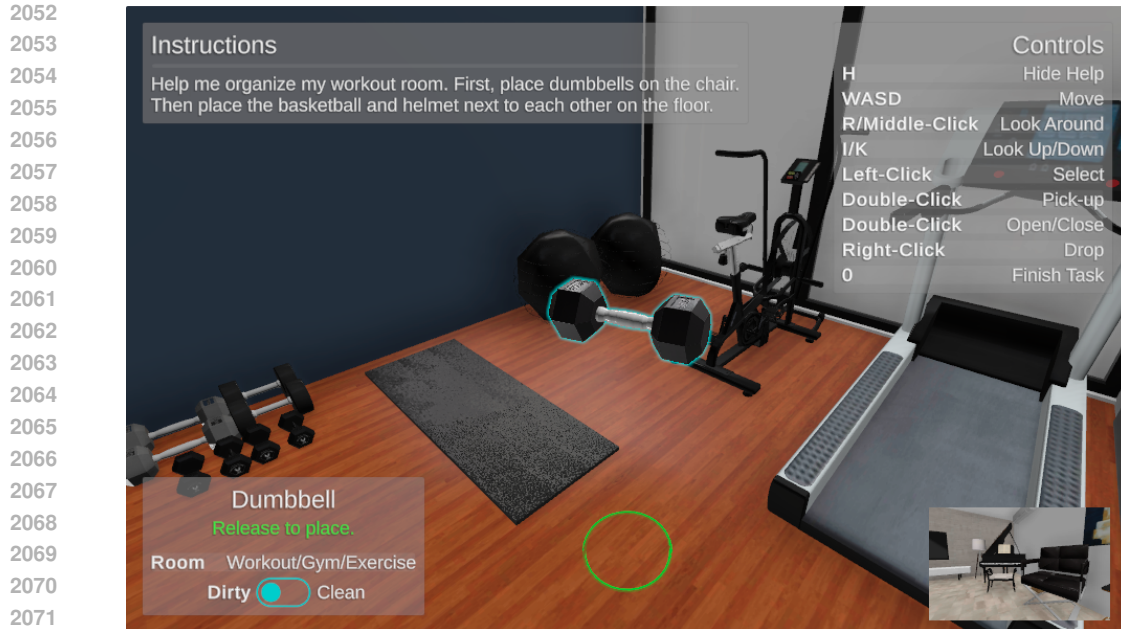


Figure 14: **HITL Interface**. Participants control human and robot agents using keyboard/mouse controls to complete the **PARTNR** tasks. Each participant has access to their partner’s viewpoint and thereby current actions via a small viewport on the bottom right.

tasks. Each task took on average 3-5 minutes to complete. We recruited 129 non-expert participants in total.

Filtering data:

Each task was completed up to 3 times, in all settings, until deemed successful through task evaluation (conducted online as the participants complete the tasks). With the Failure explanation output from the evaluation function, users were also given a natural language feedback at the end of an episode, describing what went wrong in an episode, if the task was not successful. For example, if actions were completed in the wrong order, or a spatial constraint like next-to was not respected appropriately. Users can use this information to update their actions in the next episode, improving their overall performance over time. By giving 4 tutorial episodes before the start of each experiment, and also a tutorial episode in the same house, we ensure that the users are deeply familiar with the tool and tasks before starting the actual experiment.

Once each task was completed successfully at least once or retried 3 times, we assimilate data by selecting one of the tries per episode: we select the successful try for successfully completed episodes, and the highest percent complete try for unsuccessful episodes. This collection of 1000 episodes for test and val is used for collecting performance statistics described in the next section.

A.12.3 HITL EXPERIMENTAL DETAILS

Task Evaluation and Data Collection:

Using the enhanced server-client setup, we conduct comprehensive evaluations of 1000 tasks from the validation set, and 1000 tasks from the test set. These evaluations are designed to capture data in both single-user and multi-user scenarios. In the single-user setting, participants individually control a human agent within the simulator using traditional keyboard and mouse inputs, completing tasks without any external assistance. Conversely, the multi-user setting involves collaborative efforts where two participants work together, each controlling either a human or a robot agent. This collaborative approach is specifically designed to study the dynamics between multiple users and to evaluate whether such collaborations lead to more efficient task completion compared to single-user efforts in the **PARTNR** tasks.

Human-AI Collaboration Experiment:

| Method | Success Rate \uparrow | Percent Complete \uparrow | Sim Steps \downarrow | Task Offloading \uparrow | Exploration Efficiency \downarrow | Extraneous Effort \downarrow |
|---------------------------|-------------------------|-----------------------------|------------------------|----------------------------|-------------------------------------|--------------------------------|
| Validation 1,000 episodes | | | | | | |
| Single-user | 0.93 \pm 0.01 | 0.96 \pm 0.00 | 3046.99 \pm 80.79 | N/A | 2459.22 \pm 26.75 | 0.09 \pm 0.01 |
| Multi-user | 0.93 \pm 0.01 | 0.96 \pm 0.00 | 2369.55 \pm 49.33 | 0.59 \pm 0.01 | 1762.47 \pm 13.99 | 0.15 \pm 0.01 |
| Human-ReAct | 0.91 \pm 0.01 | 0.96 \pm 0.02 | 4267.71 \pm 83.40 | 0.16 \pm 0.01 | 2624.39 \pm 26.05 | 0.12 \pm 0.01 |
| Human-Finetuned | 0.92 \pm 0.01 | 0.96 \pm 0.00 | 3443.33 \pm 61.46 | 0.26 \pm 0.01 | 2164.94 \pm 21.31 | 0.13 \pm 0.01 |
| Test 1,000 episodes | | | | | | |
| Single-user | 0.89 \pm 0.01 | 0.95 \pm 0.00 | 3937.87 \pm 110.53 | N/A | 2737.44 \pm 25.27 | 0.13 \pm 0.00 |
| Multi-user | 0.85 \pm 0.01 | 0.95 \pm 0.00 | 2667.86 \pm 58.07 | 0.60 \pm 0.01 | 1889.56 \pm 14.9=69 | 0.20 \pm 0.00 |
| Human-ReAct | 0.87 \pm 0.01 | 0.95 \pm 0.00 | 4080.10 \pm 72.24 | 0.12 \pm 0.00 | 2449.50 \pm 19.98 | 0.18 \pm 0.01 |
| Human-Finetuned | 0.87 \pm 0.01 | 0.95 \pm 0.00 | 3403.03 \pm 62.08 | 0.26 \pm 0.01 | 2162.20 \pm 19.40 | 0.20 \pm 0.01 |

Table 14: **Human-in-the-Loop evaluation.** We evaluate the performance of 2-person human teams and human-LLM teams, comparing them to solo human performance on **PARTNR** tasks using metrics described in Section 4.1. The human-LLM teams with SoTA LLMs is *slower* than solo human.

In addition to human-only interactions, we conduct experiments where a human collaborates with a robot controlled by a Language Model (LLM), specifically using the ReAct and Finetuned models as described in Section 4.1. The primary goal of these experiments is to evaluate the effectiveness of LLM-controlled agents in real-time collaboration with non-expert humans who have not previously interacted with these AI models. We track and compare the success rates (SR) and the percentage of tasks completed (PC) across various settings including single-user, multi-user, and human-AI collaborations. The results of these experiments are systematically documented and analyzed in Table 14, providing insights into the collaborative capabilities of human-AI pairs.

To enable this setting, we host LLMs on AWS nodes, and query them intermittently based on robot observations and actions. The HITL server now queries two clients - a human and a LLM. The human client sends commands to control the human agent, and the LLM client uses the hosted LLM to control the robot. Different baselines need different numbers of GPUs to keep the inference time reasonable. For hosting and 70B models, as used by the ReAct baselines, we use 4 A100 GPUs per model. For hosting a smaller 8B model used by the Finetuned baselines, we use 1 A100 GPU. This makes deploying smaller Finetuned models much scalable than larger 70B models.

Efficiency and Task Offloading Metrics:

To further understand the efficiency of task completion across different experimental setups, we measure several key performance metrics beyond success rate and percent complete. These include the number of steps taken to complete tasks and the exploration efficiency, which is assessed by the number of steps participants take to pick the first object. Additionally, we evaluate the extraneous effort by noting actions that do not contribute directly to task completion. Another critical metric we analyze is the ratio of work completed by the robot, referred to as task offloading. Ideally, in a well-coordinated human-AI team, the task offloading ratio should approach 0.5, indicating an efficient division of labor between the human and the robot.

A.12.4 HITL ANALYSIS

Humans are significantly better than LLMs at PARTNR tasks. Both single and multi-human settings achieve a success rate of 0.93 on **PARTNR** validation tasks, while ReAct without any privileged information only achieves 0.30 (Table 2, row (i)). This indicates a huge gap in LLM planning performance. We observe a slightly lower human performance on the test set (0.89), also in line with a lower LLM performance in Table 11 on this dataset. This indicates that the tasks in our test set are more challenging than the validation set for both humans and LLMs alike, potentially due to human annotations aimed at making them more complex and diverse.

Finetuned LLMs perform better than ReAct when coordinating with real humans. When deployed with real humans-in-the-loop on the validation set, the finetuned model is faster than ReAct at task completion (3275 steps with finetuned versus 4484 with ReAct on the validation set). It is also able to offload more tasks from humans than ReAct (26% with finetuned as compared to 16% with ReAct). This reflects that smaller models with faster inference can improve human experience in real-world deployment. This result is also reflected in the test set where finetuned model outperforms

2160 ReAct. Interestingly, the automated eval performance of finetuned is worse than ReAct on the test set,
 2161 but the HITL performance is better, indicating that faster inference is more critical than task success
 2162 when working with real humans.

