

TEACH: Task-driven Embodied Agents that Chat

Aishwarya Padmakumar^{* 1}, Jesse Thomason^{* 1 2}, Ayush Srivastava³, Patrick Lange¹, Anjali Narayan-Chen¹, Spandana Gella¹, Robinson Piramuthu¹, Gokhan Tur¹, Dilek-Hakkani Tur¹

¹ Amazon Alexa AI

² USC Viterbi Department of Computer Science, University of Southern California

³ Department of Electrical Engineering And Computer Science, University of Michigan

padmakua@amazon.com, jessedt@amazon.com, ayshrv@umich.edu, patlange@amazon.com, naraanja@amazon.com, sgella@amazon.com, robinpir@amazon.com, gokhatur@amazon.com, hakkanit@amazon.com

Abstract

Robots operating in human spaces must be able to engage in natural language interaction with people, both understanding and executing instructions, and using conversation to resolve ambiguity and recover from mistakes. To study this, we introduce *TEACH*, a dataset of over 3,000 human–human, interactive dialogues to complete household tasks in simulation. A *Commander* with access to oracle information about a task communicates in natural language with a *Follower*. The *Follower* navigates through and interacts with the environment to complete tasks varying in complexity from MAKE COFFEE to PREPARE BREAKFAST, asking questions and getting additional information from the *Commander*. We propose three benchmarks using *TEACH* to study embodied intelligence challenges, and we evaluate initial models’ abilities in dialogue understanding, language grounding, and task execution.

1 Introduction

Many benchmarks for translating visual observations and language instructions to actions assume that an agent is initially provided with language instructions, and expect the agent to predict a complete action sequence using visual observations updated as actions are taken, but with no further language communication (Anderson et al. 2018; Shridhar et al. 2020). However, prior work has demonstrated the benefit of obtaining clarification via simulated interactions (Chi et al. 2020; Nguyen and Daumé III 2019) and understanding natural language clarifications (Thomason et al. 2019) in improving embodied navigation. Additionally, human-human dialogue history for navigation can be used as an effective tool for disambiguating navigation targets (Thomason et al. 2019; Suhr et al. 2019). We hypothesize that dialogue has even more to offer when tasks additionally require object interaction and state changes and have inherent hierarchical structure.

We introduce *Task-driven Embodied Agents that Chat* (*TEACH*) - a dataset to study how agents can learn to ground natural language (Harnad 1990; Bisk et al. 2020) to the visual world and actions, while considering long-term and

intermediate goals, and using dialogue to communicate. *TEACH* contains over 3,000 human–human sessions interleaving utterances and environment actions where a *Commander* with oracle task and world knowledge and a *Follower* with the ability to interact with the world communicate in written English to complete household chores (Figure 1).

TEACH dialogues are unconstrained, not turn-based, yielding variation in instruction granularity, completeness, relevance, and overlap. Utterances include coreference with previously mentioned entities, past actions, and locations. Because *TEACH* sessions are human, rather than planner-based (Ghallab et al. 1998), *Follower* trajectories include mistakes and corresponding, language-guided backtracking and correction.

We propose three benchmarks based on *TEACH* sessions to study the ability of learned models to achieve aspects of embodied intelligence: Execution from Dialog History (EDH), Trajectory from Dialog (TfD) and Two-Agent Task Completion (TATC). We also demonstrate baseline performance on these benchmarks. To model the *Follower* agent for the EDH and TfD benchmarks, we build on the Episodic Transformer (E.T.) model (Pashevich, Schmid, and Sun 2021) that has performed well on ALFRED (Shridhar et al. 2020) as a baseline. When modeling both agents for end-to-end task completion, we demonstrate the difficulty of engineering rule-based solvers.

The main contributions of this work are:

- *TEACH*, a dataset of over 3000 human-human dialogs simulating the experience of a user interacting with their robot to complete tasks in their home, that interleaves dialogue messages with actions taken in the environment.
- An extensible task definition framework (§3) that can be used to define and check completion status for a wide range of tasks in a simulated environment.
- Three benchmarks based on *TEACH* sessions and experiments demonstrating initial models for each.

2 Related Work

Table 1 situates *TEACH* with respect to other datasets involving natural language instructions for visual task completion.

^{*}Authors contributed equally

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Dataset	— Object —		— Language —		Demonstrations	
	Interaction	State Changes	Conversational	# Sessions	Freeform	
R2R (Anderson et al. 2018)	✗	✗	✗	-	-	Planner
CHAI (Misra et al. 2018)	✓	✓	✗	-	-	Human
CVDN (Thomason et al. 2019)	✗	✗	✓	2050	✗	Human
CerealBar (Suhr et al. 2019)	✓	✗	✓	1202	✗	Human
MDC (Narayan-Chen et al. 2019)	✓	✗	✓	509	✓	Human
ALFRED (Shridhar et al. 2020)	✓	✓	✗	-	-	Planner
III (Abramson et al. 2020)	✓	✗	✗	-	-	Human
<i>TEACH</i>	✓	✓	✓	3215	✓	Human

Table 1: *TEACH* is the first dataset where human-human, conversational dialogues were used to perform tasks involving object interaction, such as picking up a knife, and state changes, such as slicing bread, in a visual simulation environment. *TEACH* task demonstrations are created by the human *Follower*, who engages in a free-form, rather than turn-taking, dialogue with the human *Commander*. Compared to past dialogue datasets for visual tasks, *TEACH* contains many more individual dialogues.

Vision & Language Navigation (VLN) tasks agents with taking in language instructions and a visual observation to produce an action, such as turning or moving forward, to receive a new visual observation. VLN benchmarks have evolved from the use of symbolic environment representations (MacMahon, Stankiewicz, and Kuipers 2006; Chen and Mooney 2011; Mei, Bansal, and Walter 2016) to photorealistic indoor (Anderson et al. 2018) and outdoor environments (Chen et al. 2019), as well as the prediction of continuous control (Blukis et al. 2018). *TEACH* goes beyond navigation to object interactions for task completion, and beyond single instructions to dialogue.

Vision & Language Task Completion involves actions beyond navigation. Models have evolved from individual rule-based or learned components for language understanding, perception and action execution (Matuszek et al. 2013; Kollar et al. 2013), to end-to-end models in fully observable blocks worlds (Bisk et al. 2018; Misra et al. 2018). More complex tasks involve partially observable worlds (Kim et al. 2020) and object state changes (Misra et al. 2018; Puig et al. 2018; Shridhar et al. 2020). Some works use a planner to generate ideal demonstrations that are then labeled, while others first gather instructions and gather human demonstrations (Misra et al. 2018; Shah et al. 2021; Abramson et al. 2020). In *TEACH*, human instructions and demonstrations are gathered simultaneously.

Vision & Dialogue Navigation and Task Completion Agents that engage in dialogue instead of simply following natural language instructions can be learned by combining individual rule-based or learned components (Tellex et al. 2016; Arumugam et al. 2018; Thomason et al. 2020). Simulated clarification can also improve end-to-end VLN models (Chi et al. 2020; Nguyen and Daumé III 2019). Models are also able to take advantage of conversational history in human-human dialogues to perform better navigation (Thomason et al. 2019; Zhu et al. 2020), learn agent-agent policies for navigating and speaking (Roman et al. 2020; Srivastava et al. 2021), and deploy individual agent policies for human-in-the-loop evaluation (Suhr et al. 2019). However, such models and underlying datasets are limited to

navigation actions and turn-taking conversation. In contrast, *TEACH* involves *Follower* navigation and object interaction, as well as freeform dialogue acts with the *Commander*. The Minecraft Dialogue Corpus (MDC) (Narayan-Chen, Jayanavar, and Hockenmaier 2019) gives full dialogues between two humans for assembly tasks. MDC is similar in spirit to *TEACH*; we introduce a larger action space and resulting object state changes, such as slicing and toasting bread, as well as collecting many more human-human dialogues.

3 The *TEACH* Dataset

We collect 3,047 human-human *gameplay sessions* for completing household tasks in the AI2-THOR simulator (Kolve et al. 2017). Each session includes an initial environment state, *Commander* actions to access oracle task and object information, utterances between the *Commander* and *Follower*, movement actions taken by each, and object interactions taken by the *Follower*. Figure 2 gives an overview of the annotation interface.

3.1 Household Tasks

We design a *task definition language* (TDL) to define household tasks in terms of object properties to satisfy, and implement a framework over AI2-THOR that evaluates these criteria. For example, for a task to make coffee, we consider the environment to be in a successful state if there is a mug in the environment that is clean and filled with coffee.

Parameterized tasks such as PUT ALL X ON Y enable task variation. Parameters can be object classes, such as putting all *forks* on a *countertop*, or predefined abstract hypernyms, for example putting all *silverware*—forks, spoons, and knives—on the counter. *TEACH* task definitions are also hierarchical. For example, PREPARE BREAKFAST contains the subtasks MAKE COFFEE and MAKE PLATE OF TOAST. We also incorporate handling of determiners such as “a”, “all” and numbers such as 2 to enable easy definition of a wide range of tasks, such as N SLICES OF X IN Y. The *TEACH* TDL includes template-based language prompts to describe tasks and subtasks to *Commanders* (Figure 3).

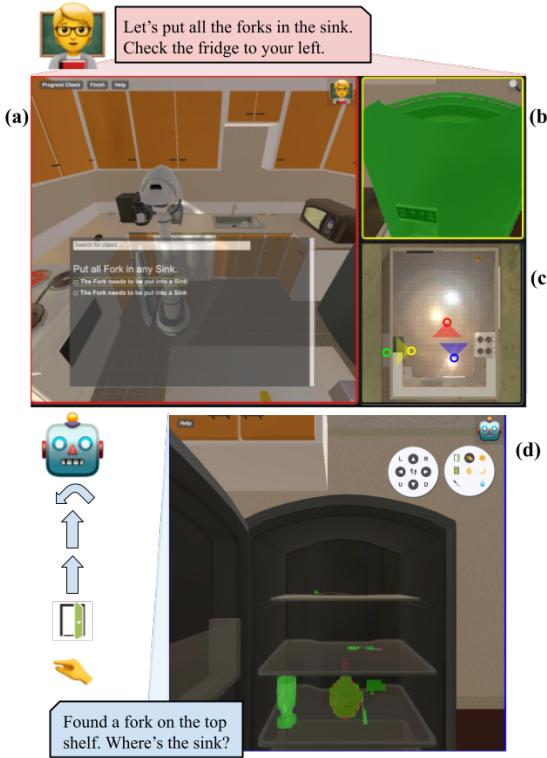


Figure 1: The *Commander* has oracle task details (a), object locations (b), a map (c), and egocentric views from both agents. The *Follower* carries out the task and asks questions (d). The agents can only communicate via language.

3.2 Gameplay Session Collection

Annotators first completed a tutorial task demonstrating the interface to vet their understanding. For each session, two vetted crowdworkers were paired using a web interface and assigned to the *Commander* and *Follower* roles (Figure 2). The *Commander* is shown the task to be completed and the steps needed to achieve this given the current state of the environment, using template-based language prompts, none of which are accessible to the *Follower*. The *Commander* can additionally search for the location of objects, either by string name, such as “sink”, or by clicking a task-relevant object in the display (Figure 3). The *Commander* and *Follower* must use text chat to communicate the parameters of the task and clarify object locations. Only the *Follower* can interact with objects in the environment.

We obtained initial states for each parameterized task by randomizing AI2-THOR environments and retaining those that satisfied preconditions such as task-relevant objects being present and reachable. For each session, we store the initial simulator state S_i , the sequence of actions $A = (a_1, a_2, \dots)$ taken, and the final simulator state S_f . *TEACH* *Follower* actions are Forward, Backward, Turn Left, Turn Right, Look Up, Look Down, Strafe Left, Strafe Right, Pickup, Place, Open, Close, ToggleOn, ToggleOff, Slice, and Pour. Navigation actions move the agent in discrete steps.

Object manipulation expects the agent to specify an object via a relative coordinate (x, y) on *Follower* egocentric frame. The *TEACH* wrapper on the AI2-THOR simulator examines the ground truth segmentation mask of the agent’s egocentric image, selects an object in a 10×10 pixel patch around the coordinate if the desired action can be performed on it, and executes the action in AI2-THOR. The *Commander* can execute a Progress Check and SearchObject actions, demonstrated in Figure 3. *TEACH Commander* actions also allow navigation, but the *Commander* is a disembodied camera.

3.3 *TEACH* Statistics

TEACH is comprised of 3,047 successful gameplay sessions, each of which can be replayed using the AI2-THOR simulator for model training, feature extraction, or model evaluation. In total, 4,365 crowdsourced sessions were collected with a human-level success rate of 74.17% (3320 sessions) and total cost of \$105k; more details in appendix. Some successful sessions were not included in the final split used in benchmarks due to replay issues. *TEACH* sessions span all 30 AI2-THOR kitchens, and include most of the 30 each AI2-THOR living rooms, bedrooms, and bathrooms.

Successful *TEACH* sessions consist of over 45k utterances, with an average of 8.40 *Commander* and 5.25 *Follower* utterances per session. The average *Commander* utterance length is 5.70 tokens and the average *Follower* utterance length is 3.80 tokens. The *TEACH* data has a vocabulary size of 3,429 unique tokens.¹ Table 2 summarizes such metrics across the 12 task types in *TEACH*. Simple tasks like MAKE COFFEE require fewer dialogue acts and *Follower* actions on average than complex, composite tasks like PREPARE BREAKFAST which subsume those simpler tasks.

4 *TEACH* Benchmarks

We introduce three benchmarks based on *TEACH* sessions to train and evaluate the ability of embodied AI models to complete household tasks using natural language dialogue. **Execution from Dialogue History** and **Trajectory from Dialogue** require modeling the *Follower*. **Two-Agent Task Completion**, by contrast, requires modeling both the *Commander* and *Follower* agents to complete *TEACH* tasks end-to-end. For each benchmark, we define how we derive benchmark instances from *TEACH* gameplay sessions, and by what metrics we evaluate model performance.