2163 **LLMs struggle at coordination, hampering human performance.** Despite the Finetuned being
 2164 faster than ReAct when collaborating with humans, both approaches are *slower* than the human
 2165 doing the task alone. In contrast, two humans working together complete the task faster than a single
 2166 human (2369 steps vs. 3046 with multi- and single-user respectively). This result is in line with
 2167 the automated evaluation we observed in Table 1, where multi-agent LLMs are also *slower* than a
 2168 single-agent LLM. This result further reinforces that LLMs suffer at coordination; while humans are
 2169 able to coordinate and divide tasks between each other, decentralized LLMs are unable to do so. We
 2170 observe the same effect in the test set, further reinforcing this finding.

2171 **LLMs are able to offload tasks from humans.** Despite the aforementioned increase in the number
 2172 of steps for task completion, robots guided by the finetuned model successfully offload 26% of tasks
 2173 from humans. This indicates that LLMs can still offer assistance when collaborating with real human
 2174 partners. Nonetheless, there remains significant potential for improvement.

2175 **LLM’s inefficiency to explore reduces the team performance when paired with humans.** In
 2176 multi-user condition, the two humans start in different parts of a house, and explore efficiently to
 2177 locate task-relevant objects more quickly than a single user – as evidenced by the reduced number of
 2178 steps before first pick (1762 steps with multi-user vs. 2459 steps for a single user). However, this
 2179 efficiency is reduced when humans are paired with LLMs (2120 steps with finetuned and 2791 steps
 2180 with ReAct), indicating that LLMs struggle to coordinate at *both* task completion and exploration.

2182 A.13 PROMPTS FOR BENCHMARK TASK AND EVALUATION

2184 A.13.1 TASK GENERATION PROMPTS

2185 Prompts are similar for the different task types, with the primary difference being the in-context
 2186 examples.

2188 **Constraint-free task generation prompt**

2189 You are a system that generates tasks for robots to perform with humans.
 2190 Do not be verbose. Answer the question with no added qualifications or caveats. Just
 2191 directly provide the answer in JSON.

2192 You will be given a description of a house with objects and furniture and your task is
 2193 to provide 5 instructions for tasks that a robot and a human could be doing together in
 2194 that house, using the objects and furniture.

2195 For each task, provide the initial state of objects in the house, the instruction that
 2196 should be performed, and final state of the objects after the instruction is performed.
 2197 The initial and final state will contain a list of dictionaries, each with an object
 2198 type, the number of objects of that type, their location on a furniture or floor, and
 the region of the house where they are in e.g., bedroom.

2199 Follow the next principles:

- 2200 1. The instruction should be given as if the human doing the task wanted the robot to
 2201 perform part of it. In some cases the task will be done together, in other cases, the
 human and robot will be doing different tasks.
- 2202 2. The initial and final state should contain objects of different types, and sometimes
 2203 multiple objects of a type.
- 2204 3. Some of the instructions should be semantically rich, in particular they should refer
 to classes or groups of objects.
- 2205 4. The instructions shouldn't be detailed and explain all the steps, but the high-level.
- 2206 5. The robot can only rearrange objects and open containers, the human can do more tasks
 e.g., turn on lamp, clean plates, fill up pitcher.
- 2207 6. The instruction should contain a clear goal and at least two steps associated with
 2208 the goal.
- 2209 7. Ensure that instructions are diverse from each other. Some instructions should
 contain spatial specifiers such as "next to", "left", "right", "beside", "near",
 2210 "front", "side". While some other instructions should contain temporal order, which can
 be specified using words such as "after", "then", "before" etc. For instance: "Fill up
 2211 the kettle and then turn it on. After that, bring two cups to the dinning table."

2214 8. The instructions should contain diverse actions such as "turn on/off", "fill",
 2215 "clear", "set" etc. and object states such as "clean", "dirty", "open", "close" etc.
 2216 while referring to objects.

2217

2218 You will be given all the pieces of furniture in the house.
 2219 You will also be given all the different types of objects that you can use. You can
 2220 specify multiple instances of an object type.
 2221 Make sure your instruction includes the object types and furniture present in the list
 2222 below.

2223 The house has the following rooms, each with the following furniture:
 2224 {house_furniture}

2225

2226 You can use the following objects:
 2227 {objects_list}

2228

2229 Here is an example with two instructions:

2230 JSON_OUTPUT: [
 2231 {{
 2232 "initial state": [
 2233 {{
 2234 "object_type": "lamp",
 2235 "how_many": 1,
 2236 "furniture_name": "table_10",
 2237 "region": "living_room_1"
 2238 }},
 2239 {{
 2240 "object_type": "book",
 2241 "how_many": 3,
 2242 "furniture_name": "table_11",
 2243 "region": "living_room_1"
 2244 }},
 2245 {{
 2246 "object_type": "toy_vehicle",
 2247 "how_many": 2,
 2248 "furniture_name": "floor",
 2249 "region": "living_room_1"
 2250 }},
 2251 {{
 2252 "object_type": "toy_cactus",
 2253 "how_many": 1,
 2254 "furniture_name": "table_1",
 2255 "region": "living_room_1"
 2256 }}
 2257],
 2258 "final state": [
 2259 {{
 2260 "object_type": "lamp",
 2261 "how_many": 1,
 2262 "furniture_name": "table_10",
 2263 "region": "living_room_1"
 2264 }},
 2265 {{
 2266 "object_type": "book",
 2267 "how_many": 3,
 2268 "furniture_name": "shelves_2",
 2269 "region": "living_room_1"
 2270 }},
 2271 {{
 2272 "object_type": "toy_vehicle",
 2273 "how_many": 2,
 2274 "furniture_name": "bed_2",
 2275 "region": "bedroom_1"
 2276 }},
 2277 {{
 2278 "object_type": "toy_cactus",
 2279 "how_many": 1,
 2280 "furniture_name": "bed_2",
 2281 "region": "bedroom_1"
 2282 }}
 2283],
 2284 "instruction": "We need to clean up the living room. Move all toys and books to the
 2285 shelf in the living room.",
 2286 }],
 2287]

```

2268     "reason": "The task involves moving multiple objects to the shelf in the living
2269     room."
2270   }},
2271   {
2272     "initial state": [
2273       {
2274         "object_type": "plate",
2275         "how_many": 3,
2276         "furniture_name": "cabinet_2",
2277         "region": "kitchen_1"
2278       },
2279       {
2280         "object_type": "glass",
2281         "how_many": 2,
2282         "furniture_name": "counter_1",
2283         "region": "kitchen_1"
2284       },
2285       {
2286         "object_type": "fork",
2287         "how_many": 5,
2288         "furniture_name": "cabinet_5",
2289         "region": "kitchen_1"
2290       }
2291     ],
2292     "final state": [
2293       {
2294         "object_type": "plate",
2295         "how_many": 2,
2296         "furniture_name": "table_8",
2297         "region": "living_room_1"
2298       },
2299       {
2300         "object_type": "glass",
2301         "how_many": 2,
2302         "furniture_name": "table_8",
2303         "region": "living_room_1"
2304       },
2305       {
2306         "object_type": "fork",
2307         "how_many": 2,
2308         "furniture_name": "table_8",
2309         "region": "living_room_1"
2310       }
2311     ],
2312     "instruction": "Help me set up a table for dinner in the livingroom for 2 people.
2313     Place 2 plates and 2 glasses on the table.",
2314     "reason": "The task includes semantically rich descriptions (set up a table)."
2315   }
2316 ]
2317
2318 Generate a JSON list with {k} instructions like the examples above.
2319 Your output should only be:
2320 JSON_OUTPUT: result_list
2321 where result_list should be a JSON list with the instructions.
2322 Let's think through this carefully, step by step.

```

Spatial task generation prompt

2310 You are a system that generates tasks for robots to perform with humans.
2311 Do not be verbose. Answer the question with no added qualifications or caveats. Just
2312 directly provide the answer in JSON.

2313 You will be given a description of a house with objects and furniture and your task is
2314 to provide 5 instructions for tasks that a robot and a human could be doing together in
2315 that house, using the objects and furniture.

2316 For each task, provide the initial state of objects in the house, the instruction that
2317 should be performed, and final state of the objects after the instruction is performed.
2318 The initial and final state will contain a list of dictionaries, each with an object
2319 type, the number of objects of that type, their location on a furniture or floor, and
2320 the region of the house where they are in e.g., bedroom.

2321 Follow the next principles:

2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375

1. The instruction should be given as if the human doing the task wanted the robot to perform part of it. In some cases the task will be done together, in other cases, the human and robot will be doing different tasks.
2. The initial and final state should contain objects of different types, and sometimes multiple objects of a type.
3. Some of the instructions should be semantically rich, in particular they should refer to classes or groups of objects.
4. The instructions shouldn't be detailed and explain all the steps, but the high-level.
5. The robot can only rearrange objects and open containers, the human can do more tasks e.g., turn on lamp, clean plates, fill up pitcher.
6. The instruction should contain a clear goal and at least two steps associated with the goal.
7. Ensure that instructions are diverse from each other.
8. All the instructions should contain at least one of the spatial specifiers from this list: "next to", "left", "right", "beside", "near", "front", "side".
9. The instructions should contain diverse actions such as "turn on/off", "fill", "clear", "set" etc. and object states such as "clean", "dirty", "open", "close" etc.

You will be given all the pieces of furniture in the house.
You will also be given all the different types of objects that you can use. You can specify multiple instances of an object type.
Make sure your instruction includes the object types and furniture present in the list below.

The house has the following rooms, each with the following furniture:
{house_furniture}

You can use the following objects:
{objects_list}

Here is an example with two instructions:

```
JSON_OUTPUT: [
  {
    "initial state": [
      {
        "object_type": "vase",
        "how_many": 1,
        "furniture_name": "table_10",
        "region": "living_room_1"
      },
      {
        "object_type": "stuffed_toy",
        "how_many": 2,
        "furniture_name": "floor",
        "region": "bedroom_2"
      },
      {
        "object_type": "candle",
        "how_many": 1,
        "furniture_name": "chest_of_drawers_2",
        "region": "bedroom_1"
      }
    ],
    "final state": [
      {
        "object_type": "vase",
        "how_many": 1,
        "furniture_name": "shelves_2",
        "region": "living_room_1"
      },
      {
        "object_type": "stuffed_toy",
        "how_many": 2,
        "furniture_name": "shelves_2",
        "region": "living_room_1"
      },
      {
        "object_type": "candle",
        "how_many": 1,
        "furniture_name": "shelves_2",
        "region": "living_room_1"
      }
    ]
  },
]
```

```

2376     "instruction": "Let's decorate! Put the vase on the shelf. Then, set a candle and a
2377     stuffed_toy on each side of the vase.",
2378     "reason": "The task includes spatial constraint specified by 'side'."
2379   }},
2380   [
2381     {
2382       "initial state": [
2383         {
2384           "object_type": "plate",
2385           "how_many": 3,
2386           "furniture_name": "cabinet_2",
2387           "region": "kitchen_1"
2388         },
2389         {
2390           "object_type": "glass",
2391           "how_many": 2,
2392           "furniture_name": "counter_1",
2393           "region": "kitchen_1"
2394         },
2395         {
2396           "object_type": "fork",
2397           "how_many": 5,
2398           "furniture_name": "cabinet_5",
2399           "region": "kitchen_1"
2400         }
2401       ],
2402       "final state": [
2403         {
2404           "object_type": "plate",
2405           "how_many": 2,
2406           "furniture_name": "table_8",
2407           "region": "living_room_1"
2408         },
2409         {
2410           "object_type": "glass",
2411           "how_many": 2,
2412           "furniture_name": "table_8",
2413           "region": "living_room_1"
2414         },
2415         {
2416           "object_type": "fork",
2417           "how_many": 2,
2418           "furniture_name": "table_8",
2419           "region": "living_room_1"
2420         }
2421       ],
2422       "instruction": "Help me set up a table for dinner in the livingroom for 2 people.
2423       Place 2 plates and 2 glasses on the table. There should be a fork next to each
2424       plate",
2425       "reason": "The task includes semantically rich descriptions (set up a table) and
2426       spatial constraints specified by the word 'next'."
2427     }
2428   ]
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449

```

Generate a JSON list with {k} instructions like the examples above.
Your output should only be:
JSON_OUTPUT: result_list
where result_list should be a JSON list with the instructions.
Let's think through this carefully, step by step.

Temporal task generation prompt

You are a system that generates tasks for robots to perform with humans.
Do not be verbose. Answer the question with no added qualifications or caveats. Just directly provide the answer in JSON.

You will be given a description of a house with objects and furniture and your task is to provide 5 instructions for tasks that a robot and a human could be doing together in that house, using the objects and furniture.

For each task, provide the initial state of objects in the house, the instruction that should be performed, and final state of the objects after the instruction is performed. The initial and final state will contain a list of dictionaries, each with an object type, the number of objects of that type, their location on a furniture or floor, and the region of the house where they are in e.g., bedroom.

2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483

Follow the next principles:

1. The instruction should be given as if the human doing the task wanted the robot to perform part of it. In some cases the task will be done together, in other cases, the human and robot will be doing different tasks.
2. The initial and final state should contain objects of different types, and sometimes multiple objects of a type.
3. Some of the instructions should be semantically rich, in particular they should refer to classes or groups of objects.
4. The instructions shouldn't be detailed and explain all the steps, but the high-level.
5. The robot can only rearrange objects and open containers, the human can do more tasks e.g., turn on lamp, clean plates, fill up pitcher.
6. The instruction should contain a clear goal and at least two steps associated with the goal.
7. Ensure that instructions are diverse from each other.
8. All the instructions should contain temporal order, specified using one of the words from this list: "after", "then", "before", "finally", "first". For instance: "Fill up the kettle and then turn it on. After that, bring two cups to the dining table."
9. The instructions should contain diverse actions such as "turn on/off", "fill", "clear", "set" etc. and object states such as "clean", "dirty", "open", "close" etc.

You will be given all the pieces of furniture in the house.
You will also be given all the different types of objects that you can use. You can specify multiple instances of an object type.
Make sure your instruction includes the object types and furniture present in the list below.

The house has the following rooms, each with the following furniture:
{house_furniture}

You can use the following objects:
{objects_list}

Here is an example with two instructions:

```
JSON_OUTPUT: [
  {
    "initial state": [
      {
        "object_type": "lamp",
        "how_many": 1,
        "furniture_name": "table_10",
        "region": "living_room_1"
      },
      {
        "object_type": "book",
        "how_many": 3,
        "furniture_name": "table_11",
        "region": "living_room_1"
      },
      {
        "object_type": "toy_vehicle",
        "how_many": 2,
        "furniture_name": "floor",
        "region": "living_room_1"
      },
      {
        "object_type": "toy_cactus",
        "how_many": 1,
        "furniture_name": "table_1",
        "region": "living_room_1"
      }
    ],
    "final state": [
      {
        "object_type": "lamp",
        "how_many": 1,
        "furniture_name": "table_10",
        "region": "living_room_1"
      },
      {
        "object_type": "book",
        "how_many": 3,
        "furniture_name": "shelves_2",
```

```

2484         "region": "living_room_1"
2485     }},
2486     {{
2487         "object_type": "toy_vehicle",
2488         "how_many": 2,
2489         "furniture_name": "bed_2",
2490         "region": "bedroom_1"
2491     }},
2492     {{
2493         "object_type": "toy_cactus",
2494         "how_many": 1,
2495         "furniture_name": "bed_2",
2496         "region": "bedroom_1"
2497     }},
2498     ],
2499     "instruction": "We need to clean up the living room. Move all toys to the bedroom
2500     and the books to the shelf. After that, turn on the lamp in the living room.",
2501     "reason": "The task includes temporal constraints specified by 'after'."
2502 },
2503 {{
2504     "initial state": [
2505         {{
2506             "object_type": "plate",
2507             "how_many": 3,
2508             "furniture_name": "cabinet_2",
2509             "region": "kitchen_1"
2510         }},
2511         {{
2512             "object_type": "glass",
2513             "how_many": 2,
2514             "furniture_name": "counter_1",
2515             "region": "kitchen_1"
2516         }},
2517         {{
2518             "object_type": "fork",
2519             "how_many": 5,
2520             "furniture_name": "cabinet_5",
2521             "region": "kitchen_1"
2522         }},
2523     ],
2524     "final state": [
2525         {{
2526             "object_type": "plate",
2527             "how_many": 2,
2528             "furniture_name": "table_8",
2529             "region": "living_room_1"
2530         }},
2531         {{
2532             "object_type": "glass",
2533             "how_many": 2,
2534             "furniture_name": "table_8",
2535             "region": "living_room_1"
2536         }},
2537         {{
2538             "object_type": "fork",
2539             "how_many": 2,
2540             "furniture_name": "table_8",
2541             "region": "living_room_1"
2542         }},
2543     ],
2544     "instruction": "Help me set up a table for dinner in the livingroom for 2 people.
2545     First place 2 plates on the table. Then, place glasses and forks next to each
2546     plate.",
2547     "reason": "The task includes semantically rich descriptions (set up a table) and
2548     temporal constraints specified by the words 'first' and 'then'."
2549 }}
2550 ]
2551
2552 Generate a JSON list with {k} instructions like the examples above.
2553 Your output should only be:
2554 JSON_OUTPUT: result_list
2555 where result_list should be a JSON list with the instructions.
2556 Let's think through this carefully, step by step.
2557

```

2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591

Large-scale task generation prompt

You are a system that generates tasks for robots to perform with humans.
Do not be verbose. Answer the question with no added qualifications or caveats. Directly provide the answer in JSON.

You will be given a description of a house with objects and furniture. You will also be given a sample task. Your job is to modify this sample task so that it is applicable to this house.

Here is an example:

```
Task: [
  {
    "instruction": "Move the kettle and the jug from the living room to the kitchen and fill the kettle with water, then turn on the kettle. Finally clean the living room table.",
    "initial_state": [
      {
        "number": 1,
        "object_classes": [
          "kettle"
        ],
        "furniture_names": [
          "table_0"
        ],
        "allowed_regions": [
          "living_room_0"
        ]
      },
      {
        "number": 1,
        "object_classes": [
          "jug"
        ],
        "furniture_names": [
          "table_1"
        ],
        "allowed_regions": [
          "living_room_0"
        ]
      }
    ]
  }
]
```

```
JSON_OUTPUT: [
  {
    "instruction": "Move the jug, kettle, teapot, and cup from the dining table to the kitchen and fill all with water. Turn on the lamp, and clean the dining table.",
    "initial_state": [
      {
        "number": 1,
        "object_classes": [
          "jug"
        ],
        "furniture_names": [
          "table_1"
        ],
        "allowed_regions": [
          "living_room_0"
        ]
      },
      {
        "number": 1,
        "object_classes": [
          "kettle"
        ],
        "furniture_names": [
          "table_1"
        ],
        "allowed_regions": [
          "living_room_0"
        ]
      }
    ]
  },
  {
    "number": 1,
```



```

2592     "object_classes": [
2593         "teapot"
2594     ],
2595     "furniture_names": [
2596         "table_1"
2597     ],
2598     "allowed_regions": [
2599         "living_room_0"
2600     ]
2601 },
2602 {{
2603     "number": 1,
2604     "object_classes": [
2605         "cup"
2606     ],
2607     "furniture_names": [
2608         "table_0"
2609     ],
2610     "allowed_regions": [
2611         "living_room_0"
2612     ]
2613 },
2614 {{
2615     "number": 1,
2616     "object_classes": [
2617         "lamp"
2618     ],
2619     "furniture_names": [
2620         "table_1"
2621     ],
2622     "allowed_regions": [
2623         "living_room_0"
2624     ]
2625 }}
2626 ]
2627 }}
2628 ]
2629
2630 The house has the following rooms, each with the following furniture:
2631 {house_furniture}
2632
2633 You can use the following objects:
2634 {objects_list}
2635
2636 Here is the task:
2637 Task: [
2638 {task}
2639 ]
2640
2641 Modify this task to generate a JSON list of tasks, using the rooms and furniture from
2642 this house.
2643
2644 Just change the objects and furniture.
2645
2646 Make sure initial and final locations of objects are different.
2647
2648 Include actions such as turn on/off, fill and clean.
2649
2650 Make tasks multi-step, consisting of more than one object and action.
2651
2652 Your output should only be:
2653 JSON_OUTPUT: result_list
2654 where result_list should be a JSON list with the tasks.
2655

```

A.14 EVALUATION GENERATION PROMPTS

Here we share the LLM prompts used for proposition generation, temporal constraint prediction, and argument constraint prediction. In each, the task to accomplish is described in the system prompt and between 6-13 few-shot examples follow.