Each session has an initial state S_i , the sequence of actions $A = (a_1, a_2, \dots)$ taken by the *Commander* and *Follower* including dialogue and environment actions, and the final state S_f . We denote the subsequence of all dialogue actions as A^D , and of all navigation and interaction as A^I . Following ALFRED, we create validation and test splits in both seen and unseen environments (Table 3). Seen splits contain sessions based in AI2-THOR rooms that were seen in the training, whereas unseen splits contain only sessions in rooms absent from the training set.

¹Using the spaCy tokenizer: <https://pypi.org/project/spacy/>



Figure 2: To collect *TEACH*, the *Commander* knows the task to be completed and can query the simulator for object locations. Searched items are highlighted in green for the *Commander*; highlights blink to enable seeing the underlying true scene colors. The *Commander* has a topdown map of the scene, with the current camera position shown in red, the *Follower* position shown in blue, and the object search camera position shown in yellow. The *Follower* moves around in the environment and interacts with objects, such as placing a fork (middle). Target objects for each interaction action are highlighted.

	Parameter Variants	Unique Scenes	Total Sessions	Utterances per Session	<i>Follower</i> Actions/Session	All Actions/Session
WATER PLANT	1	10	176	6.37± 4.36	51.86± 30.71	67.93± 40.70
MAKE COFFEE	1	30	308	7.75± 5.08	55.25± 33.61	72.29± 50.85
CLEAN ALL X	19	52	336	9.65± 7.03	74.06± 59.66	96.92± 71.31
PUT ALL X ON Y	209	92	344	8.66± 5.82	82.13± 66.39	103.53± 80.97
BOIL POTATO	1	26	202	10.65± 7.61	104.66± 79.50	130.13± 94.80
MAKE PLATE OF TOAST	1	27	225	12.26± 8.51	108.30± 55.81	136.11± 70.73
N SLICES OF X IN Y	16	29	304	13.50±10.86	113.62± 94.25	146.23±113.96
PUT ALL X IN ONE Y	84	50	302	11.32± 7.03	115.74± 90.13	147.80±104.45
N COOKED X SLICES IN Y	10	30	240	14.94± 9.43	155.18± 75.17	189.26± 87.90
PREPARE SANDWICH	5	28	241	18.03± 9.96	195.93± 83.96	241.61±100.86
PREPARE SALAD	9	30	323	20.47±10.80	206.29±111.47	253.94±130.09
PREPARE BREAKFAST	80	30	308	27.67±14.73	295.06±138.76	359.90±162.33
TEACH Overall	438	109	3320	13.67±10.81	131.80±109.68	164.65±130.89

Table 2: The 12 tasks represented in *TEACH* sessions vary in complexity. Tasks like PUT ALL X ON Y take object class parameters and can require more actions per session to finish. Composite tasks like PREPARE SALAD contain sub-tasks like N SLICES OF X IN Y. Per session data are averages with standard deviation across task types.

Fold	Split	# Sessions	# EDH Instances
Train		1482 (49%)	6603 (48%)
Val	Seen	181 (6%)	756 (5%)
	Unseen	614 (20%)	2615 (19%)
Test	Seen	181 (6%)	888 (6%)
	Unseen	589 (19%)	2963 (22%)

Table 3: Session and EDH instances in *TEACH* data splits.

4.1 Execution from Dialogue History (EDH)

We segment *TEACH* sessions into EDH instances. We construct EDH instances (S^E, A_H, A_R^I, F^E) where S^E is the initial state of the EDH instance, A_H is an action history, and the agent is tasked with predicting a sequence of actions that changes the environment state to F^E , using A_R^I reference interaction actions taken in the session as supervision.

We constrain instances to have $|A_H^D| > 0$ and at least one object interaction in A_R^I . Each EDH instance is punctuated by a dialogue act starting a new instance or the session end. We append a *Stop* action to each A_R^I . An example is included in Figure 4.

To evaluate inferred EDH action sequences, we compare the simulator state changes \hat{E} at the end of inference with F^E using similar evaluation criteria generalized from the ALFRED benchmark.

- Success {0, 1}: 1 if all expected state changes F^E are present in \hat{E} , else 0. We average over all trajectories.
- Goal-Condition Success (GC) (0, 1): The fraction of expected state changes in F^E present in \hat{E} . We average over all trajectories.²

²We follow ALFRED in using a macro-, rather than micro-average for Goal-Conditioned Success Rate.

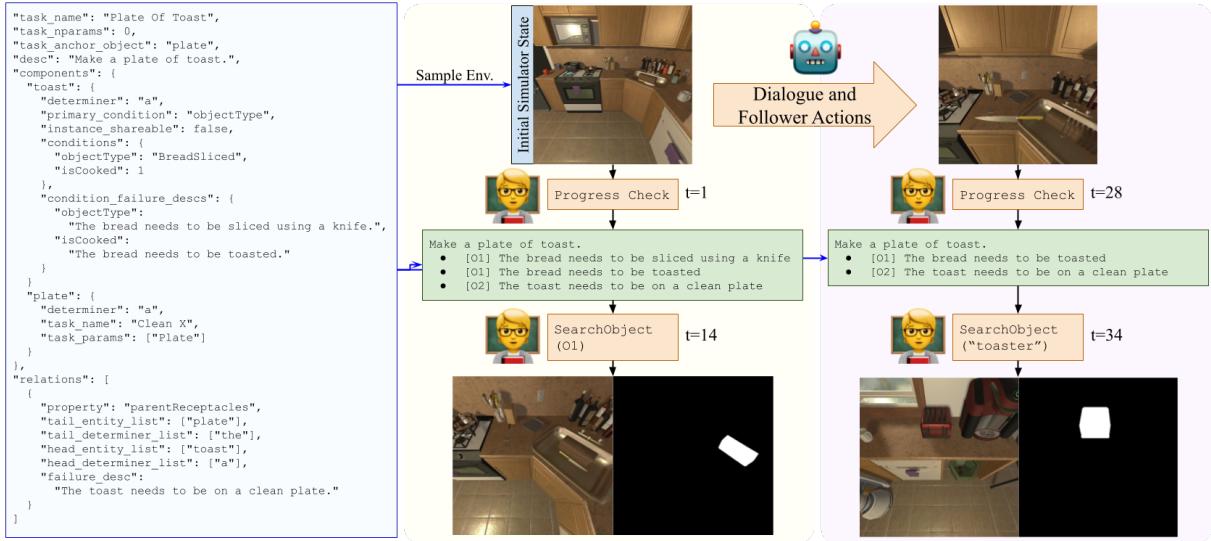


Figure 3: An example task definition from the *TEACH* task definition language (left) and how it informs the initial simulator state and the *Commander* Progress Check action. The *Commander* can SearchObject with a string query (right) or object instance (center) returned by the Progress Check task status, yielding a camera view, segmentation mask, and location.

- Trajectory Weighted Metrics: For a reference trajectory A_R^I and inferred action sequence \hat{A}^I , we calculate trajectory length weighted metric for metric value m as

$$TLW-m = \frac{m * |A_R^I|}{\max(|A_R^I|, |\hat{A}^I|)}.$$

During inference, the learned *Follower* agent predicts actions until either it predicts the Stop action, hits a limit of 1000 steps, or hits a limit of 30 failed actions.

4.2 Trajectory from Dialogue (TfD)

A *Follower* agent model is tasked with inferring the whole sequence of *Follower* environmental actions taken during the session conditioned on the dialogue history. A TfD instance is (S_i, A_H^D, A_R^I, S_f) , where A_H^D is all dialogue actions taken by both agents, and A_R^I is all non-dialogue actions taken by the *Follower*. We append a Stop action to A_R^I . The agent does not observe dialogue actions in context, however, we use this task to test long horizon action prediction with a block of instructions, analogous to ALFRED or TouchDown (Chen et al. 2019). We calculate success and goal-conditioned success by comparing \hat{E} against state changes between S_i and S_f .

4.3 Two-Agent Task Completion (TATC)

To explore modeling both a *Commander* and *Follower* agent, the TATC benchmark gives as input only environment observations to both agents. The *Commander* model must use the Progress Check action to receive task information, then synthesize that information piece by piece to the *Follower* agent via language generation. The *Follower* model can communicate back via language generation. The TATC benchmark represents studying the “whole” set of

challenges the *TEACH* dataset provides. We calculate success and goal-conditioned success by comparing \hat{E} against state changes between S_I and S_f .

5 Experiments and Results

We implement initial baseline models and establish the richness of *TEACH* data and difficulty of resulting benchmarks.

5.1 Follower Models for EDH and TfD

We use a single model architecture to train and evaluate on the EDH and TfD benchmark tasks.

Model. We establish baseline performance for the EDH and TfD tasks using the Episodic Transformer (E.T.) model (Pashevich, Schmid, and Sun 2021), designed for the ALFRED benchmark. The original E.T. model trains a transformer language encoder and uses a ResNet-50 backbone to encode visual observations. Two multimodal transformer layers are used to fuse information from the language, image, and action embeddings, followed by a fully connected layer to predict the next action and target object category for interaction actions. E.T. uses a MaskRCNN (He et al. 2017) model pretrained on ALFRED images to predict a segmentation of the egocentric image for interactive actions, matching the predicted mask to the predicted object category. We convert the centroid of this mask to a relative coordinate specified to the *TEACH* API wrapper for AI2-THOR.

We modify E.T. by learning a new action prediction head to match *TEACH Follower* actions. Given an EDH or TfD instance, we extract all dialogue utterances from the action history A_H^D and concatenate these with a separator between utterances to form the language input. The remaining actions A_H^I are fed in order as the past action input with associated image observations. Consequently, our adapted E.T.

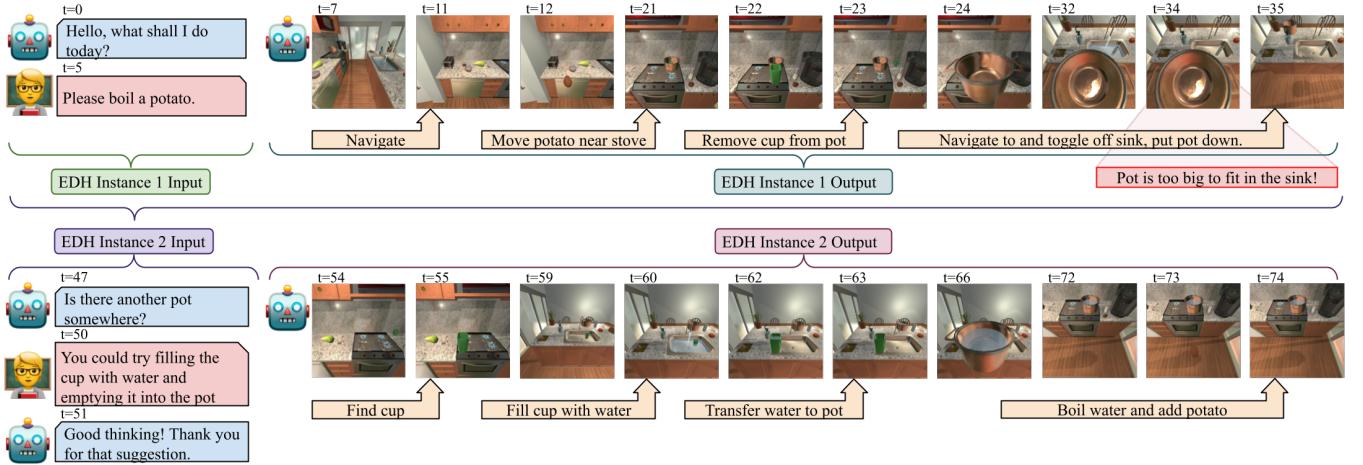


Figure 4: Two EDH instances are constructed from this real example from the *TEACH* data. The first instance input contains only dialogue actions. After inference on the first instance, the agent is evaluated based on whether it moved the potato, pot, and the items cleared out of the sink to their target destinations. In this example, the pot cannot fit into the sink. The second instance input has both dialogue and environment actions, and is evaluated at inference by whether the pot lands on the stove filled with water, and whether the potato is inside the pot and boiled.

does not have temporal alignment between dialogue actions and environment actions.

Following the mechanism used in the original E.T. paper, we provide image observations from both actions in the history A_H^I , and the reference actions A_R^I , and task the model to predict the entire sequence of actions. The model parameters are optimized using cross entropy loss between the predicted action sequence and the ground truth action sequence. For EDH, we ablate a history loss (H) as cross entropy over the entire action sequence—actions in both A_H^I and A_R^I , to compare against loss only against actions in A_R^I . Note that in TfD, $|A_H^I| = 0$.

We additionally experiment with initializing the model using weights trained on the ALFRED dataset. Note that since the language vocabulary and action space change, some layers need to be retrained. For EDH, we experiment with initializing the model both with weights from the E.T. model trained only on base ALFRED annotations (A) and the model trained on ALFRED augmented with synthetic instructions (S) (from Pashevich, Schmid, and Sun (2021)). We also perform unimodal ablations of the E.T. model to determine whether the model is simply memorizing sequences from the training data (Thomason, Gordon, and Bisk 2018).

At inference time, the agent uses dialogue history as language input, and the environment actions in A_H^I as past action input along with their associated visual observations. At each time step we execute the predicted action, with predicted object coordinate when applicable, in the simulator. The predicted action and resulting image observation are added to agent’s input for the next timestep. The appendix details model hyperparameters.

Results. Table 4 summarizes our adapted E.T. model performance on the EDH and TfD benchmarks.

We observe that all E.T. model conditions in EDH are significantly better than Random and Lang-Only con-

dition on all splits on SR and GC, according to a paired two-sided Welch t -test with Bonferroni corrections. Compared to the Vision-Only baseline, the improvements of the E.T. models are statistically significant on unseen splits, but not on seen splits. Qualitatively, we observe that the Random baseline only succeeds on very short EDH instances that only include one object manipulation involving a large target object, for example placing an object on a countertop. The same is true of most of the successful trajectories of the Lang-Only baseline. The success rate of the Vision-Only baseline suggests that the E.T.-based models are not getting much purchase with language signal. Notably, E.T. performs well below its success rates on ALFRED, where it achieves 38.24% on the ALFRED test-seen split and 8.57% on the ALFRED test-unseen split. Additionally, although there appears to be a small benefit from initializing the E.T. model with pretrained weights from ALFRED, these differences are not statistically significant. *TEACH* language is more complex, involving multiple speakers, irrelevant phatic utterances, and dialogue anaphora.

E.T. model performance on TfD is poor but non-zero, unlike a Random baseline. We do not perform additional ablations for TfD given the low initial performance. Notably, in addition to the complexity of language, TfD instances have substantially longer average trajectory length (~ 130) than those in ALFRED (~ 50).

5.2 Rule-based Agents for TATC

In benchmarks like ALFRED, a PDDL (Ghallab et al. 1998) planner can be used to determine what actions are necessary to solve relatively simple tasks. In VLN, simple search algorithms yield the shortest paths to goals. Consequently, some language-guided visual task models build a semantic representation of the environment, then learn a hierarchical policy to execute such planner-style goals (Blukis et al. 2021).

Model	EDH Validation				EDH Test			
	Seen		Unseen		Seen		Unseen	
	SR [TLW]	GC [TLW]	SR [TLW]	GC [TLW]	SR [TLW]	GC [TLW]	SR [TLW]	GC [TLW]
Rand	0.68 [0.14]	0.40 [3.44]	0.46 [0.04]	0.25 [1.09]	0.83 [0.18]	0.45 [1.40]	0.72 [0.18]	0.30 [0.81]
Lang	1.14 [0.06]	0.64 [0.54]	0.46 [0.02]	0.26 [0.13]	0.83 [0.04]	0.80 [0.51]	0.83 [0.05]	0.93 [0.87]
Vision	5.14 [0.23]	5.51 [5.00]	2.76 [0.11]	2.19 [2.14]	4.17 [0.21]	5.42 [5.25]	2.99 [0.17]	2.67 [2.62]
E.T.	7.88 [0.64]	8.19 [8.60]	5.06 [0.31]	4.38 [5.26]	5.83 [0.47]	7.79 [11.70]	5.91 [0.35]	6.14 [7.83]
+H	6.05 [0.33]	8.08 [11.80]	4.71 [0.25]	3.62 [4.87]	6.55 [0.29]	8.08 [9.81]	4.81 [0.36]	5.98 [7.53]
+A	6.62 [0.49]	9.77 [12.99]	4.35 [0.24]	3.97 [4.52]	6.67 [0.43]	10.26 [14.22]	5.72 [0.26]	6.63 [6.30]
+H+A	6.74 [0.46]	7.89 [9.84]	3.89 [0.36]	3.32 [6.51]	7.26 [0.45]	8.43 [8.89]	4.58 [0.37]	5.41 [7.90]
+S	7.88 [0.62]	8.23 [9.96]	4.71 [0.25]	4.88 [5.42]	5.24 [0.37]	9.68 [17.90]	6.17 [0.30]	7.92 [8.61]
+H+S	6.85 [0.36]	9.24 [11.45]	4.35 [0.27]	4.43 [5.80]	6.67 [0.41]	8.62 [9.72]	4.51 [0.28]	4.92 [6.59]
TfD Validation				TfD Test				
Rand	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
E.T.	1.02 (0.17)	1.42 (4.82)	0.48 (0.12)	0.35 (0.59)	0.51 (0.23)	1.60 (6.46)	0.17 (0.04)	0.67 (2.50)

Table 4: E.T. outperforms random and unimodal baselines (**bold**). We ablate history loss (H), initializing with ALFRED (A), and initializing with ALFRED synthetic language (S). Metrics are success rate (SR) and goal condition success rate (GC). Trajectory length weighted metrics are included in [brackets]. All values are percentages. For all metrics, higher is better.

Task (Shrnd)	Success Rate	Rule Agent Actions/Session	Human Actions/Session
PLANT	26.70	230.26± 54.65	67.93± 40.70
COFFEE	54.55	120.24± 66.55	72.29± 50.85
CLEAN	52.98	182.38± 79.84	96.92± 71.31
ALL X Y	52.91	126.82± 64.75	103.53± 80.97
N SLICES	22.51	248.77± 98.57	146.23±113.96
X ONE Y	50.98	150.09± 97.12	147.80±104.45
COOKED	1.67	424.25±135.57	189.26± 87.90
SALAD	1.55	351.20± 82.09	253.94±130.09
Overall	24.40	161.54± 92.00	164.65±130.89

Table 5: Rule-based agent policies were expansive enough to solve some simple tasks about half the time, while being unable to solve most compositional tasks at all. For brevity, we omit rows where rule-based agents achieve no success, though these are included in the overall success rate. Note that TATC performance is not directly comparable to EDH or TfD due to two-agent modeling in TATC.

Inspired by such planning-based solutions, we attempted to write a pair of rule-based *Commander* and *Follower* agents to tackle the TATC benchmark. In a loop, the rule-based *Commander* executes a Progress Check action, then forms a language utterance to the *Follower* consisting of navigation and object interaction actions needed to accomplish the next sub-goal in the response. Each sub-goal needs to be identified by the language template used to describe it, then a hand-crafted policy must be created for the rule-based *Commander* to reference. For example, for the PUT ALL X ON Y task, all sub-goals are of the form “X needs to be on some Y” for a particular instance of object X, and so a rule-based policy can be expressed as “navigate to the X instance, pick up the X instance, navigate to Y, put X down on Y.” *Commander* utterances are simplified to se-

quences of action names with a one-to-one mapping to *Follower* actions to execute, with interaction actions including (x, y) screen click positions to select objects. The rule-based agents perform *no learning*.

Table 5 summarizes the success rate of these rule-based agents across task types. Note that for the tasks BOIL POTATO, MAKE PLATE OF TOAST, MAKE SANDWICH, and BREAKFAST, sub-goal policies were not successfully developed. The rule-based agents represent about 150 hours of engineering work to hand-craft subgoal policies. While success rates could certainly be increased by increasing subgoal policy coverage and handling simulation corner cases, it is clear that, unlike ALFRED and navigation-only tasks, a planner-based solution is not reasonable for *TEACH* data and the TATC benchmark. The appendix contains detailed implementation information about the rule-based agents.

6 Conclusions and Future Work

We introduce *Task-driven Embodied Agents that Chat (TEACH)*, a dataset of over 3000 situated dialogues in which a human *Commander* and human *Follower* collaborate in natural language to complete household tasks in the AI2-THOR simulation environment. *TEACH* contains dialogue phenomena related to grounding dialogue in objects and actions in the environment, varying levels of instruction granularity, and interleaving of utterances between speakers in the absence of enforced turn taking. We also introduce a task definition language that is extensible to new tasks and even other simulators. We propose three benchmarks based on *TEACH*. To study *Follower* models, we define the Execution from Dialogue History (EDH) and Trajectory from Dialogue (TfD) benchmarks, and evaluate an adapted Episodic Transformer (Pashevich, Schmid, and Sun 2021) as an initial baseline. To study the potential of *Commander* and *Follower* models, we define the Two-Agent Task Completion benchmark, and explore the difficulty of defining rule-based agents from *TEACH* data.

In future, we will apply other ALFRED modeling approaches (Blukis et al. 2021; Kim et al. 2021; Zhang and Chai 2021; Suglia et al. 2021) to the EDH and TfD *Follower* model benchmarks. However, *TEACh* requires learning several different tasks, all of which are more complex than the simple tasks in ALFRED. Models enabling few shot generalization to new tasks will be critical for *TEACh Follower* agents. For *Commander* models, a starting point would be to train a *Speaker* model (Fried et al. 2018) on *TEACh* sessions.

7 Acknowledgements

We would like to thank Ron Rezac, Shui Hu, Lucy Hu, Hangjie Shi for their assistance with the data and code release. We would also like to thank Nicole Chartier, Savanna Stiff, Ana Sanchez, Ben Kelk, Joel Sachar, Govind That-tai, Gaurav Sukhatme, Joel Chengottusseriyil, Tony Bissell, Qiaozi Gao, Kaixiang Lin, Karthik Gopalakrishnan, Alexandros Papangelis, Yang Liu, Mahdi Namazifar, Behnam He-dayatnia, Di Jin, Seokhwan Kim and Nikko Strom for feedback and suggestions over the course of the project.

References

- Abramson, J.; Ahuja, A.; Brussee, A.; Carnevale, F.; Cassin, M.; Clark, S.; Dudzik, A.; Georgiev, P.; Guy, A.; Harley, T.; Hill, F.; Hung, A.; Kenton, Z.; Landon, J.; Lillicrap, T.; Mathewson, K.; Muldal, A.; Santoro, A.; Savinov, N.; Varma, V.; Wayne, G.; Wong, N.; Yan, C.; and Zhu, R. 2020. Imitating Interactive Intelligence. *arXiv*.
- Anderson, P.; Wu, Q.; Teney, D.; Bruce, J.; Johnson, M.; Sünderhauf, N.; Reid, I.; Gould, S.; and van den Hengel, A. 2018. Vision-and-Language Navigation: Interpreting Visually-Grounded Navigation Instructions in Real Environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Arumugam, D.; Karamchetti, S.; Gopalan, N.; Williams, E. C.; Rhee, M.; Wong, L. L.; and Tellex, S. 2018. Grounding Natural Language Instructions to Semantic Goal Representations for Abstraction and Generalization. *Autonomous Robots*.
- Bisk, Y.; Holtzman, A.; Thomason, J.; Andreas, J.; Bengio, Y.; Chai, J.; Lapata, M.; Lazaridou, A.; May, J.; Nisnevich, A.; Pinto, N.; and Turian, J. 2020. Experience Grounds Language. In *Empirical Methods in Natural Language Processing (EMNLP)*.
- Bisk, Y.; Shih, K.; Choi, Y.; and Marcus, D. 2018. Learning Interpretable Spatial Operations in a Rich 3D Blocks World. In *Proceedings of the Thirty Second AAAI Conference on Artificial Intelligence (AAAI)*, volume 32.
- Blukis, V.; Misra, D.; Knepper, R. A.; and Artzi, Y. 2018. Mapping Navigation Instructions to Continuous Control Actions with Position Visitation Prediction. In *Proceedings of the Conference on Robot Learning (CoRL)*.
- Blukis, V.; Paxton, C.; Fox, D.; Garg, A.; and Artzi, Y. 2021. A Persistent Spatial Semantic Representation for High-level Natural Language Instruction Execution. *arXiv preprint arXiv:2107.05612*.
- Chen, D.; and Mooney, R. 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. In *Proceedings of the Twenty Fifth AAAI Conference on Artificial Intelligence (AAAI)*, volume 25.
- Chen, H.; Suhr, A.; Misra, D.; Snavely, N.; and Artzi, Y. 2019. Touchdown: Natural Language Navigation and Spatial Reasoning in Visual Street Environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 12538–12547.
- Chi, T.-C.; Shen, M.; Eric, M.; Kim, S.; and Hakkani-Tür, D. 2020. Just Ask: An Interactive Learning Framework for Vision and Language Navigation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.
- Fried, D.; Hu, R.; Cirik, V.; Rohrbach, A.; Andreas, J.; Morency, L.-P.; Berg-Kirkpatrick, T.; Saenko, K.; Klein, D.; and Darrell, T. 2018. Speaker-Follower Models for Vision-and-Language Navigation. In *Neural Information Processing Systems (NeurIPS)*.
- Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL The Planning Domain Definition Language. *Yale Center for Computational Vision and Control*.
- Harnad, S. 1990. The Symbol Grounding Problem. *Physica D: Nonlinear Phenomena*, 42(1-3): 335–346.
- He, K.; Gkioxari, G.; Dollár, P.; and Girshick, R. B. 2017. Mask R-CNN. *International Conference on Computer Vision (ICCV)*.
- Kim, B.; Bhambri, S.; Singh, K. P.; Mottaghi, R.; and Choi, J. 2021. Agent with the Big Picture: Perceiving Surroundings for Interactive Instruction Following. In *Embodied AI Workshop CVPR*.
- Kim, H.; Zala, A.; Burri, G.; Tan, H.; and Bansal, M. 2020. ArraMon: A Joint Navigation-Assembly Instruction Interpretation Task in Dynamic Environments. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.
- Kollar, T.; Tellex, S.; Walter, M. R.; Huang, A.; Bachrach, A.; Hemachandra, S.; Brunskill, E.; Banerjee, A.; Roy, D.; Teller, S.; et al. 2013. Generalized Grounding Graphs: A Probabilistic Framework for Understanding Grounded Language. *Journal of Artificial Intelligence Research*.
- Kolve, E.; Mottaghi, R.; Han, W.; VanderBilt, E.; Weihs, L.; Herrasti, A.; Gordon, D.; Zhu, Y.; Gupta, A.; and Farhadi, A. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv preprint arXiv:1712.05474*.
- MacMahon, M.; Stankiewicz, B.; and Kuipers, B. 2006. Walk the Talk: Connecting Language, Knowledge, and Action in Route Instructions. In *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence (AAAI)*, volume 20.
- Matuszek, C.; Herbst, E.; Zettlemoyer, L.; and Fox, D. 2013. Learning to Parse Natural Language Commands to a Robot Control System. In *Experimental Robotics*, 403–415. Springer.
- Mei, H.; Bansal, M.; and Walter, M. 2016. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*, volume 30.