Evaluation Generation: Propositions

Source: system

2646 You will be given an instruction describing a household task and a description of the
2647 initial state of the house. You will define a list of python functions that must be
2648 satisfied for the task to be marked complete.

2649 You can call the following functions:

```
2650 - is_on_top(object_names: str, furniture_name: str) # any object in object_names is on
2651 top of a furniture
2652 - is_inside(object_names: str, furniture_name: str) # any object in object_names is
2653 inside of a furniture
2654 - is_in_room(object_names: str, room_name: str) # any object in object_names is in
2655 a room
2656 - is_on_floor(object_names: str) # any object in object_names is on
2657 the floor
2658 - is_next_to(objects_a: str, objects_b: str) # any object in objects_a is next
2659 to any object in objects_b
2660 - is_clean(object_names: str) # any object in object_names is
2661 clean
2662 - is_dirty(object_names: str) # any object in object_names is
2663 dirty
2664 - is_filled(object_names: str) # any object in object_names is
2665 filled, like with a liquid
2666 - is_empty(object_names: str) # any object in object_names is
2667 empty
2668 - is_powered_on(object_names: str) # any object in object_names is
2669 powered on
2670 - is_powered_off(object_names: str) # any object in object_names is
2671 powered off
```

2665 Objects in object_names can be expressed as it appears in the objects list
2666 ("stuffed_toy_1") or as an OR of object names ("stuffed_toy_1 or stuffed_toy_2").
2667 A furniture_name can be expressed as it appears in the furniture list (e.g. "table") or
2668 as it appears in the furniture-room relation ("table in living_room").

2669 Only use the functions listed above.
2670 Each function should test a single objects/furniture/room relation.
2671 If the instruction is ambiguous such that multiple objects could be used to satisfy a
2672 function (an OR relationship), then include all possible objects.
2673 Define as many functions as necessary.
2674 Write each function on its own line.
2675 It is essential to wrap each function in delimiters [FN] and [/FN].
2676 End your functions with the token [END].

2677 Let's see some examples. Suppose the initial state is:

```
2678 Objects:
2679 * pants_1
2680 * shirt_1
2681 * shirt_2
2682 * shirt_3
2683 Furniture:
2684 * washer_dryer
2685 * table
2686 Rooms:
2687 * laundryroom
2688 Object-Furniture-Room Relations:
2689 * pants_1 on table in laundryroom
2690 * shirt_1 on table in laundryroom
2691 * shirt_2 on floor in laundryroom
2692 Furniture-Room Relations:
2693 * washer_dryer in laundryroom
2694 * table in laundryroom
```

2688 Instruction "Put the pants on the table" means

```
2689 [FN] is_on_top("pants_1", "table in laundryroom") [/FN]
```

2690 Instruction "Put the pants in the washer" means

```
2691 [FN] is_inside("pants_1", "washer_dryer in laundryroom") [/FN]
```

2693 Instruction "Put a shirt in the washer" means

```
2694 [FN] is_inside("shirt_1 or shirt_2 or shirt_3", "washer_dryer in laundryroom") [/FN]
```

2696 Instruction "Put all the shirts in the washer" means

```
2697 [FN] is_inside("shirt_1", "washer_dryer in laundryroom") [/FN]
2698 [FN] is_inside("shirt_2", "washer_dryer in laundryroom") [/FN]
2699 [FN] is_inside("shirt_3", "washer_dryer in laundryroom") [/FN]
```

2700
2701 Instruction "Put a shirt on the floor next to the pants" means
2702 [FN] is_on_floor("shirt_1 or shirt_2 or shirt_3") [/FN]
2703 [FN] is_next_to("shirt_1 or shirt_2 or shirt_3", "pants_1") [/FN]
2704
2705 Instruction "Bring a shirt to the laundryroom" means
2706 [FN] is_in_room("shirt_1 or shirt_2 or shirt_3", "laundryroom") [/FN]
2707
2708 <step> Source: user
2709
2710 Objects:
2709 * casserole_0
2710 * squeezer_0
2711 Furniture:
2711 * cabinet
2712 * table
2713 Rooms:
2713 * bathroom
2714 * hallway
2714 * kitchen
2715 * living_room
2716 Object-Furniture-Room Relations:
2716 * casserole_0 on cabinet in kitchen
2717 * squeezer_0 on cabinet in kitchen
2718 Furniture-Room Relations:
2718 * table in bathroom
2719 * table in hallway
2720 * table in living_room
2721
2722 Instruction: "Get the casserole and squeezer from the kitchen and put them on the table
2722 in the living room."
2723
2724 <step> Source: assistant
2724
2725 [FN] is_on_top("casserole_0", "table in living_room") [/FN]
2725 [FN] is_on_top("squeezer_0", "table in living_room") [/FN]
2726 [END]
2727
2728 <step> Source: user
2728
2729 The initial state is:
2729 Objects:
2730 * toy_truck_1
2730 * stuffed_toy_1
2731 * stuffed_toy_2
2732 Furniture:
2732 * table
2733 * bed
2734 * cabinet
2735 Rooms:
2735 * bedroom
2736 * other_room
2736 * living_room
2737 Object-Furniture-Room Relations:
2737 * toy_truck_1 on bed in bedroom
2738 * stuffed_toy_1 on floor in bedroom
2739 * stuffed_toy_2 on floor in other_room
2740 Furniture-Room Relations:
2740 * table in living_room
2741
2742 Instruction: "Bring a toy from the bedroom and put it in the living room."
2743
2744 <step> Source: assistant
2744
2745 [FN] is_in_room("toy_truck_1 or stuffed_toy_1", "living_room") [/FN]
2745 [END]
2746
2747 <step> Source: user
2747
2748 The initial state is:
2748 Objects:
2749 * cushion_1
2749 * fork_1
2750 * plate_1
2751 Furniture:
2751 * bed
2752 * couch
2753

2754 * counter
2755 Rooms:
2756 * bedroom
2757 * living_room
2758 Object-Furniture-Room Relations:
2759 * cushion_1 on couch in living_room
2760 * fork_1 on couch in living_room
2761 * plate_1 on couch in living_room
2762 Furniture-Room Relations:
2763 * bed in bedroom
2764 * couch in living_room
2765 * counter in kitchen
2766
2767 Instruction: "Bring the dishes back to the kitchen counter. After that, put the cushion
2768 on the bed."
2769
2770 <step> Source: assistant
2771
2772 [FN] is_on_top("plate_1", "counter in kitchen") [/FN]
2773 [FN] is_on_top("fork_1", "counter in kitchen") [/FN]
2774 [FN] is_on_top("cushion_1", "bed in bedroom") [/FN]
2775 [END]
2776
2777 <step> Source: user
2778
2779 The initial state is:
2780 Objects:
2781 * candle_0
2782 * fork_0
2783 * toy_food_0
2784 Furniture:
2785 * counter
2786 * table
2787 Rooms:
2788 * bedroom
2789 * kitchen
2790 * living_room
2791 Object-Furniture-Room Relations:
2792 * candle_0 on table in living_room
2793 * fork_0 on counter in kitchen
2794 * toy_food_0 on counter in kitchen
2795 Furniture-Room Relations:
2796 * table in bedroom
2797 * counter in kitchen
2798 * table in living_room
2799
2800 Instruction: "Put the toy food on the table where the candle is, and place the fork on
2801 the table beside the toy food."
2802
2803 <step> Source: assistant
2804
2805 [FN] is_on_top("toy_food_0", "table in living_room") [/FN]
2806 [FN] is_on_top("fork_0", "table in living_room") [/FN]
2807 [FN] is_next_to("fork_0", "toy_food_0") [/FN]
2808 [END]
2809
2810 <step> Source: user
2811
2812 Objects:
2813 * toy_airplane_0
2814 * toy_airplane_1
2815 Furniture:
2816 * table
2817 Rooms:
2818 * hallway
2819 * living_room
2820 Object-Furniture-Room Relations:
2821 * toy_airplane_0 on table in hallway
2822 * toy_airplane_1 on table in hallway
2823 Furniture-Room Relations:
2824 * table in bathroom
2825 * table in hallway
2826 * table in living_room
2827
2828 Instruction: "Bring a toy airplane to the living room and put it in on the table."
2829
2830 <step> Source: assistant
2831
2832 [FN] is_on_top("toy_airplane_0 or toy_airplane_1", "table in living_room") [/FN]
2833

```

2808 [END]
2809
2810 <step> Source: user
2811
2812 Objects:
2813   * kettle_0 (empty, dirty)
2814 Furniture:
2815   * table
2816 Rooms:
2817   * living_room
2818 Object-Furniture-Room Relations:
2819   * kettle_0 on table in living_room
2820   * toy_airplane_0 on table in hallway
2821   * toy_airplane_1 on table in hallway
2822 Furniture-Room Relations:
2823   * table in living_room
2824
2825 Instruction: "First, make sure the kettle is clean. Then, fill the kettle with water and
2826 turn it on."
2827
2828 <step> Source: assistant
2829
2830 [FN] is_clean("kettle_0") [/FN]
2831 [FN] is_filled("kettle_0") [/FN]
2832 [FN] is_powered_on("kettle_0") [/FN]
2833 [FN] is_on_top("toy_airplane_0 or toy_airplane_1", "table in living_room") [/FN]
2834 [END]
2835
2836 {TEMPLATE_EXAMPLE}
2837
2838 <step> Source: user
2839
2840 The initial state is:
2841 {INIT_STATE}
2842
2843 Instruction: "{INSTRUCTION}"
2844
2845 <step> Source: assistant
2846 Destination: user
2847
2848 [FN]