- Misra, D. K.; Bennett, A.; Blukis, V.; Niklasson, E.; Shatkhin, M.; and Artzi, Y. 2018. Mapping Instructions to Actions in 3D Environments with Visual Goal Prediction. In Riloff, E.; Chiang, D.; Hockenmaier, J.; and Tsujii, J., eds., *Empirical Methods in Natural Language Processing (EMNLP)*.
- Narayan-Chen, A.; Jayannavar, P.; and Hockenmaier, J. 2019. Collaborative Dialogue in Minecraft. In *Association for Computational Linguistics (ACL)*.
- Nguyen, K.; and Daumé III, H. 2019. Help, Anna! Visual Navigation with Natural Multimodal Assistance via Retrospective Curiosity-Encouraging Imitation Learning. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.
- Pashevich, A.; Schmid, C.; and Sun, C. 2021. Episodic Transformer for Vision-and-Language Navigation. *arXiv preprint arXiv:2105.06453*.
- Puig, X.; Ra, K.; Boben, M.; Li, J.; Wang, T.; Fidler, S.; and Torralba, A. 2018. VirtualHome: Simulating Household Activities via Programs. In *Computer Vision and Pattern Recognition (CVPR)*.
- Roman, H. R.; Bisk, Y.; Thomason, J.; Celikyilmaz, A.; and Gao, J. 2020. RMM: A Recursive Mental Model for Dialog Navigation. In *Findings of Empirical Methods in Natural Language Processing (EMNLP Findings)*.
- Shah, R.; Wild, C.; Wang, S. H.; Alex, N.; Houghton, B.; Guss, W.; Mohanty, S.; Kanervisto, A.; Milani, S.; Topin, N.; et al. 2021. The MineRL BASALT Competition on Learning from Human Feedback. *arXiv preprint arXiv:2107.01969*.
- Shridhar, M.; Thomason, J.; Gordon, D.; Bisk, Y.; Han, W.; Mottaghi, R.; Zettlemoyer, L.; and Fox, D. 2020. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *Computer Vision and Pattern Recognition (CVPR)*.
- Shrivastava, A.; Gopalakrishnan, K.; Liu, Y.; Piramuthu, R.; Tür, G.; Parikh, D.; and Hakkani-Tür, D. 2021. VISITRON: Visual Semantics-Aligned Interactively Trained Object-Navigator. *arXiv preprint arXiv:2105.11589*.
- Suglia, A.; Gao, Q.; Thomason, J.; Thattai, G.; and Sukhatme, G. 2021. Embodied BERT: A Transformer Model for Embodied, Language-guided Visual Task Completion. *arXiv preprint arXiv:2108.04927*.
- Suhr, A.; Yan, C.; Schluger, J.; Yu, S.; Khader, H.; Mouallem, M.; Zhang, I.; and Artzi, Y. 2019. Executing Instructions in Situated Collaborative Interactions. In *Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.
- Tellex, S.; Knepper, R. A.; Li, A.; Roy, N.; and Rus, D. 2016. Asking for Help Using Inverse Semantics. In *Robotics: Science and Systems Conference (RSS)*.
- Thomason, J.; Gordon, D.; and Bisk, Y. 2018. Shifting the Baseline: Single Modality Performance on Visual Navigation & QA. *arXiv preprint arXiv:1811.00613*.
- Thomason, J.; Murray, M.; Cakmak, M.; and Zettlemoyer, L. 2019. Vision-and-Dialog Navigation. In *Conference on Robot Learning (CoRL)*.
- Thomason, J.; Padmakumar, A.; Sinapov, J.; Walker, N.; Jiang, Y.; Yedidsion, H.; Hart, J.; Stone, P.; and Mooney, R. 2020. Jointly improving parsing and perception for natural language commands through human-robot dialog. *Journal of Artificial Intelligence Research*, 67: 327–374.
- Zhang, Y.; and Chai, J. 2021. Hierarchical Task Learning from Language Instructions with Unified Transformers and Self-Monitoring. *arXiv preprint arXiv:2106.03427*.
- Zhao, Y.; Lin, K.; Jia, Z.; Gao, Q.; Thattai, G.; Thomason, J.; and Sukhatme, G. S. 2021. LUMINOUS: Indoor Scene Generation for Embodied AI Challenges. *OpenReview Submission*.
- Zhu, W.; Hu, H.; Chen, J.; Deng, Z.; Jain, V.; Ie, E.; and Sha, F. 2020. BabyWalk: Going Farther in Vision-and-Language Navigation by Taking Baby Steps. In *Association for Computational Linguistics (ACL)*.
- We provide additional statistics about *TEACH* (§A), a summary of the data collection procedure for *TEACH* sessions (§B), additional details about EDH and TfD benchmark experiments (§C), additional details about the rule-based agents we implemented for the TTAC benchmark (§D), an explanation of the task definition language that guides the *TEACH* (§E), and representative examples and qualitative analysis of *TEACH* sessions (§F).

A Additional *TEACH* Statistics

During data collection, we aimed to obtain at least 50 sessions per task in the unseen validation and test splits. The exact number finally obtained varies due to the success rate of annotators in completing the tasks, and replicability issues related to non-determinism in the simulator. The final number of sessions per task per split is included in table 6.

	# Sessions		
	train	val-seen	val-unseen
Make Coffee	145	18	57
Water Plant	91	11	63
Make Plate Of Toast	90	11	52
Boil Potato	86	10	44
N Slices Of X In Y	164	20	56
N Cooked Slices	107	13	53
Prepare Salad	176	22	53
Prepare Sandwich	111	13	51
Clean All X	178	22	59
Put All X In One Y	167	20	50
Put All X On Y	184	23	50
Prepare Breakfast	162	20	53

Table 6: Number of sessions per task per split. Test split information is not currently included

We created unseen splits using the same floorplans as the split used in ALFRED to enable easy sharing of models between *TEACH* and ALFRED. In the future, we also plan to explore generating entirely new floorplans and layouts to expand the test scene distribution with controllable generation methods (Zhao et al. 2021).

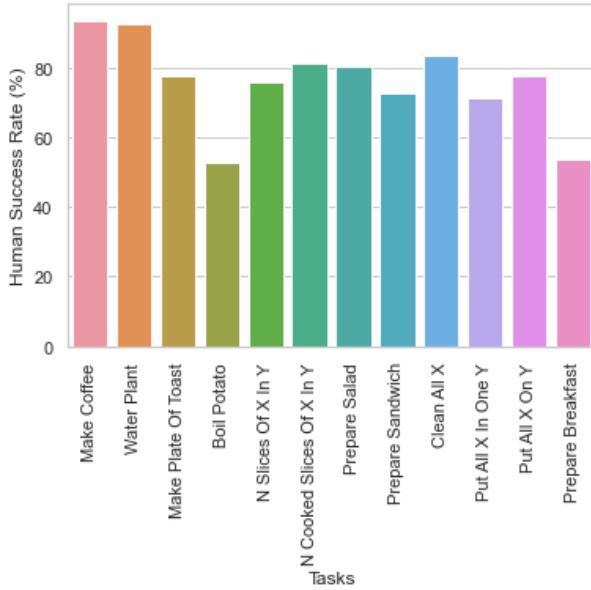


Figure 5: Human success rate for different tasks during data collection. Note that *TEACH* benchmarks only contains successful dialogue sessions, so human performance here is more a measure of how complex tasks were for annotators to complete against both coordination and simulator quirks.

The success rate of human annotators on different tasks when gathering data can be seen in Figure 5. We find that success rates are much higher for simpler tasks such as Make Coffee and Water Plant compared to more difficult tasks. The lowest success rates were obtained with the Prepare Breakfast task which had the most steps, and consequently the maximum number of possible issues annotators could run into, and the Boil Potato task which required some additional reasoning from annotators in order to use a smaller container such as a Cup to fill a Pot or Bowl with water, in which the the Potato could then be boiled. Common causes of failure across tasks included difficulties in placement of objects (an artifact of AI2-THOR), objects being in initial positions where they are difficult to see and hence manipulate, connection problems, and timeouts due to one annotator becoming unresponsive. Note that only sessions in which humans were successful are included in *TEACH* benchmarks.

We include vocabulary distributions of the 100 most common words in successful sessions in *TEACH*, as well as the 100 most common each of verbs, nouns and adjectives in Figure 7 (with POS tagging done using spaCy³). We also include the distribution of the frequency with with objects of different types are interacted with by the *Follower* (Figure 6). We can see that some objects get interacted with in many tasks, for example the counter is often used as a space to move things around, and faucets have to be interacted with frequently since many kitchen tasks involve the cleaning of utensils. Overall, since kitchen objects have more affor-

dances, many of our tasks are set in the kitchen. The Clean All X task can be done in both the kitchen and the bathroom, but only the Put All X On Y and Put All X In One Y tasks can be done in the bedroom and living room.

Our task definition language is extensible. We have defined Prepare Study Desk—analogous to Prepare Breakfast in scope and compositionality, for bedrooms and living rooms—together with some simpler tasks like Turn On/Off All Lights that better represent different room types. We plan to incorporate these *unseen tasks* into future iterations of the TATC benchmark.

To analyze language in *TEACH*, we include a distribution of the number of utterances per session in figure 8. We observe a significant number of games for a range of utterance lengths up to about 40 utterances per session, and the longest session has 139 utterances. The distribution of utterance lengths can be seen in Figure 9.

We include a distribution of the number of environment actions taken by the *Follower* in in 10. Our sessions are quite long, often involving several hundred actions per session.

B Annotator Instructions

Our annotator pool on Mechanical Turk was drawn using a private vendor service that provides high quality work in exchange for considerably higher pay than is normalized in the Mechanical Turk marketplace.

Annotators first completed a tutorial version of the task individually. In the tutorial, the annotator could see the tasks to be completed in the way the *Commander* does in the main annotation, but can also act in the environment as the *Follower* does. They are then provided step by step instructions to control the *Follower* to complete two tasks - making coffee and cooking a slice of a potato. Only annotators who successfully completed the tasks in the tutorial were allowed to participate in the main collection.

Annotators were primed with the following description of the task:

2-Player Game:

Now when you log into the game, you will be one of two players, the User/Commander or the Robot/Driver.

The User will have the progress check button but will not have the buttons to pick up / place objects or do other things with them.

The Robot cannot see the progress check button but has the buttons to pick up / place objects or do other things with them.

There is a chat box on the bottom right of the screen for the User and Robot to chat with each other. Both players have to work together to complete the task.

When you open the game, see if you have a Progress Check button on the top left.

Robot:

- If you don't have a Progress Check button, you are the Robot.*

³<https://pypi.org/project/spacy/>

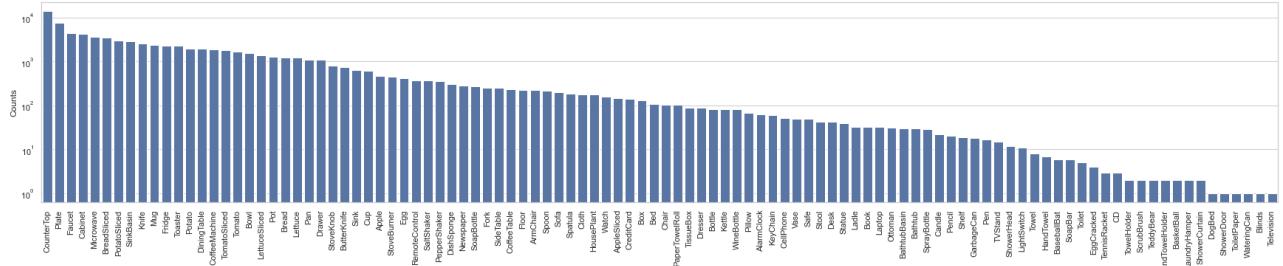


Figure 6: Object Distribution: Frequency with which objects are interacted with by the *Follower* across all sessions. Log scale.

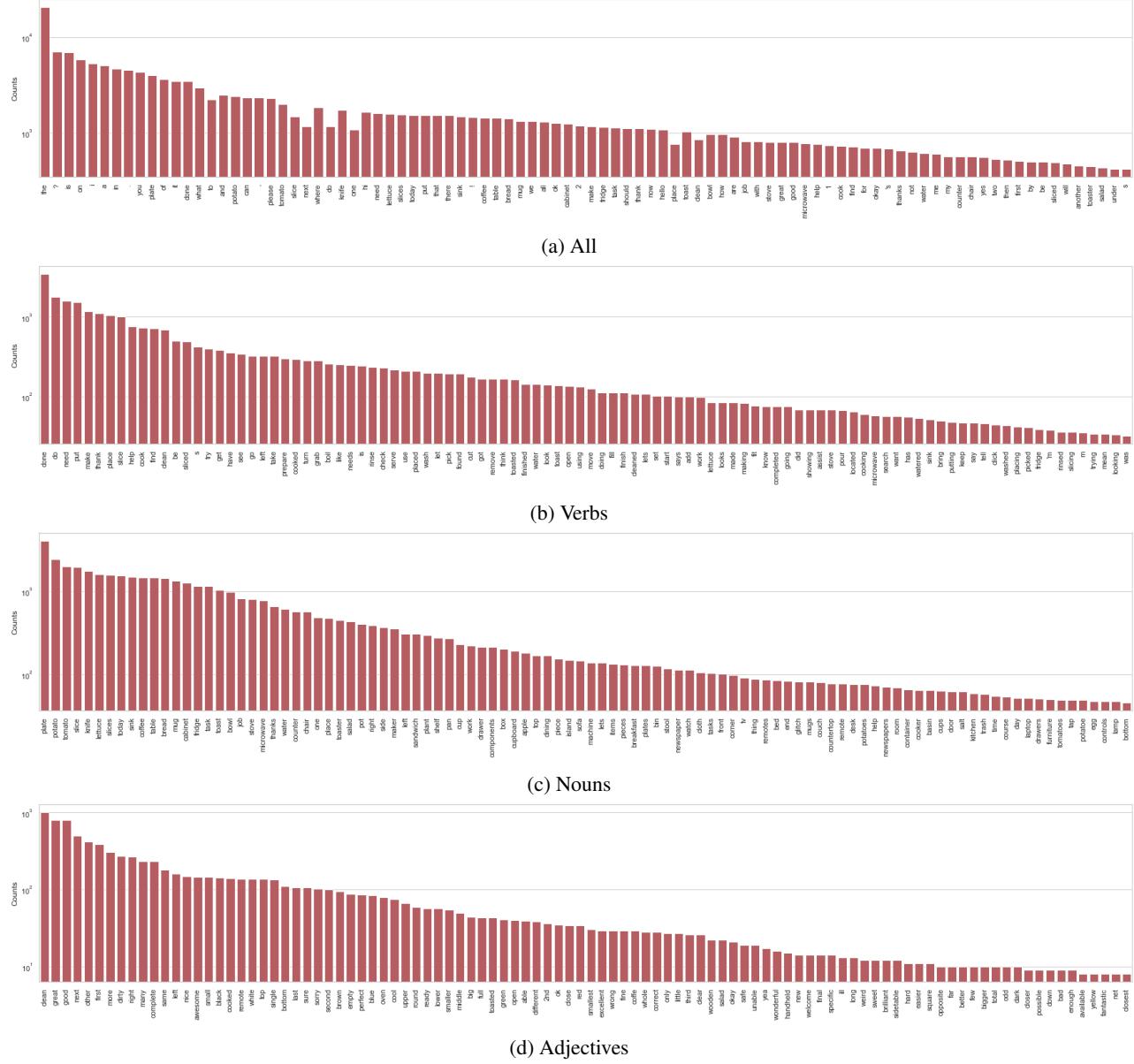


Figure 7: Vocabulary Distributions: Frequency distributions of the 100 most common words, and 100 most common each of verbs, nouns and adjectives. Best viewed zoomed in. Log scale.

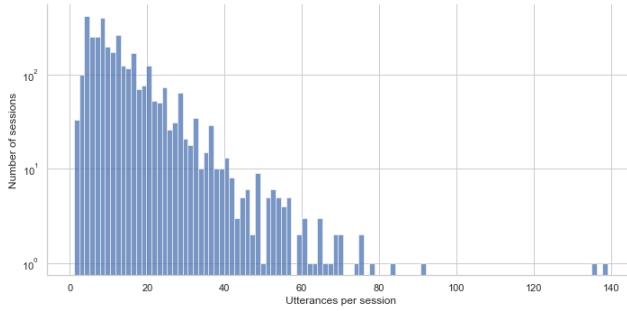


Figure 8: Distribution of dialogue lengths in terms of number of utterances per session. Log scale.

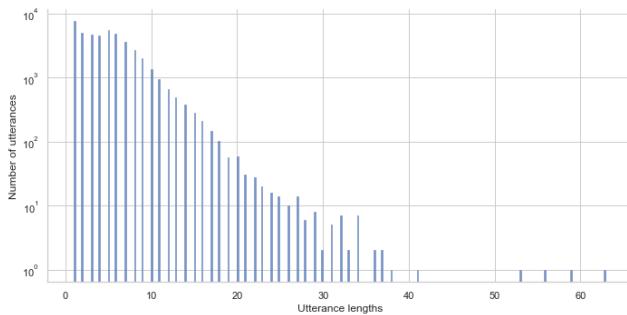


Figure 9: Distribution of utterance lengths in terms of number of tokens across sessions. Log scale.

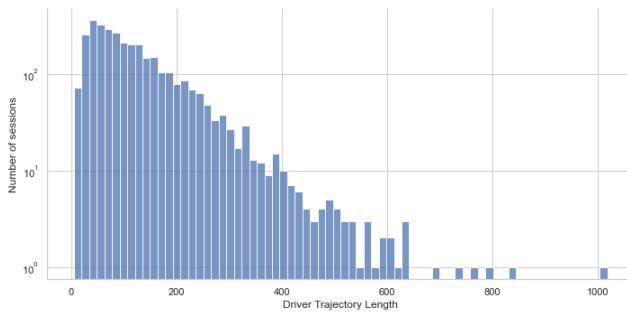


Figure 10: Distribution of *Follower* action trajectory lengths across sessions. Log scale.

- You need to pretend you are a robot who is completing tasks in this house for your user.
- Enter in the chat box “What should I do today?” so that your User knows you’re there.
- Once the User tells you what to do, try to complete the task. You can use the chat to ask them questions, ask them to search for objects, or check whether steps have been completed. When the task is completed, your partner has to hit Finish to take both of you to the survey where you will rate your partner. When you submit the survey, you will get the code to enter in the HIT.

User:

- When you open the game, if you have the Progress Check button, you are the User.
- You need to pretend that the house in the game is your house and you are telling your Robot to complete tasks for you.
- Remember that “the robot” is another worker like you pretending to be a robot to be respectful when talking to them.
- To get started, click on the Progress Check button to see what tasks you have to do. Each HIT will have a slightly different task. Use the chat box to tell your Robot what task to do.
- The Robot can ask you questions as they do the task. You can search for objects to help them and confirm whether they have successfully completed the task.
- When the Progress Check button says that the task is done, hit Finish. That will take both you and your partner to the survey where you will rate each other. When you submit the survey, you will get the code to enter in the HIT.

Robot/ Driver Do’s and Don’ts

When you finish a step, use the chat box to tell the User and ask for the next step. E.g.: “I toasted the bread. What should I do next?”

When you need an object, ask the User to search for it first. If they cannot find it you will have to search for it yourself. Remember to open drawers and cabinets while searching. For example:

User: We need to make toast

Robot: Can you help me find the bread?

User: The bread is inside the fridge.

Robot: can directly go to the fridge and get the bread.

Robot: I also need a knife. Do you know where that is?

User: I don’t know. Can you search for it?

Robot: should search for knife.

User/ Commander Do’s and Don’ts

Use the search function in the Progress Checker to help the Driver find objects. If you don’t understand what it says you can say you don’t know. For example:

User: We need to make toast

Robot: Can you help me find the bread?

User: The bread is inside the fridge.

Robot: I also need a knife. Do you know where that is?

User: I don’t know. Can you search for it?

If a task has many steps. Don’t tell all of them as once. Wait for your Robot to finish a step before giving them the next step.

Good example:

User: Today we need to put all the forks in the sink. You can start with the one inside the microwave.

Robot: I got the fork from the microwave. Heading to the sink now.

Robot: I placed that fork in the sink. Are there more?
User: There is another fork on a plate to the left of the stove.
Robot: Found it. I will take it to the sink now.
Robot: That's in the sink. What should I do next?
User: I think we're done.

Bad example (don't do this):
User: Today we need to put all the forks in the sink. There is one inside the microwave and another on a plate to the left of the stove.

Try to help the Robot solve problems. For example:

User: Today we need to put all the forks in the sink. You can start with the one inside the microwave.
Robot: I found the fork but I am not able to place it in the sink.
User: Is the water running?
Robot: Yes it is
User: Try turning it off first
Robot: Thanks I tried that but I am still not able to place the fork in the sink
User: What else is there in the sink?
Robot: There is a plate and a few cups
User: Try removing something from the sink first.

After reading priming instructions, workers gathered 2-player sessions where they were randomly assigned one of the roles of *Commander* or *Follower*.

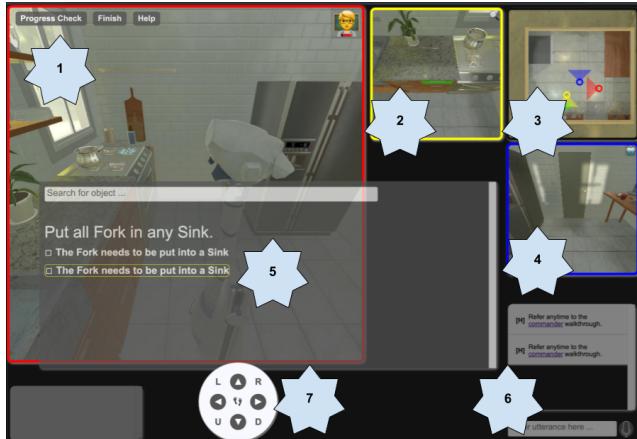


Figure 11: Interface seen by the crowdworker playing the *Commander*. Numbered stars are added for the purpose of explanation and correspond to the descriptions in this section.

The interface seen by the *Commander* is shown in Figure 11. The components of this interface are:

1. Main panel: The *Commander* is allowed to move around in the environment. The Main Panel shows their egocentric view.
2. Target object panel: If the *Commander* clicks on an instruction or searches for an object, a visual indication of

the location of the object is shown here. For example in Figure 11, one of the instruction steps is clicked and the target object panel is highlighting a drawer next to the sink. This indicates that a fork is present in that drawer which needs to be used in that instruction step.

3. Top down view: A top down view of the room the *Follower* is in. The blue circle is the *Follower*'s current position, and the translucent blue triangle represents the view cone of the *Follower*. The red circle and cone correspond to the position of the *Commander*.
4. *Follower* view panel: This shows the current egocentric view of the *Follower*.
5. Progress Check Menu: This is shown when the Progress Check button has been clicked. It shows a list of steps to be completed.
6. Chat window to send messages to the *Follower*.
7. The navigation control panel used by the *Commander* to move around. The *Commander* can move through walls and objects but cannot interact with objects.



Figure 12: Interface seen by the crowdworker playing the *Follower*. Numbered stars are added for the purpose of explanation and correspond to the descriptions in this section.

The interface used by the *Follower* is shown in Figure 12. The components of this interface are:

1. Main panel: This shows the egocentric view of the *Follower*.
2. Top down view: A top down view of the room the *Follower* is in. The blue circle is the *Follower*'s current position, and the translucent blue triangle represents the view cone of the *Follower*: the direction in the room the *Follower* is currently facing.
3. Chat window to chat with the *Commander*.
4. Navigation control panel to move the *Follower* and change its orientation.
5. Object interaction control panel to interact with objects.
6. Console log: This shows messages from the simulator to give annotators hints for what is going wrong when an

action fails. For example, if the *Follower* tries to place an object but the placement fails, a simulator message might let the annotator know that the receptacle they’re trying to place into is too full (e.g. they may need to clear out the sink before placing a new plate into it).

A session is ended by the *Commander* clicking the “Finish” button adjacent to the “Progress Check” button on their interface, then confirming that they would like to end the game. The “Progress Check” display changes to let the *Commander* know when the task is completed, prompting the *Commander* to end the game. *TEACh* contains only successful sessions where the task was completed.

C Additional EDH and TfD Experiment Details

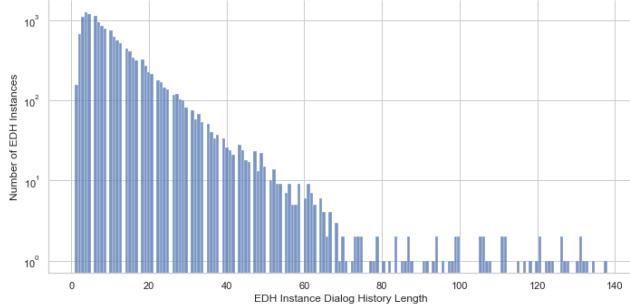


Figure 13: Distribution of dialog history length across EDH instances. Log scale. Note that EDH “double counts” many histories, skewing to a much longer, compounded average than full *TEACh* sessions.

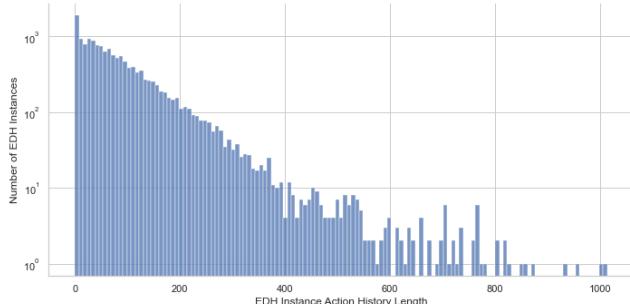


Figure 14: Distribution of action history lengths across EDH instances.

We include a distribution of EDH dialogue and action history lengths in Figures 13 and 14, respectively. While the average action history length for EDH instances is 86.97 actions, a significant number of EDH instances have an action history of over 200 actions.

We include the distribution of the number of all and object interaction actions to be predicted in Figures 15 and 16, respectively. While on average, a model for EDH needs to predict 19.76 actions of which on average 4.74 actions are object interaction actions, EDH instances may require as many

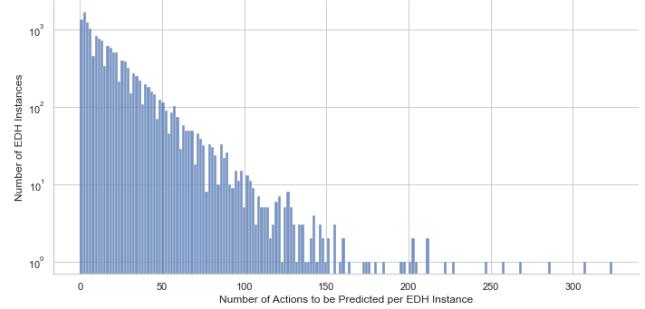


Figure 15: Distribution of number of actions to be predicted per instance across EDH instances.

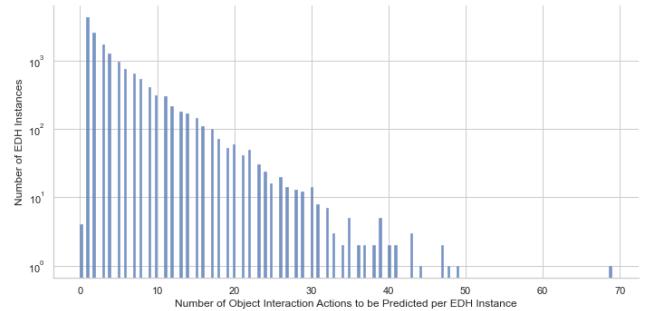


Figure 16: Distribution of number of object interaction actions to be predicted per instance across EDH instances.

as 324 actions to be predicted, with many instances requiring over 50 actions to be predicted. A significant number of EDH instances also require 10-20 object interactions to be predicted in order to successfully complete the instance.

Moeling EDH and TfD with E.T. For our human demonstrations, we showed image observations of size 900 x 900, and hence obtained images of the same size during replay for modeling. Images are resized to 224 x 244 for the ResNet-50 backbone of the main E.T. transformer and to 300 x 300 for the MaskRCNN model. We use the pretrained visual encoder based on Faster R-CNN and mask generator based on Mask R-CNN from E.T., where they are trained on 325K frames of expert demonstrations from the ALFRED train fold (which matches our train fold in terms of floorplans and objects visible). Additionally, as in E.T., we do not update the visual encoder or mask generator during model training for any of our tasks. The E.T. visual encoder average-pools ResNet features 4 times and adds a dropout of 0.3 to obtain feature maps of 512 x 7 x 7. These are then fed into 2 convolutional layers of with 256 and 64 filters of size 1 x 1 respectively, and mapped using a fully connected layer to size 768. Additionally, as in E.T., we use transformer encoders each with 2 blocks, 12 self attention heads, hidden size of 768, and dropout of 0.1. We also follow E.T in using the AdamW optimizer with 0.33 weight decay with a learning rate of $1e - 4$ for the first 10 epochs and $1e - 5$ for the last 10 epochs. We trained all models for 20 epochs with a batch size of 3, and report results from the final epoch.

We reused all hyperparameters except the batch size from the released E.T. model without further tuning, and used the largest batch size that could fit in a single GPU of a p3.16x EC2 instance. One change we made was that E.T. samples 30000 instances per epoch from the pool of training examples with replacement. We replace sampling with rotation permutations of our training dataset per epoch, ensuring that every train example is seen exactly once in our dataset.