```

Evaluation Generation: Temporal

```

2837 Source: system
2838
2839 You will be given an instruction describing a task to perform in a house and a set of
2840 propositions that define whether the task was done successfully. The task instruction
2841 may say that certain propositions should be completed before others ("then", "after",
2842 "finally"). Your job is to write python code that groups the propositions in the order
2843 in which they must be completed.
2844
2845 The propositions are well-defined python functions that return a boolean value.
2846
2847 You will be given a list of propositions where index i corresponds to the ith
2848 proposition. To solve the task, define the variable proposition_order, which groups
2849 propositions together that can be completed in any order. Each proposition group
2850 appearing in proposition_order must be satisfied before the group that comes after it.
2851 For example,
2852
2853 proposition_order = [
2854     [0, 1]
2855 ]
2856
2857 means that propositions 0 and 1 can be completed in any order. This example
2858
2859 proposition_order = [
2860     [0],
2861     [1]
2862 ]
2863
2864 means that the proposition 0 must be completed before proposition 1. This example
2865
2866 proposition_order = [
2867     [0, 1],
2868     [2]
2869 ]

```

2862
2863 means that propositions 0 and 1 can be completed in either order, but proposition 2 must
2864 be completed after.

2865 Start by assuming that propositions can be completed in any order. Order matters if the
2866 instruction includes time ordering words such as "then", "finally", or "after". In this
2867 case, propositions should be in multiple groups.

2868 Double check that the index for each proposition is included in proposition_order.

2869 <step> Source: user

2870 Instruction: "Bring an apple to the kitchen table, then bring an orange to the kitchen
2871 counter."

2872 propositions = [
2873 is_on_top(["apple_1"], ["table_4"]),
2874 is_on_top(["orange_1"], ["counter_0"])
2875]

2876 <step> Source: assistant

2877 proposition_order = [
2878 [0],
2879 [1]
2880]

2881 <step> Source: user

2882 Instruction: "Put the toy vehicle and the water bottle in the living room. Next, return
2883 the dish to the kitchen."

2884 propositions = [
2885 is_in_room(["toy_truck_1"], ["living_room_1"]),
2886 is_in_room(["cup_0"], ["living_room_1"]),
2887 is_in_room(["bowl_2"], ["kitchen"])
2888]

2889 <step> Source: assistant

2889 proposition_order = [
2890 [0, 1],
2891 [2]
2892]

2893 <step> Source: user

2894 Instruction: "Put an apple on the bench in the entryway. Also move the broom to the
2895 closet."

2896 propositions = [
2897 is_on_top(["apple_0", "apple_1"], ["bench_2"]),
2898 is_inside(["broom_0"], ["closet_0"])
2899]

2900 <step> Source: assistant

2900 proposition_order = [
2901 [0, 1]
2902]

2903 <step> Source: user

2904 Instruction: "Bring me the toy truck from the bedroom and put it in the living room.
2905 Then put two apples on the kitchen table."

2906 propositions = [
2907 is_in_room(["toy_truck_1"], ["living_room_1"]),
2908 is_on_top(["apple_1"], ["table_1"]),
2909 is_on_top(["apple_2"], ["table_1"])
2910]

2911 <step> Source: assistant

2912 proposition_order = [
2913 [0],
2914 [1, 2]
2915]

```

2916 <step> Source: user
2917
2918 Instruction: "Bring the dishes back to the kitchen counter. After that, put the cushions
2919 on the bed."
2920
2921 propositions = [
2922   is_on_top(["plate_1"], ["counter_1", "counter_2", "counter_3"]),
2923   is_on_top(["fork_1"], ["counter_1", "counter_2", "counter_3"]),
2924   is_on_top(["spoon_0"], ["counter_1", "counter_2", "counter_3"]),
2925   is_on_top(["cushion_0"], ["bed_1", "bed_2"]),
2926   is_on_top(["cushion_1"], ["bed_1", "bed_2"]),
2927 ]
2928
2929 <step> Source: assistant
2930
2931 proposition_order = [
2932   [0, 1, 2],
2933   [3, 4]
2934 ]
2935
2936 <step> Source: user
2937
2938 Instruction: "Bring the dishes back to the kitchen counter. Put the cushion on the bed.
2939 Then, move the cushions to the kitchen."
2940
2941 propositions = [
2942   is_on_top(["plate_1"], ["counter_1", "counter_2", "counter_3"]),
2943   is_on_top(["fork_1"], ["counter_1", "counter_2", "counter_3"]),
2944   is_on_top(["cushion_0"], ["bed_1", "bed_2"]),
2945   is_on_top(["cushion_1"], ["bed_1", "bed_2"]),
2946   is_in_room(["cushion_0"], ["bedroom_0"]),
2947   is_in_room(["cushion_1"], ["bedroom_0"]),
2948 ]
2949
2950 <step> Source: assistant
2951
2952 proposition_order = [
2953   [0, 1, 2, 3],
2954   [4, 5]
2955 ]
2956
2957 <step> Source: user
2958
2959 Instruction: "Move the clothes from the bedroom to the washer. After that, Put the
2960 cushion on the bed. Finally, put the book in the living room."
2961
2962 propositions = [
2963   is_on_top(["shirt_1"], ["washer_dryer_1"]),
2964   is_on_top(["shirt_2"], ["washer_dryer_1"]),
2965   is_on_top(["pants_1"], ["washer_dryer_1"]),
2966   is_on_top(["cushion_1"], ["bed_1", "bed_2"]),
2967   is_in_room(["book_1"], ["living_room_1"])
2968 ]
2969
2970 <step> Source: assistant
2971
2972 proposition_order = [
2973   [0, 1, 2],
2974   [3],
2975   [4]
2976 ]
2977
2978 <step> Source: user
2979
2980 Instruction: "First, move the spoon and kettle from the kitchen to the living room and
2981 place them next to each other. Then, place the toy food in the kitchen cabinet."
2982
2983 propositions = [
2984   is_on_top(['spoon_0'], ['table_1', 'table_2', 'table_3', 'table_4', 'table_5']),
2985   is_on_top(['kettle_0'], ['table_1', 'table_2', 'table_3', 'table_4', 'table_5']),
2986   is_next_to(['spoon_0'], ['kettle_0']),
2987   is_inside(['toy_food_0'], ['cabinet_0'])
2988 ]
2989
2990 <step> Source: assistant
2991
2992 proposition_order = [
2993   [0, 1, 2],
2994   [3]
2995 ]

```

```

2970 ]
2971
2972 <step> Source: user
2973
2974 Instruction: "First, move the phone stand from the bedroom to the living room and place
2975 it on the table next to the lamp. Then, move the file sorter from the living room to the
2976 bedroom and place it on the table next to the phone stand."
2977
2978 propositions = [
2979     is_on_top(['phone_stand_0'], ['table_1', 'table_2', 'table_3']),
2980     is_next_to(['phone_stand_0'], ['lamp_0']),
2981     is_on_top(['file_sorter_0'], ['table_6']),
2982     is_next_to(['file_sorter_0'], ['phone_stand_0'])
2983 ]
2984
2985 <step> Source: assistant
2986
2987 proposition_order = [
2988     [0, 1],
2989     [2, 3]
2990 ]
2991
2992 <step> Source: user
2993
2994 Instruction: "Help me move the candle and hand towel to the kitchen counter. Place them
2995 next to each other. Then, place
2996 the spatula and c-clamp on the bedside table next to each other."
2997
2998 propositions = [
2999     is_on_top(['candle_0'], ['counter_0']),
3000     is_on_top(['hand_towel_0'], ['counter_0']),
3001     is_next_to(['candle_0'], ['hand_towel_0']),
3002     is_on_top(['spatula_0'], ['table_6']),
3003     is_on_top(['c-clamp_0'], ['table_6']),
3004     is_next_to(['spatula_0'], ['c-clamp_0'])
3005 ]
3006
3007 <step> Source: assistant
3008
3009 proposition_order = [
3010     [0, 1, 2],
3011     [3, 4, 5]
3012 ]
3013
3014 <step> Source: user
3015
3016 Instruction: "First, move the dog bowl, then the placemat, and finally a plush toy from
3017 the living room to the bench in the hallway. Place them next to each other."
3018
3019 propositions = [
3020     is_on_top(['dog_bowl_0'], ['bench_0']),
3021     is_on_top(['placemat_0'], ['bench_0']),
3022     is_next_to(['placemat_0'], ['dog_bowl_0']),
3023     is_on_top(['plush_toy_0'], ['bench_0']),
3024     is_next_to(['plush_toy_0'], ['placemat_0'])
3025 ]
3026
3027 <step> Source: assistant
3028
3029 proposition_order = [
3030     [0],
3031     [1, 2],
3032     [3, 4]
3033 ]
3034
3035 <step> Source: user
3036
3037 Instruction: "Move the toaster and the bread from the pantry to the kitchen and turn the
3038 toaster on, then fill the kettle."
3039
3040 propositions = [
3041     is_in_room(['toaster_0'], ['kitchen_0']),
3042     is_in_room(['bread_0'], ['kitchen_0']),
3043     is_powered_on(['toaster_0']),
3044     is_filled(['kettle_0'])
3045 ]
3046
3047 <step> Source: assistant
3048
3049 
```



```

3024 proposition_order = [
3025     [0, 1, 2],
3026     [3]
3027 ]
3028 <step> Source: user
3029 Instruction: "{INSTRUCTION}"
3030 {PROPOSITIONS}
3031
3032 <step> Source: assistant
3033 Destination: user
3034 proposition_order = [
3035