E.T. uses 2 cross entropy losses: one over actions and one over object categories during object interaction actions. We use an equal weight for these two losses. E.T. additionally uses auxiliary losses for overall and subgoal progress based on ALFRED but we do not use these as they are tailored for ALFRED and we do not have equivalent subgoal progress signals in *TEACH* benchmarks.

Overall, our episode replay phase to generate image observations for training takes about 6 hours using 50 threads on a single p3.16x EC2 instance. The preprocessing phase of E.T. involving extracting image features and vectorizing language inputs takes about 7 hours using all GPUs of a single p3.16x EC2 instance. Training a model for 20 epochs takes about 5 hours using a single GPU of a single p3.16x EC2 instance. Evaluation of EDH instances takes about 6 hours using 2 GPUs of a single p3.16x EC2 instance for seen splits, and about 14 hours using 3 GPUs of a single p3.16x EC2 instance for unseen splits, but we did see a considerable amount of variation across runs. Tfd evaluation takes about 2 hours using 2 GPUs of a single p3.16x EC2 instance for seen splits and 3 hours using 2 GPUs of a single p3.16x EC2 instance for unseen splits. We believe it should be possible to improve evaluation runtimes through optimizations to our wrapper over AI2-THOR.

We include a breakdown of EDH success rates across tasks in table 7. Note that the task referenced here is the task for the original gameplay session the EDH instance is created from. Since our task definitions are hierarchical, it is possible for some EDH instances from different tasks to involve the same steps to be predicted. For example, the Make Coffee task is a subtask of the Prepare Breakfast task, so it is possible for there to be EDH instances from sessions of the Prepare Breakfast task where the agent is only required to make coffee—the same as it would do in the Make Coffee task.

D Rule-Based Agents for TATC

As mentioned in section 5.2, we engineered rule-based *Commander* and *Follower* agents as an attempt to solve the Two-Agent Task Completion benchmark.

Rule-based Follower maintains a queue of actions that it needs to execute. Whenever this queue gets empty, it utters “*What should I do next?*”. *Commander* in the next turn detects this utterance and executes a Progress Check action to generate a templated language instruction. The templated instruction consists of space-separated low-level actions like “*Forward Forward TurnRight LookUp Pickup Mug at 0.57 0.25*”. This instruction can be split up by the *Follower* into action tokens and each token has a one-to-one correspondence to an action in the action space of *Follower*. For interaction action like “*Pickup Mug at 0.57*

0.25”, “*Mug*” represents the object to be interacted with, and ‘*0.57 0.25*’ represents the normalized coordinates in the egocentric view of *Follower* accepted by the *TEACH* API as a click position.

Rule-based Commander executes a Progress Check action whenever it is asked to provide supervision. Listing 1 shows the Progress Check output at the start of MAKE PLATE OF TOAST task. **problem_keys** in the Progress Check output contains info about all objectives that need to be completed to solve the task. **property_name** of a problem key defines the type of the problem key to be solved. For the 11 tasks we consider, there are 7 problem keys that can be solved: parentReceptacle, isDirty, isFilledWithLiquid, isFilledWithCoffee, isBoiled, isCooked and Slice. **DesiredValue** denotes the state that the object should be in. For each of the problem keys, we can engineer a hand-crafted logic to solve it. Consequently, the scope of what rule-based agents can accomplish is limited by the engineering hours needed to identify and hand-write solutions to these problem keys in a modular planning fashion.

For parentReceptacle problem key, an object needs to be placed on/inside another object. Algorithm 1 shows the hand-crafted logic for parentReceptacle problem key. Whenever *Commander* is asked for supervision, it checks the **property_name** of problem key and if it is parentReceptacle, it will call the parentReceptacle() function. The **current_step** in the function keeps track of the supervision that needs to be provided based on the progress for the current problem key. For **current_step** = **navigation_1**, it will call a function Navigate(ObjectID) which runs a shortest path planner from the current pose of the *Follower* agent to get the low-level navigation instruction to reach ObjectID. Similar logic can be written for other problem keys. We will release the code containing handcrafted logic for all problem keys covered by our rule-based agents.

E Task Definition Language

We define a Task Definition Language to define household tasks in terms of object properties that need to be satisfied in the environment for the task to be considered successful. This Task Definition Language is based on a PDDL-like syntax (Ghallab et al. 1998). A sample task can be seen in Listing 2, which defines the MAKE A PLATE OF TOAST task in *TEACH*.

A task is specified in terms of **components** and **relations**. A **component** is specified using a set of conditions to be satisfied for the task to be considered complete. As seen in the above example, a **component** can be specified by referencing another task. In MAKE A PLATE OF TOAST, the **component** toast is described by referencing another task, Toast (Listing 3), and the **component** plate is described by referencing another task, Clean X (Listing 4), with parameter value Plate. In task definitions, **relations** are used to describe relationships between **components** that must be satisfied for the task to be considered complete. In MAKE A PLATE OF TOAST, the relation specifies that one object satisfying the conditions of

Task	Per task EDH success rates on the valid-seen split								
	Rand	Lang	Vision	E.T.	+H	+A	+H+A	+S	+H+S
Make Coffee	2.17	0.00	17.39	15.22	13.04	8.70	13.04	13.04	13.04
Water Plant	0.00	0.00	0.00	10.53	10.53	15.79	10.53	21.05	5.26
Make Plate Of Toast	0.00	2.38	0.00	7.14	4.76	4.76	2.38	7.14	4.76
Boil Potato	0.00	0.00	0.00	17.39	8.70	13.04	4.35	13.04	13.04
N Slices Of X In Y	2.08	1.04	7.29	14.58	8.33	10.42	8.33	12.50	8.33
N Cooked Slices Of X In Y	1.15	4.60	4.60	8.05	4.60	6.90	5.75	10.34	8.05
Clean All X	0.00	2.00	8.00	10.00	10.00	4.00	12.00	6.00	6.00
Put All X In One Y	0.00	0.00	0.00	12.82	5.13	2.56	7.69	7.69	2.56
Put All X On Y	0.00	0.00	6.25	4.17	10.42	8.33	8.33	10.42	6.25
Prepare Salad	0.66	0.66	4.64	3.97	4.64	6.62	3.97	5.96	6.62
Prepare Sandwich	1.02	1.02	4.08	6.12	5.10	5.10	9.18	6.12	8.16
Prepare Breakfast	0.00	0.56	4.52	4.52	2.82	4.52	4.52	3.39	4.52
Per task EDH success rates on the valid-unseen split									
Make Coffee	2.00	0.00	4.00	8.00	9.00	7.00	8.00	7.00	10.00
Water Plant	0.00	0.00	2.30	2.30	5.75	2.30	1.15	2.30	3.45
Make Plate Of Toast	0.00	0.00	1.17	1.17	2.34	2.34	1.75	2.92	2.92
Boil Potato	0.00	0.00	0.94	6.60	6.60	1.89	4.72	3.77	7.55
N Slices Of X In Y	0.69	0.69	4.17	9.72	4.17	9.03	6.25	9.03	6.94
N Cooked Slices Of X In Y	0.00	0.53	3.19	4.26	4.26	6.38	2.13	4.79	4.26
Clean All X	0.91	0.00	4.55	3.64	9.09	3.64	6.36	4.55	2.73
Put All X In One Y	1.02	0.00	2.04	6.12	1.02	4.08	3.06	3.06	3.06
Put All X On Y	0.00	2.94	0.00	0.00	5.88	4.41	4.41	0.00	4.41
Prepare Salad	0.75	0.38	1.13	6.04	2.26	4.53	1.51	3.40	3.02
Prepare Sandwich	0.40	0.40	2.78	5.16	3.57	3.97	3.97	4.37	3.17
Prepare Breakfast	0.27	0.82	4.37	5.19	6.28	3.28	5.19	6.56	4.37
Per task EDH success rates on the test-seen split									
Make Coffee	0.00	0.00	4.17	16.67	14.58	10.42	10.42	12.50	12.50
Water Plant	0.00	0.00	10.00	15.00	5.00	15.00	15.00	15.00	15.00
Make Plate Of Toast	0.00	0.00	8.33	6.25	8.33	6.25	6.25	4.17	6.25
Boil Potato	0.00	0.00	0.00	4.17	4.17	4.17	0.00	0.00	4.17
N Slices Of X In Y	1.43	1.43	4.29	4.29	7.14	5.71	8.57	5.71	4.29
N Cooked Slices Of X In Y	0.00	0.00	2.67	5.33	8.00	9.33	4.00	5.33	2.67
Clean All X	4.05	0.00	6.76	8.11	8.11	6.76	16.22	6.76	13.51
Put All X In One Y	2.00	6.00	0.00	4.00	6.00	4.00	8.00	4.00	6.00
Put All X On Y	0.00	0.00	2.13	2.13	4.26	6.38	4.26	0.00	6.38
Prepare Salad	1.38	0.00	4.83	6.90	4.83	6.90	4.14	6.90	6.21
Prepare Sandwich	0.00	0.00	1.47	0.00	2.94	2.94	2.94	1.47	2.94
Prepare Breakfast	0.00	1.75	4.68	4.68	6.43	6.43	8.77	4.09	6.43
Per task EDH success rates on the test-unseen split									
Make Coffee	0.69	2.08	1.39	4.86	9.72	5.56	9.72	5.56	10.42
Water Plant	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Make Plate Of Toast	0.37	0.75	4.10	5.22	3.36	4.10	5.97	4.10	5.22
Boil Potato	0.53	0.53	4.28	3.21	2.67	6.95	3.74	6.42	3.21
N Slices Of X In Y	0.97	0.49	3.88	4.85	6.80	3.88	4.85	6.31	5.34
N Cooked Slices Of X In Y	1.87	1.49	1.87	7.84	3.73	6.34	4.85	8.58	4.48
Clean All X	0.51	1.54	3.08	10.26	5.64	9.74	5.13	8.21	7.18
Put All X In One Y	0.00	0.00	3.00	3.00	5.00	5.00	2.00	5.00	2.00
Put All X On Y	0.00	0.00	0.00	5.22	5.22	5.22	2.24	2.24	2.24
Prepare Salad	1.06	1.06	4.22	7.12	4.22	7.12	2.37	9.23	4.75
Prepare Sandwich	0.00	1.01	3.36	6.04	4.36	4.36	3.02	5.03	2.35
Prepare Breakfast	0.87	0.22	2.17	4.99	4.99	4.99	5.21	4.77	3.69

Table 7: Success rates on EDH benchmark divided by instance source task. All values are percentages.

component plate must be the container of (captured by AI2-THOR property `parentReceptacles`) one object satisfying the conditions of **component** toast.

The task CLEAN X is an example of a parameterized task. The parameter #0 is set to Plate when this task is refer-

enced as a part of the more complex task PLATE OF TOAST (Listing 2). In CLEAN X, the parameter is intended to be a custom object class (such as Plate or Utensil) which have been pre-defined. Parameters can also be used to specify in determiners or even free text to go into a natural lan-

Algorithm 1: Handcrafted logic for parentReceptacle

Input: step

Output: instruction, step

parentReceptacle(step=“navigation_1”):

```
1: ObjectID = get_object_id()
2: ParentID = get_parent_id() % None if ObjectID not
   present inside an object
3: if step = “navigation_1” then
4:   instruction = Navigate(ObjectID)
5:   step = “interaction_1_1”
6: else if step = “interaction_1_1” then
7:   if ObjectID inside ParentObject then
8:     instruction = “ToggleOff ParentObject”
9:     instruction += “Open ParentObject”
10:    step = “interaction_1_2”
11:  else
12:    instruction = “Pickup ObjectID”
13:    step = “step_completed”
14:  end if
15: else if step = “interaction_1_2” then
16:   instruction = “Pickup ObjectID”
17:   step = “interaction_1_3”
18: else if step = “interaction_1_3” then
19:   instruction = “Close ParentObject”
20:   step = “step_completed”
21: end if
22: return instruction, step
```

guage description, for example “Put all Fork *in* any Sink” versus “Put all cups *on* any Table.” When we check for task completion, parameters can be thought of as macros—we first do a text replacement of the parameter value wherever the parameter occurs in the task definition, and then the task definition is processed as if it does not have parameters. The use of parameters allows easy creation of different variants of a task with low manual effort, thus allowing us to create a more diverse dataset.

More formally, a TASK is defined by

- **task_id** - A unique ID for the task
- **task_name** - A unique name for the task used to reference it in other tasks
- **task_nparams** - Number of parameters required by the task
- **desc** - Natural language prompt describing the task to provide to a *Commander*, (be it human or agent model).
- **components** - Dictionary specifying sets of conditions to be satisfied by objects of different types. This is used to specify both precondition objects required to complete the task such as knife if slicing is required or a sink if cleaning is required, as well as objects that need to be converted to the correct state as part of the task such as *toast* in Listing 3. A component can also be described using another Task, such as CLEAN X being used to define the target receptacle on which toast should sit in

MAKE A PLATE OF TOAST.

- **relations** - List of conditions that relate one set of objects to another. Currently the only relationship used in our task definitions is parentReceptacles which checks if one object is the container for other objects. However, pairwise operators could also be used to capture other spatial relations or time of completion of components.
- **task_anchor_object** - This is either the key of a **component** or null. This is used to identify the specific object in the simulation environment whose properties would be checked when a **component** specified by this **Task** is used in a **relation**. For example, the MAKE A PLATE OF TOAST TASK (Listing 2) contains a **relation** that says that its *toast* **component** should be contained in its *plate* **component**. Looking at the task definition for task *Toast* (Listing 3), we find that its **task_anchor_object** is its **component** *toast* (and not its **component** *knife*) which will resolve to an object of type *BreadSliced*. Looking at the task definition for CLEAN X (Listing 4), we find that its **task_anchor_object** is the **component** whose key is set to the value of parameter #0 which will be resolved to an object of type #0. Since MAKE A PLATE OF TOAST passes the parameter value *Plate* to CLEAN X, the *plate* **component** in MAKE A PLATE OF TOAST will resolve to an object of type *Plate*. Thus overall the relation should check for an object of type *BreadSliced* (which also satisfies other conditions specified in **component** *toast*) to be placed on the object of type *Plate* (which also satisfies other conditions specified in **component** *plate*). Note that if the **task_anchor_object** for a Task is null, it cannot be used in **relations** in other Tasks (but can still be a **component**).