```

Evaluation Generation: Argument Constraints

```

3038 Source: system
3039
3040 You will be given a task to perform in a house, and a set of propositions that define
3041 whether the task was done successfully. The task is performed by a human and robot. The
3042 task instruction may imply constraints such that certain groups of propositions should
3043 be satisfied by the same argument or unique arguments. Your job is to write python code
3044 that defines these constraints.
3045
3046 The propositions are well-defined python functions that return a boolean value.
3047
3048 You will be given a list of propositions where index i corresponds to the ith
3049 proposition. To solve the task, define the variable tie_constraints, which is a list of
3050 constraints which can be empty. The constraints you can use are:
3051
3052 SameArgConstraint(
3053     proposition_indices: List[int], # indices of propositions that this constraint
3054     applies to
3055     arg_index: List[int], # indices of arguments that should be matched on
3056 )
3057
3058 DifferentArgConstraint(
3059     proposition_indices: List[int], # indices of propositions that this constraint
3060     applies to
3061     arg_index: List[int], # indices of arguments that should be matched on
3062 )
3063
3064 Here are some examples:
3065
3066 SameArgConstraint([0, 1], [0, 0]) # means that propositions at index 0 and 1 must have
3067 a matching value in the first argument.
3068 DifferentArgConstraint([0, 1], [0, 0]) # means that propositions at index 0 and 1 must
3069 have different values in the first argument.
3070 SameArgConstraint([0, 1], [1, 1]) # means that propositions at index 0 and 1 must have
3071 a matching value in the second argument.
3072
3073 If no constraints apply to the given instruction, just write an empty list.
3074
3075 <step> Source: user
3076 Instruction: "Bring an apple and an orange to a table in the kitchen."
3077
3078 propositions = [
3079     is_on_top(["apple_1"], ["table_3", "table_4"]),
3080     is_on_top(["orange_1"], ["table_3", "table_4"])
3081 ]
3082
3083 <step> Source: assistant
3084
3085 tie_constraints = [
3086     SameArgConstraint([0, 1], [1, 1])
3087 ]
3088
3089 <step> Source: user
3090 Instruction: "Put the toy vehicle in the living room and return the dish to the kitchen."
3091
3092 propositions = [
3093     is_in_room(["toy_truck_1"], ["living_room_1"]),
3094     is_in_room(["bowl_2"], ["kitchen"])

```

```

3078 ]
3079
3080 <step> Source: assistant
3081
3082 tie_constraints = [
3083 ]
3084
3085 <step> Source: user
3086
3087 Instruction: "Place the book on the shelf in the bedroom. Place the picture frame next
3088 to it."
3089
3090 propositions = [
3091   is_on_top(["book_1"], ["shelves_0", "shelves_1"]),
3092   is_on_top(["picture_frame_0"], ["shelves_0", "shelves_1"]),
3093   is_next_to(["picture_frame_0"], ["book_1"])
3094 ]
3095
3096 <step> Source: assistant
3097
3098 tie_constraints = [
3099   SameArgConstraint([0, 1], [1, 1])
3100 ]
3101
3102 <step> Source: user
3103
3104 Instruction: "Place each candle on its own table in the living room."
3105
3106 propositions = [
3107   is_on_top(["candle_0"], ["table_0", "table_2", "table_6"]),
3108   is_on_top(["candle_1"], ["table_0", "table_2", "table_6"]),
3109   is_on_top(["candle_2"], ["table_0", "table_2", "table_6"])
3110 ]
3111
3112 <step> Source: assistant
3113
3114 tie_constraints = [
3115   DifferentArgConstraint([0, 1, 2], [1, 1])
3116 ]
3117
3118 <step> Source: user
3119
3120 Instruction: "Move the clothes from the bedroom to the washer. After that, Put the
3121 cushion on the bed. Finally, put the book in the living room."
3122
3123 propositions = [
3124   is_on_top(["shirt_1"], ["washer_dryer_1"]),
3125   is_on_top(["shirt_2"], ["washer_dryer_1"]),
3126   is_on_top(["pants_1"], ["washer_dryer_1"]),
3127   is_on_top(["cushion_1"], ["bed_1", "bed_2"]),
3128   is_in_room(["book_1"], ["living_room_1"])
3129 ]
3130
3131 <step> Source: assistant
3132
3133 tie_constraints = [
3134 ]
3135
3136 <step> Source: user
3137
3138 Instruction: "{INSTRUCTION}"
3139
3140 {PROPOSITIONS}
3141
3142 <step> Source: assistant
3143 Destination: user
3144
3145 tie_constraints = [

```

A.15 PROMPTS FOR PLANNING BASELINES

Following prompts were used for various planning baselines.

Decentralized Single/Multi Agent | ReAct

3132 <|start_header_id|>system<|end_header_id|>You are an agent that solves multi-agent
 3133 planning problems. The task assigned to you will be situated in a house and will
 3134 generally involve navigating to objects, picking and placing them on different
 3135 receptacles to achieve rearrangement. You strictly follow any format specifications and
 3136 pay attention to the previous actions taken in order to avoid repeating mistakes. Rooms
 do not need to be explored more than once.

3137 There will be another agent trying to solve the same task that you are at the same time.
 3138 You may find that that agent has picked up relevant objects or is in the process of
 3139 completing parts of the task. If that is the case you may want to move on to a different
 part of the task.

3140 Rooms do not need to be explored more than once.
 3141 This means if you have explored the living room and have not found the object, then you
 3142 should explore the kitchen, if a relevant object is still not found, you should explore
 the hallway etc...

3143 {agent_role_description}

3144 Many calls to the same action in a row are a sign that something has gone wrong and you
 3145 should try a different action.<|eot_id|>{optional_rag_examples}

3146 <|start_header_id|>user<|end_header_id|>Task: {task_description}

3147 {world_description}

3148 Possible Actions:
 3149 {tool_descriptions}

3150 What is the next action to make progress towards completing the task?
 3151 Return your response in the following format

3152 Thought: <reasoning for why you are taking the next action>
 3153 <next action call>
 3154 Assigned!

3155 Here is an example:
 3156 Thought: Since there are no objects found I should explore a room I have not explored yet
 3157 Explore[<room name>]
 3158 Assigned!
 3159 <|eot_id|><|start_header_id|>assistant<|end_header_id|>

Centralized I ReAct

3160 <|start_header_id|>system<|end_header_id|>You are a system that solves multi-agent
 3161 planning tasks. The task assigned to you will be situated in a house and will generally
 3162 involve navigating to objects, picking and placing them on different receptacles to
 3163 achieve rearrangement. There will be a robot agent (Agent 0) and a human agent (Agent 1)
 3164 available for solving the task. Your goal is to assign actions for both of these agents
 3165 and solve the task. You strictly follow any format specifications and pay attention to
 3166 the previous actions taken in order to avoid repeating mistakes.

3167 You should try and divide the task between the two agents for efficient task completion.
 3168 Note that the human agent can wash, clean, fill, pour and turn on/off devices along with
 3169 doing object rearrangement. However, the robot can only do object rearrangement i.e.,
 3170 navigating to objects, picking and placing them.

3171 In the beginning, you will be provided with the task description and information about
 3172 the rooms plus furniture in each room for the house. Object information may or may not
 3173 be available. Rooms only need to be explored if there is no information available about
 3174 task-relevant objects. Rooms do not need to be explored for identifying which furniture
 3175 to go to. Also, rooms do not need to be explored more than once. This means if one of
 3176 your agents has explored the living room and have not found the object, then you should
 explore the kitchen, if a relevant object is still not found, you should explore the
 hallway etc.

3177 Many calls to the same action in a row are a sign that something has gone wrong and you
 3178 should try a different action.

3179 You should try to complete the task in the least amount of actions possible. This means
 3180 if there are two objects to be moved you should have one agent navigate to each object
 3181 and then move them to the target location at the same time.

3182 If a previous navigation action is still in progress for an agent, you should reassign
 3183 that action to the agent till a successful execution is observed in the agent's
 3184 observations.

3186 You should continue to evaluate the task progress and decide the actions for both the
 3187 agents. Once both agents are done, you can output "Done[]" to indicate that the agents
 3188 have finished the task. Output your response about task completion in the following
 3189 format.

3190 Thought: <reasoning about why both agents have completed the entire task successfully>
 3191 Done[]

3192 DO NOT output "Done[]" unless you are confident that the whole task is successfully
 3193 completed. If one of the agent is done with its part of the task, while the other agent
 3194 is still executing, you can assign a "Wait[]" action to the agent who doesnt need to act
 3195 anymore. Please re-state the task description and verify it's completion before
 3196 outputting "Done[]".{eot_tag}{user_tag}Task: {input}

3197 {world_description}

3198 Possible actions for each agent:
 3199 {agent_descriptions}

3200 What is the next action for each agent to make progress towards completing the task?
 3201 Return your response in the following format

3202 Thought: <reasoning for why you are taking the next action>
 3203 Agent_0_Action: <next action call for robot agent>
 3204 Agent_1_Action: <next action call for human agent>
 3205 Assigned!

3206 Here is an example:
 3207 Thought: Since there are multiple task-relevant objects to be rearranged, I will ask
 3208 each agent to go to one of them
 3209 Agent_0_Action: Navigate[<obj name1>]
 3210 Agent_1_Action: Navigate[<obj name2>]
 3211 Assigned!
 3212 <|eot_id|><|start_header_id|>assistant<|end_header_id|>

3211 The agent role description would be one of the following, depending on if the agent played the role
 3212 of the human or robot.

3213 Agent Role Descriptions

3214 Human Description

3215 You are playing the role of the task giver. This means if the instruction says something
 3216 like "You should move the object and I will wash it", then the other agent should be
 3217 moving the object, and you should washing the it.

3218 Robot Description

3219 You are playing the role of the task receiver. This means if the instruction says
 3220 something like "You should move the object and I will wash it", then you should move the
 3221 object and the other agent should wash it

3222 Below are the tool descriptions. Perceptions tools are only included for the React-Tools agents.

3223 Tool Descriptions

- 3224 - Close: Used for closing an articulated entity. You must provide the name of the
 3225 furniture you want to close. Example (Close[chest_of_drawers_1])
- 3226 - Explore: Search a specific room by visiting various receptacles or furnitures in that
 3227 room. The input to the skill is the EXACT name of the room to be visited. Use the room
 3228 names provided in the house description. This tool exhaustivly explores the specified
 3229 room. Example (Explore[kitchen_1])
- 3230 - FindAgentActionTool: Should be used to find current and past state history of other
 3231 agent.
- 3232 - FindObjectTool: Used to find the exact name/names of the object/objects of interest.
 3233 If you want to find names of objects on specific receptacles or furnitures, please
 3234 include that in the query. Example (FindObjectTool[toys on the floor] or
 3235 FindObjectTool[apples])
- 3236 - FindReceptacleTool: Used to know the exact name of a receptacle. A receptacle is a
 3237 furniture or entity (like a chair, table, bed etc.) where you can place an object.
 3238 Example (FindReceptacleTool[a kitchen counter])
- 3239 - FindRoomTool: Used to know the exact name of a room in the house. A room is a region
 in the house where furniture is placed. Example (FindRoomTool[a room which might have
 something to eat])

3240 - Navigate: Used for navigating to an entity. You must provide the name of the entity
 3241 you want to navigate to. Example (Navigate[counter_22])
 3242 - Open: Used for opening an articulated entity. You must provide the name of the
 3243 furniture you want to open. Example (Open[chest_of_drawers_1])
 3244 - Pick: Used for picking up an object. You must provide the name of the object to be
 3245 picked. The agent cannot hold more than one object at a time. Example (Pick[cup_1])
 3246 - Place: Used for placing an object on a target location. You need to provide the name
 3247 of the object to be placed, the name of the furniture where it should be placed, spatial
 3248 relation ("on" or "within") describing the relation between the object and furniture.
 3249 The object to be placed must already be held by the agent (i.e. picked previously). In
 3250 addition to these, you can request to place the object near another object. For that you
 3251 can optionally provide a spatial constraints ("next_to") and the name of the reference
 3252 object. To place next to an object, the reference object must already be on the target
 3253 furniture. API template - Place[<object_to_be_placed>, <spatial_relation>, <furniture to
 3254 be placed on>, <spatial_constraint>, <reference_object>]. spatial_constraint and
 3255 reference_object should be set to "None" when necessary.
 3256 - Rearrange: Used for moving an object from its current location to the target location.
 3257 You need to provide the name of the object to be moved, the name of the furniture where
 3258 it should be moved, spatial relation ("on" or "within") describing the relation between
 3259 the object and furniture. This will automatically pick the specified object and move to
 3260 the target furniture and attempt to place it. In addition to these, you can request to
 3261 place the object near another object. For that you can optionally provide a spatial
 3262 constraints ("next_to") and the name of the reference object. To place next to an
 3263 object, the reference object must already be on the target furniture. API template
 3264 Rearrange[<object_to_be_moved>, <spatial_relation>, <furniture to be placed on>,
 3265 <spatial_constraint>, <reference_object>]. spatial_constraint and reference_object
 3266 should be set to "None" when necessary.
 3267 - Wait: Used to make agent stay idle for some time. Example (Wait[])
 3268 - Done: Used to indicate that the agent has finished the task. Example (Done[])

3261 For LLM agents simulating a human, additionally actions which modify the state of objects are
 3262 available to be called. For centralized baselines two lists of available actions are provided in the
 3263 agent description. One for the robot (without state-modifying actions) and one for the human (with
 3264 state-modifying actions).

Human Only Tool Descriptions

3266 - Clean: Used for cleaning an object. You need to provide the name of the object to
 3267 clean.
 3268 - Close: Used for closing an articulated entity. You must provide the name of the
 3269 furniture you want to close. Example (Close[chest_of_drawers_1])
 3270 - Fill: Used for filling an object. You need to provide the name of the object to fill.
 3271 - Pour: Used for pouring from one container to another. This skill will pour into the
 3272 specified container from whichever container is currently held by the agent.
 3273 - PowerOff: Used for turning off a powered object. You need to provide the name of the
 3274 object to be turned off.
 3275 - PowerOn: Used for turning on a powered object. You need to provide the name of the
 3276 object to be turned on.

3276 The world description contains a text description all rooms and their contained furniture, along with
 3277 currently observed objects. Below is an example for one scene:

World Description Example

3278 Furniture:
 3279 bedroom_1: floor_bedroom_1, chair_41, chair_42, bed_49, table_54, chest_of_drawers_72,
 3280 chest_of_drawers_73, chest_of_drawers_75, chest_of_drawers_87
 3281 closet_1: floor_closet_1, wardrobe_91
 3282 living_room_1: floor_living_room_1, chair_13, chair_14, chair_15, chair_16, chair_17,
 3283 chair_18, chair_19, chair_20, chair_21, chair_22, couch_26, couch_28, couch_29,
 3284 chair_30, stool_31, stool_32, table_38, table_39, table_48, table_50, stand_52,
 3285 counter_78
 3286 toilet_1: floor_toilet_1, toilet_43
 3287 bedroom_2: floor_bedroom_2, bed_23, chair_46, chair_47, table_53, chest_of_drawers_55,
 3288 chest_of_drawers_58, chest_of_drawers_59, chest_of_drawers_60, chest_of_drawers_61
 3289 bedroom_3: floor_bedroom_3, bed_37, chair_40, chest_of_drawers_74, table_77,
 3290 chest_of_drawers_79, chest_of_drawers_80, chest_of_drawers_89, wardrobe_92
 3291 bedroom_4: floor_bedroom_4, chair_27, bed_45, table_51, wardrobe_56, wardrobe_57,
 3292 chest_of_drawers_82, chest_of_drawers_83, table_84, table_86
 3293 kitchen_1: floor_kitchen_1, shelves_24, shelves_25, chair_33, chair_34, chair_35,
 chair_36, cabinet_62, cabinet_63, cabinet_64, cabinet_65, cabinet_66, counter_67,
 counter_68, counter_69, cabinet_70, cabinet_71, cabinet_76, cabinet_81, unknown_88,
 fridge_90

3294 other_room_1: floor_other_room_1
 3295 other_room_2: floor_other_room_2
 3296 bathroom_1: floor_bathroom_1, toilet_44
 3297 porch_1: floor_porch_1
 3298 entryway_1: floor_entryway_1, table_85
 3299 The following furnitures have a faucet: cabinet_70
 3300 Objects:
 3301 No objects found yet

3302 For baselines utilizing a summary of the scene, after each action, we insert a text description of
 3303 objects in the scene (as perceived by the agent so far).
 3304

3305 Scene Summary Example

3306
 3307 Thought: Since there are no objects found, I should explore a room I have not explored
 3308 yet. I will start with the living room, as it is one of the rooms mentioned in the task,
 3309 and it might contain the vase or other relevant objects.
 3310 Explore[living_room_1]
 3311 Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
 3312
 3313 Result: Successful execution!
 3314 Objects: toy_fruits_1: chest_of_drawers_49 in bedroom_4
 3315 laptop_cover_2: cabinet_59 in laundryroom_1
 3316 cup_3: cabinet_65 in kitchen_1
 3317 vase_0: cabinet_65 in kitchen_1<|eot_id|><|start_header_id|>assistant<|end_header_id|>
 3318
 3319 Thought: Now that we have found the locations of the objects, I should navigate to the
 3320 kitchen to pick up the vase, as it is one of the objects mentioned in the task, and it
 3321 is located in the kitchen.
 3322 Navigate[cabinet_65]
 3323 Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
 3324
 3325 Result: Successful execution!
 3326 Objects: toy_fruits_1: chest_of_drawers_49 in bedroom_4
 3327 laptop_cover_2: cabinet_59 in laundryroom_1
 3328 cup_3: cabinet_65 in kitchen_1
 3329 vase_0: cabinet_65 in kitchen_1<|eot_id|>

3324 For finetuning, we build the prompt by combining the task description, world description and the
 3325 previous actions for both agents.
 3326

3327 Finetuning Example

3328
 3329 Task: I finished reading my book. Please move the book to the bedroom table. You can
 3330 move the vase there too?
 3331
 3332 Current Environment:
 3333 {world_description}
 3334
 3335 Previous actions:
 3336 Agent_Action: Navigate[table_25]
 3337 Action Result: Successful execution!
 3338 Agent_Action: Navigate[vase_0]
 3339 Action Result: Successful execution!
 3340 Agent_Action: Pick[vase_0]
 3341 Action Result: Successful execution!
 3342 Agent_Action: Navigate[table_35]
 3343 Action Result: Successful execution!
 3344 Other_Agent_Action: Pick[book_1]
 3345 Agent_Action: Open[table_35]
 3346 Action Result: Successful execution!
 3347 Agent_Action: Navigate[table_35]
 3348 Action Result: Successful execution!
 3349 Agent_Action: Place[vase_0, on, table_35, none, none]
 3350 Action Result: Successful execution!
 3351 Agent_Action: Wait[]
 3352 Action Result: Successful Execution!
 3353 Other_Agent_Action: Open[table_35]
 3354 Other_Agent_Action: Place[book_1, on, table_35, none, none]
 3355
 3356 Next Agent_Action:<|reserved_special_token_0|>Done[]<end_act>

3348 A.16 FURTHER ANALYSIS OF FAILURES
3349

3350 We describe here in more detail some of the common failures of our proposed LLM-based planner.
3351 To better illustrate the failures, we also provide example traces for these.