A **component** can be of one of two types:

- Atomic **component** - A **component** that is specified in terms of base simulator properties, for example all **components** of Tasks TOAST and CLEAN X.
- Task **component** - A **component** that is specified in terms of another Task, for example the components in MAKKE A PLATE OF TOAST).

Atomic **components** are specified using the following keys:

- **conditions** - Set of **property** : **desired_value** pairs for this **component** to be considered satisfied. For example the conditions for the *toast* **component** in TOAST look for an object of type *BreadSliced* whose property *isCooked* has been set to 1.
- **condition_failure_descs** - For properties in conditions that correspond to changes that have to happen by the annotator taking an action, this specifies the description to be provided to the annotator if the property is currently not set to the desired value. For example, in **component** *toast* the value for **condition_failure_descs** specifies that if there is

Listing 1: Sample Progress Check response for MAKE A PLATE OF TOAST

```
{
  "task_desc": "Make a plate of toast.",
  "success": 0,
  "subgoals": [
    {
      "representative_obj_id": "Bread/-00.58| 00.27|-01.27",
      "step_successes": [0],
      "success": 0,
      "description": "Make a slice of toast.",
      "steps": [
        {
          "success": 0,
          "objectId": "Bread/-00.58| 00.27|-01.27",
          "objectType": "Bread",
          "desc": "The bread needs to be sliced using a knife.", },
        {
          "success": 0,
          "objectId": "Bread/-00.58| 00.27|-01.27",
          "objectType": "Bread",
          "desc": "The bread needs to be toasted.", } ],
      "problem_keys": {
        "Bread|-00.58| 00.27|-01.27": [
          {
            "objectType": "Bread",
            "determiner": "a",
            "property_name": "objectType",
            "desired_property_value": "BreadSliced" },
          {
            "objectType": "Bread",
            "determiner": "a",
            "property_name": "isCooked",
            "desired_property_value": 1 } ] } },
    {
      "representative_obj_id": "Plate/-01.18| 00.21|-01.27",
      "step_successes": [1, 0],
      "success": 0,
      "description": "Clean a Plate.",
      "steps": [
        {
          "success": 0,
          "objectId": "Plate/-01.18| 00.21|-01.27",
          "objectType": "Plate",
          "desc": "The Plate is dirty. Rinse with water." }],
      "problem_keys": {
        "Plate|-01.18| 00.21|-01.27": [
          {
            "objectType": "Plate",
            "determiner": "a",
            "property_name": "isDirty",
            "desired_property_value": 0 } ] } ] }
  ]
}
```

no sliced bread in the scene, we should send the message “The bread needs to be sliced using a knife” and if there is no toasted bread slice in the scene we should send the message “The bread needs to be toasted”.

- **determiner** - This is used to specify how many object instances should satisfy this set of conditions. The possible values are **a**, **all** or a positive integer. For example in the Task Toast, the toast **component** has **determiner** **a** so we would say this component is satisfied if there is any slice of toasted bread in the scene. Instead if the **determiner** was 2, we would only say

that the **component** is satisfied if there are at least 2 slices of toasted bread in the scene. If it was **all**, we would say that the **component** is satisfied if all slices of bread present in the scene are toasted.

- **primary_condition** - This is used to find candidate objects that an annotator needs to modify to satisfy this **component**. It is usually an object type or class.
- **instance_shareable** - This is a parameter used to handle how numbers cascade across hierarchies. In the MAKE A PLATE OF TOAST, the **determiner** for component toast is **a**. Suppose instead that this was 2.

Listing 2: Sample task: Make a Plate of Toast

```
{
  "task_id": 106,
  "task_name": "Plate Of Toast",
  "task_nparams": 0,
  "task_anchor_object": "plate",
  "desc": "Make a plate of toast.",
  "components": {
    "toast": {
      "determiner": "a",
      "task_name": "Toast",
      "task_params": []
    },
    "plate": {
      "determiner": "a",
      "task_name": "Clean X",
      "task_params": ["Plate"]
    }
  },
  "relations": [
    {
      "property": "parentReceptacles",
      "tail_entity_list": ["plate"],
      "tail_determiner_list": ["the"],
      "head_entity_list": ["toast"],
      "head_determiner_list": ["a"],
      "failure_desc": "The toast needs to be on a clean plate."
    }
  ]
}
```

By default we would multiply the **determiners** of all **components** of TOAST (except all) by 2 (treating a as 1). So to check if the TOAST-2 is satisfied we would check both if there are 2 slices of toast and if there are 2 knives. But the knife has only been specified as a precondition and we do not actually need 2 knives in TOAST-2. This exception is captured by the property **instance.shareable**. The knife **component** has **instance.shareable** = true so regardless of the **determiner** associated with TOAST, we would only check for one knife, but the **component** toast has **instance.shareable** = false so we would require n slices of toast in TOAST-2.

Task relations are specified by:

- **property** - The property being checked (currently we only have support for `parentReceptacles`)
- **head_entity_list**, **tail_entity_list** - While exactly which object is the head and which object is the tail is arbitrary and would be decided by implementation used to check a property, we assume that if we examine the property value of the head entities, the tail entities would be specified in them. Currently these are imple-

Listing 3: Sample task: Make Toast

```
{
  "task_id": 101,
  "task_name": "Toast",
  "task_nparams": 0,
  "task_anchor_object": "toast",
  "desc": "Make a slice of toast.",
  "components": {
    "toast": {
      "determiner": "a",
      "primary_condition": "objectType",
      "instance_shareable": false,
      "conditions": {
        "objectType": "BreadSliced",
        "isCooked": 1
      },
      "condition_failure_descs": {
        "objectType": "The bread needs to be sliced using a knife.",
        "isCooked": "The bread needs to be toasted."
      }
    },
    "knife": {
      "determiner": "a",
      "primary_condition": "objectType",
      "instance_shareable": true,
      "conditions": {
        "objectType": "Knife"
      },
      "condition_failure_descs": {}
    }
  },
  "relations": []
}
```

mented as lists to handle the very specific case where we want to define that multiple objects need to be placed in a single container (e.g.: multiple sandwich components in a plate). The entities are specified using the **component** keys and we recursively check **task.anchor_object** of Task components to find the exact object to be used when checking the relation.

- **head_determiner_list** - A list of the same length as **head_entity_list** where each entry can take values a, all, or a number and specify how many objects matching conditions specified by the respective **component** are involved in this relation.
- **tail_determiner_list** - A list of the same length as **tail_entity_list** where each entry can take values a or the. To illustrate the difference, compare the Tasks PUT ALL X ON Y (Listing 5) and PUT ALL X IN ONE Y (Listing 6). Suppose there are two objects of

Listing 4: Sample task: Clean X

```
{
  "task_id": 103,
  "task_name": "Clean X",
  "task_nparams": 1,
  "task_anchor_object": "#0",
  "desc": "Clean a #0.",
  "components": {
    "#0": {
      "determiner": "a",
      "primary_condition": "objectClass",
      "instance_shareable": false,
      "conditions": {
        "objectClass": "#0",
        "isDirty": 0
      },
      "condition_failure_descs": {
        "isDirty": "The #0 is dirty.  
Rinse with water."
      }
    },
    "sink": {
      "determiner": "a",
      "primary_condition": "objectType",
      "instance_shareable": true,
      "conditions": {
        "objectType": "Sink",
        "receptacle": 1
      },
      "condition_failure_descs": {}
    }
  },
  "relations": []
}
```

type X (x_1 and x_2) and two objects of type Y (y_1 and y_2) in the scene. If x_1 is placed in/on y_1 and x_2 is placed in/on y_2 , this would satisfy the TASK PUT ALL X ON Y (because each object of type X is on a object of type Y) but does not satisfy PUT ALL X IN ONE Y (because there is no single object of type Y such that x_1 and x_2 are in the object of type Y).

- **failure_desc** - The message to be shown to an annotator if some action needs to be taken to make this relation satisfied.

Checking task completion: The task definition specifies all conditions that need to be satisfied by objects in the scene for a Task to be considered satisfied. To check if a Task is satisfied, first, for each **component**, we check if as many instances, specified by the **determiner** of that **component** satisfy the conditions specified in **conditions** (or in the case of Task **components**, we recursively check that the Task specified as the

Listing 5: Sample task: Put All X On Y

```
{
  "task_id": 110,
  "task_name": "Put All X On Y",
  "task_nparams": 3,
  "task_anchor_object": null,
  "desc": "Put all #0 #1 any #2.",
  "components": {
    "#0": {
      "determiner": "all",
      "primary_condition": "objectClass",
      "instance_shareable": false,
      "conditions": {
        "objectClass": "#0"
      },
      "condition_failure_descs": {}
    },
    "#2": {
      "determiner": "a",
      "primary_condition": "objectClass",
      "instance_shareable": true,
      "conditions": {
        "objectClass": "#2",
        "receptacle": 1
      },
      "condition_failure_descs": {}
    }
  },
  "relations": [
    {
      "property": "parentReceptacles",
      "tail_entity_list": ["#2"],
      "tail_determiner_list": ["a"],
      "head_entity_list": ["#0"],
      "head_determiner_list": ["all"],
      "failure_desc": "The #0 needs  
to be put #1 to a #2"
    }
  ]
}
```

component is satisfied). Next, we take the objects satisfying the conditions of each **component** and use them to check **relations**. If there exist objects within this subset that also satisfy all **relations**, the Task is considered satisfied.

F TEACH Examples and Qualitative Analysis

For several example figures below, we provide video session replays in the attached supplementary material. To compress video size and length, we play 1 action, either an utterance or environment action, per second, rather than the “real time” playback. Videos show the *Commander* and *Follower* ego-

Listing 6: Sample task: Put All X In One Y

```

{
  "task_id": 111,
  "task_name": "Put All X In One Y",
  "task_nparams": 3,
  "task_anchor_object": null,
  "desc": "Put all #0 #1 one #2.",
  "components": {
    "#0": {
      "determiner": "all",
      "primary_condition": "objectClass",
      "instance_shareable": false,
      "conditions": {
        "objectClass": "#0"
      },
      "condition_failure_descs": {}
    },
    "#2": {
      "determiner": "a",
      "primary_condition": "objectClass",
      "instance_shareable": true,
      "conditions": {
        "objectClass": "#2",
        "receptacle": 1
      },
      "condition_failure_descs": {}
    }
  },
  "relations": [
    {
      "property": "parentReceptacles",
      "tail_entity_list": [ "#2" ],
      "tail_determiner_list": [ "the" ],
      "head_entity_list": [ "#0" ],
      "head_determiner_list": [ "all" ],
      "failure_desc": "The #0 needs to be put #1to a single #2"
    }
  ]
}

```

centric view, as well as the object search camera for the *Commander* together with the segmentation mask of the searched object. Additionally, each video shows utterance data and progress check response data in JSON format.

TEACh was collected using an annotator interface that allowed unconstrained chat between annotators. Annotators need to communicate because the task information is only available to the *Commander*—during collection called the *User*—but only the *Follower*—in collection called the *Robot*—can actually take actions. We provide some guidelines for annotators on how to conduct these conversations,

detailed in §B. Our guidelines encourage annotators to explicitly request for and mention only task-relevant information. However, annotators can and do decide to provide annotations in different levels of detail and relevance.

Consider the example dialogs in Figures 17 and 18. In Figure 17, the *Commander* simply tells the *Follower* to prepare coffee, but in Figure 18, the *Commander* provides much lower level instructions, and waits for the *Follower* to complete each step. The initial instruction provided in Figure 17 (“*can you make me a coffee please?*”) are similar to goal-level instructions and requests typically seen in task-oriented dialog. The instructions in Figure 18 (“*grab the dirty mug out of the fridge, go wash in the sink*”) are more similar to the detailed instructions in datasets such as R2R (Anderson et al. 2018). Every trajectory in ALFRED (Shridhar et al. 2020) is annotated with instructions at both of these levels of granularity. In *TEACh*, by contrast, dialogues may contain either one or in some cases both levels of granularity. Thus, *TEACh* benchmark agents need to be able to effectively map instructions at different levels of granularity to low level actions.

Dialogues also contain situations where the *Commander* helps the *Follower* get “unstuck”. For example, in Figure 18, the *Commander* suggests that the *Follower* needs to clear out the sink in order to place the mug in it. In future work, we could attempt to leverage such utterances to learn more general knowledge of the environment that can be used by a *Follower* to get unstuck, either via student forcing from learned rules or by adding hand-written recovery modules analogous to the simple navigation and interaction recovery modules in ABP (Kim et al. 2021) and EmBERT (Suglia et al. 2021). For example, an agent may use the dialogue in Figure 18 to infer that if it tries to place an object in a container and fails, it must try to clear out the container.

In Figure 18, the *Follower* did not explicitly ask for help. In contrast, in Figure 17, the *Follower* asks for help when it does not find a mug in the places it initially searches, which prompts the *Commander* to correct their instruction. This session also illustrates a difference between *TEACh* where the task is completed by a human annotator based on online instructions from another human annotator, and benchmarks that elicit descriptions for trajectories generated by a planner, such as ALFRED. In Figure 20, the *Commander* keeps changing their mind about where the *Follower* should place the tissue boxes, resulting in a less efficient path. A human *Commander* may make mistakes when providing instructions, and a human *Follower* may not perfectly follow instructions. Standard teacher forcing training, including that used in our baseline experiments, does not account for such imperfections in demonstrated trajectories. However, robust agent models for *TEACh* benchmarks will need to learn to identify what information is essential and what is irrelevant or wrong.

Dialogues can contain a lot of feedback, for example the *Follower* informing the *Commander* when it has completed a step or the task, and the *Commander* affirming that a step or task has been completed. In the EDH and TfD tasks, an agent will likely need to learn to ignore these feedback steps. However, in future, these self-reported completions could be



Figure 17: Sample session the *Make_Coffee* task where the *Commander* does not explain in much detail how the task is to be completed. The session also includes an example where the *Follower* needs to ask for help because the *Commander* initially provided the wrong location for the mug.