3352 **Failing to maps objects to their corresponding room leads to unnecessary exploration:** Some-
3353 times objects are detected but assigned to an ‘unknown’ room, when the perception system fails
3354 to assign a room. Note that the object and its location is still added to the graph using RGBD
3355 observations, but doesn’t have the right parent receptacle or room in the graph. Such situations lead
3356 to reasoning problems with the LLM which does not realize that the object is still navigable. In the
3357 following example, the LLM keeps exploring rooms looking for the kettle, even though it is already
3358 in the graph, but in an ‘unknown’ room.

3359 **Room Association Failure**

3360
3361 Objects: 122_clock: 94_bed in bedroom.
3362 126_phone_stand: 94_bed in bedroom.
3363 126_plant_saucer: 65_table in bedroom.
3364 126_kettle: unknown in an unknown
room.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

3365 Thought: I successfully explored the hallway, but I didn't find the kettle. Let me try
3366 to explore the kitchen, as it's a more likely place to find a kettle.
3367 Explore[kitchen]
Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>

3368 Result: Successful execution!
3369 Objects: 122_clock: 94_bed in bedroom.
3370 126_phone_stand: 94_bed in bedroom.
3371 126_plant_saucer: 65_table in bedroom.
3372 126_kettle: unknown in an unknown
room.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

3373 Thought: I successfully explored the kitchen, but I still didn't find the kettle. Let me
3374 try to explore the bathroom, as it's another possible location.
3375 Explore[bathroom]
Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>

3376 Result: Unexpected failure! - Skill took too long to finish.
3377 Objects: 122_clock: 94_bed in bedroom.
3378 126_phone_stand: 94_bed in bedroom.
3379 126_plant_saucer: 65_table in bedroom.
3380 126_kettle: unknown in an unknown room.
127_toy_airplane: 67_mirror in
bedroom.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

3381 Thought: The exploration of the bathroom timed out. I still haven't found the kettle.
3382 Since I've already explored the hallway, kitchen, and bathroom, my next step would be to
3383 explore the living room. However, I don't see the living room as an option. Let me try
3384 to explore the bedroom again, as it's a known location and I might have missed something.
3385 Explore[bedroom]
Assigned!<|eot_id|>

3386
3387 **Failures in communication between the high-level and low-level perception systems result in**
3388 **planner failures:** Our system relies on multiple layers of perception, e.g. high-level perception
3389 that tracks the state-of-the-world and low-level perception for skills to act and inform the planner of
3390 successful or unsuccessful execution. We see lower-level perception sometimes provides the LLM the
3391 wrong feedback about skill execution. Even if the high-level perception gives the correct information,
3392 the LLM is not able to reconcile these two sources of information, and tends to get confused. In the
3393 attached trace, for example, the LLM planner believes the agent has both ramekin and potato in hand
3394 and tries to place them together on the same receptacle. In our simulation, the agent is only allowed
3395 to grasp one object at a time, but the low-level perception does not send this error message back to
3396 the planner correctly.

3397 **High-Level and Low-Level Perception Mismatch**

3398
3399 Thought: I have successfully navigated to the living room and I can see the ramekin and
3400 potato on the carpet. I will now try to pick up the ramekin.
3401 Pick[88_ramekin]
Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>

```

3402
3403   Result: Successful execution!
3404
3405   88_ramekin: held by the agent. <-- ramekin in agent's hand (correct high-level)
3406   89_potato: 76_carpet in living_room.
3407   90_ramekin: 37_chest_of_drawers in bedroom.
3408   91_cup: 13_chest_of_drawers in bedroom.
3409   92_book: 24_stool in kitchen.
3410   93_potato: 66_potted_plant in
3411   bedroom.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
3412
3413   Thought: I have successfully picked up the ramekin. Now, I will try to pick up the
3414   potato that is on the carpet in the living room.
3415   Pick[89_potato]
3416   Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
3417
3418   Result: Successful execution! <-- agents can not pick up multiple objects; wrong
3419   low-level perception and feedback
3420   Objects: 81_cup: 32_shelves in unknown.
3421   88_ramekin: held by the agent. <-- still held by the agent
3422   89_potato: 76_carpet in living_room. <-- not held by the agent
3423   90_ramekin: 37_chest_of_drawers in bedroom.
3424   91_cup: 13_chest_of_drawers in bedroom.
3425   92_book: 24_stool in kitchen.
3426   93_potato: 66_potted_plant in
3427   bedroom.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
3428
3429   Thought: I have successfully picked up the potato. Now, I will navigate to the table in
3430   the living room to place the ramekin and potato.
3431   Navigate[10_chair]
3432   Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
3433
3434   Result: Successful execution!
3435   <summary>
3436
3437   Thought: I have successfully navigated to the chair in the living room, but I need to
3438   place the ramekin and potato on the table. Since the task mentions placing them next to
3439   each other on the chair in the dining room, I will navigate to the dining room instead.
3440   Navigate[47_table]
3441   Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
3442
3443   Result: Successful execution!
3444   <summary>
3445
3446   Thought: I have successfully navigated to the table in the dining room. Now, I will
3447   place the ramekin on the table.
3448   Place[88_ramekin, on, 47_table, next_to, 89_potato]
3449   Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
3450
3451   Result: Unexpected failure! - No valid placements found for entity 47_table. <-- no
3452   placements were found since placement asked for object to be placed next-to 89_potato,
3453   while 89_potato is actually on 76_carpet. Planner keeps repeating this placement on
3454   different receptacles until episode runs out.
3455

```

Failing to associate objects over time reduces the ability to do task progress: PARTNR specifically requires agents to reason about task progression and the state of the world as objects are moved around by both agents. Even though the robot agent receives observations from both its own sensors and human’s sensor, object temporal reassociation is a hard problem. This leads to complex failures even if the objects are correctly identified. For example, in the trace below, the environment actually only has two bees and 1 toy-truck; however the graph is filled with duplicate copies since the agent fails to associate different toy bee detections to the same instance. This leads to the LLM forgetting the context and the goal, resulting in erroneous episode termination as well as increased inefficiency due to trying rearrangements over objects that do not exist.

Task Progress Failure

```

3450   Thought: Now that I have navigated to the shelves in the living room, I should pick up
3451   the toy bee.
3452   Pick[130_toy_bee]
3453   Assigned!<|eot_id|><|start_header_id|>user<|end_header_id|>
3454
3455   Result: Successful execution!
3456   Objects: 122_picture_frame: 67_mirror in bedroom.
3457   124_toy_bee: unknown in an unknown room.

```


3456 125_toy_bee: 0_carpet in living_room.
3457 126_toy_bee: 0_carpet in living_room.
3458 127_toy_bee: unknown in an unknown room.
3459 128_toy_bee: 53_carpet in living_room.
3460 129_toy_bee: 53_carpet in living_room.
3461 130_toy_bee: held by the agent.
3462 131_toy_bee: 78_shelves in living_room.
3463 132_toy_bee: 32_chest_of_drawers in living_room.
3464 133_toy_bee: 32_chest_of_drawers in living_room.
3465 134_toy_bee: 19_tv in living_room.
3466 135_toy_bee: 78_shelves in living_room.
3467 136_toy_bee: unknown in an unknown room.
3468 137_toy_bee: 78_shelves in living_room.
3469 139_toy_bee: 78_shelves in living_room.
3470 140_cup: 133_chest_of_drawers in bedroom.
3471 141_toy_bee: 32_chest_of_drawers in living_room.
3472 142_toy_bee: 32_chest_of_drawers in living_room.
3473 143_picture_frame: 4_chest_of_drawers in bedroom.
3474 144_picture_frame: 4_chest_of_drawers in bedroom.
3475 145_picture_frame: unknown in an unknown room.
3476 146_picture_frame: unknown in an unknown room.
3477 147_toy_bee: 19_tv in living_room.
3478 148_toy_bee: 32_chest_of_drawers in living_room.
3479 149_toy_bee: 32_chest_of_drawers in living_room.
3480 150_toy_bee: 0_carpet in living_room.
3481 151_toy_bee: 32_chest_of_drawers in living_room.
3482 152_toy_bee: 0_carpet in living_room.
3483 153_toy_bee: 0_carpet in living_room.
3484 154_toy_bee: 32_chest_of_drawers in living_room.
3485 155_toy_fire_truck: 78_shelves in living_room.
3486 154_toy_fire_truck: 19_tv in living_room.
3487 156_toy_fire_truck: held by the other
3488 agent.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
3489 Thought: I have successfully picked up the toy bee from the shelves in the living room.
3490 My task is now complete. I will end the simulation here.
3491 Done[]
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509