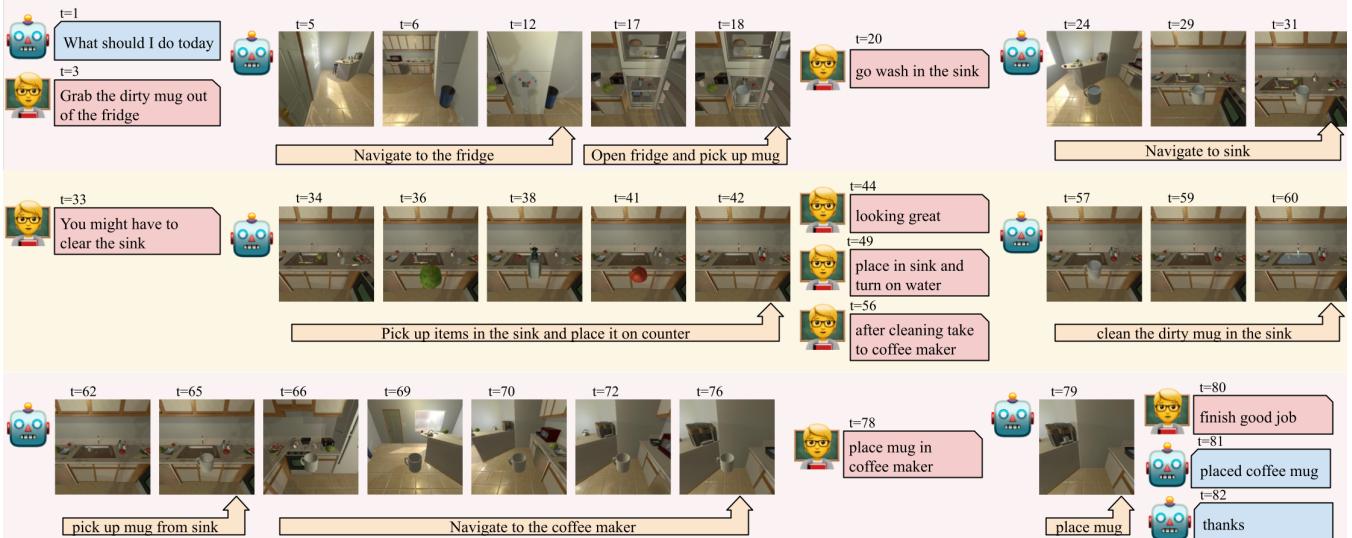


Figure 18: Another sample session the *Make_Coffee*. In this session, the *Commander* provides step by step instructions and feedback to the *Follower* even before the *Follower* asks for next instruction or help.

useful to segment large tasks into pragmatic subgoals. Unlike ALFRED, since our tasks have varying levels of hierarchy, what may constitute pragmatic subgoals for one task may be too much detail for another task.

We place no constraints on our chat interface - for example, we do not impose turn taking. Thus, chat messages from the two annotators interleave in interesting ways. For example, consider Figure 19. The *Follower*'s messages "*What task do I do today?*" and "*I have picked the purple object. What next?*" are preceded by their responses from the *Commander*. An agent performing our EDH or TfD task will need to be able to mentally reorder these messages to suc-

cessfully complete the task. To facilitate detection of interleaved messages, we provide millisecond level timesteps for each action and utterance in the *TEACH* data, though in figures we represent each action as a “timestep.”

The *Follower* can also ask for different kinds of help as they try to complete the task including clarification, for example in Figure 20, “which table? the one with the other tissue box?”, asking for the location of an object, as in Figure 17, “*Where can I find a mug?*”, and help if it is unable to perform an action requested by the *Commander*, as in Figure 17, “*I can't seem to see a mug*”. A good *Follower* model should be able to execute actions based on dialogue history,

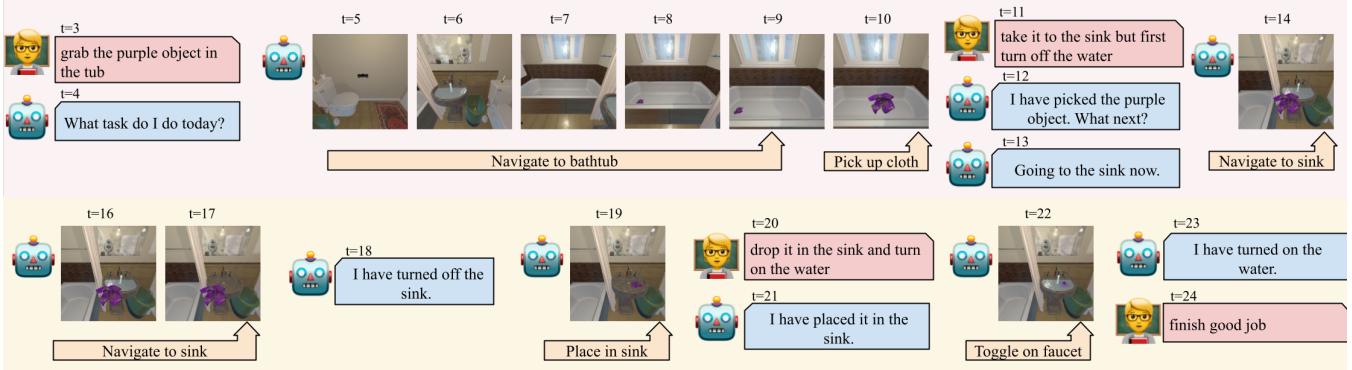


Figure 19: Sample session for the Clean All X task in a bathroom. While the task could be solved more efficiently by simply turning on the faucet in the bathtub, the *Commander* and *Follower* instead choose to clean the cloth in sink. This session also demonstrates examples of how utterances get out of order due to the absence of forced turn taking.



Figure 20: Sample session for the Put All X In One Y task. In this session, *Commander* corrects the *Follower* to pick up the correct tissue box. Then *Commander* does not realize that all the tissue boxes need to be placed on the same side table, and hence initially gives the *Follower* incorrect instructions.

while also being able to interact with the *Commander* in natural language - clarifying ambiguous instructions, obtaining additional information as needed, learning to solve problems, and providing feedback as it completes tasks. To accomplish these needs, a model may have to identify different dialog acts, translate dialog history to actions (EDH), detect situations where additional information is needed, and generate appropriate dialog responses in these situations. Jointly learning a *Commander* and *Follower* model may begin to enable these strategies.

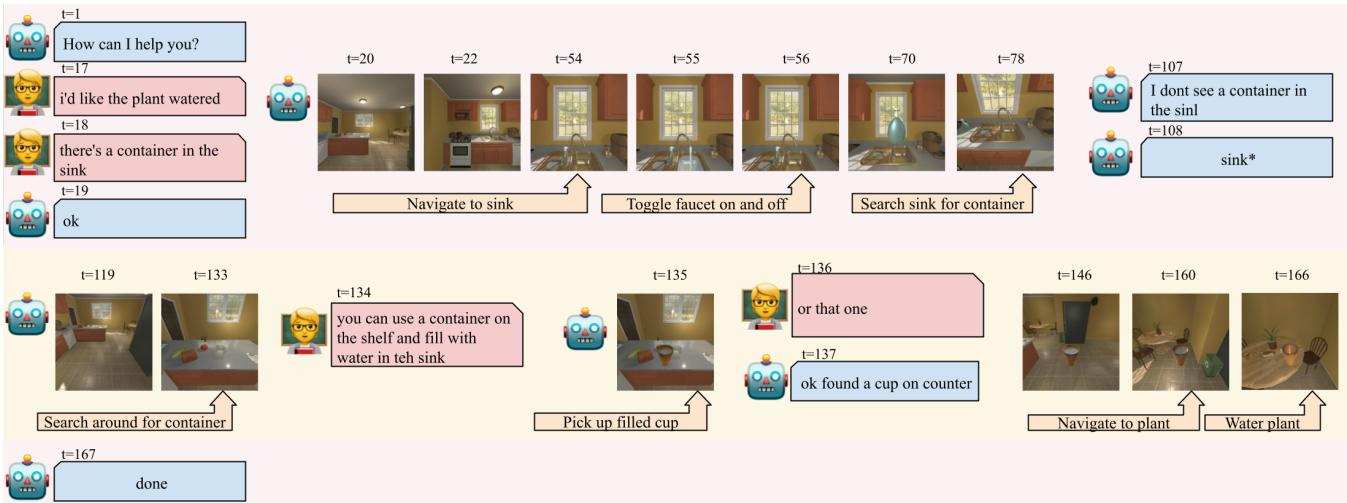


Figure 21: Sample session for the Water Plant task. The *Commander* initially gives an incorrect instruction requiring the *Follower* to ask for help and search for a container. The *Follower* finds another container before getting help from the *Commander*.

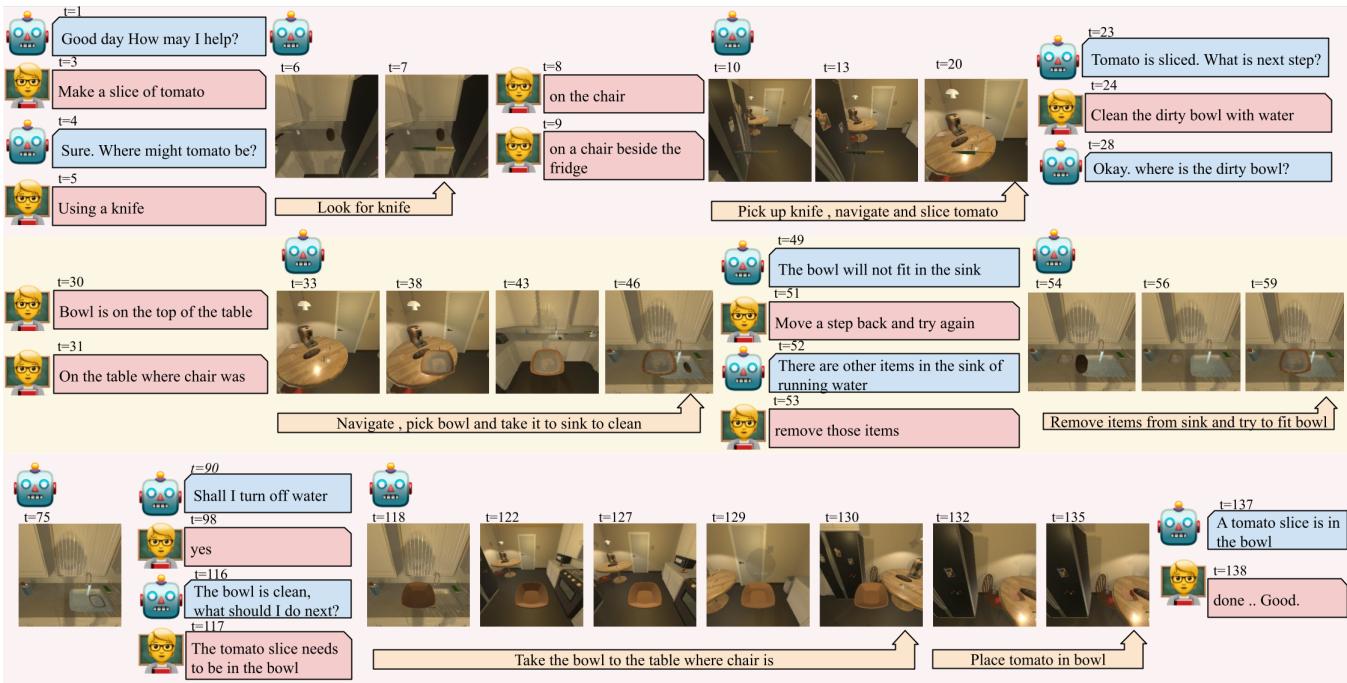


Figure 22: Sample session for the N Slices Of X In Y task. This example demonstrates interleaving chat messages between the *Commander* and *Follower*. The *Commander* uses referring expressions, such as *On the table where chair was*, to help the *Follower* locate the target object. The *Follower* also asks the *Commander* for help, and gives confirmation, frequently.

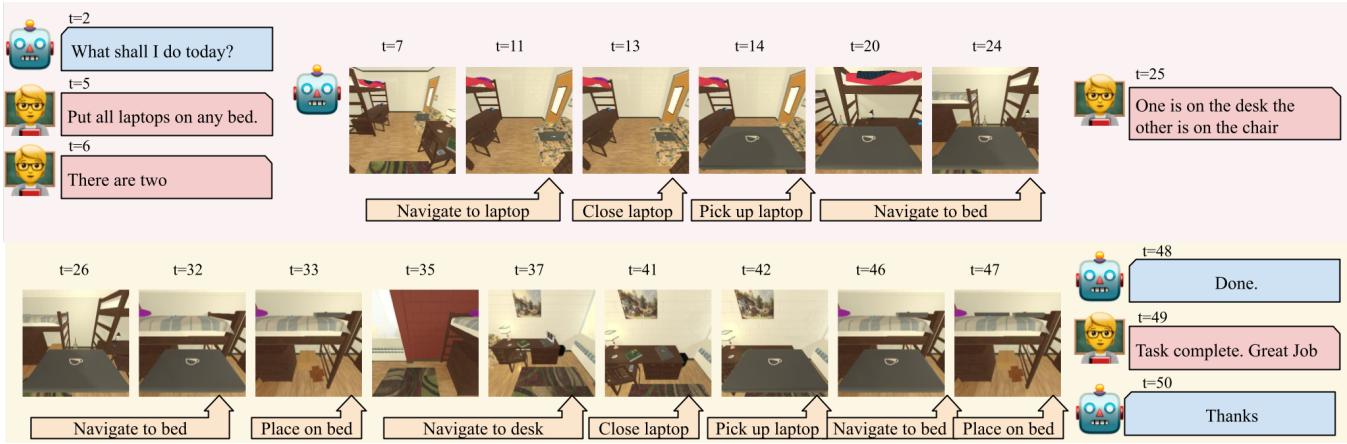


Figure 23: Sample session for the `Put All X On Y` task in a bedroom. The *Commander* intends to give step by step instructions but provides the next step before the *Follower* has finished the previous step.



Figure 24: Sample session for the `Sandwich` task. The *Follower* requires the task of making a sandwich to be broken down into simpler steps, finding the bread and knife before being explicitly asked to.

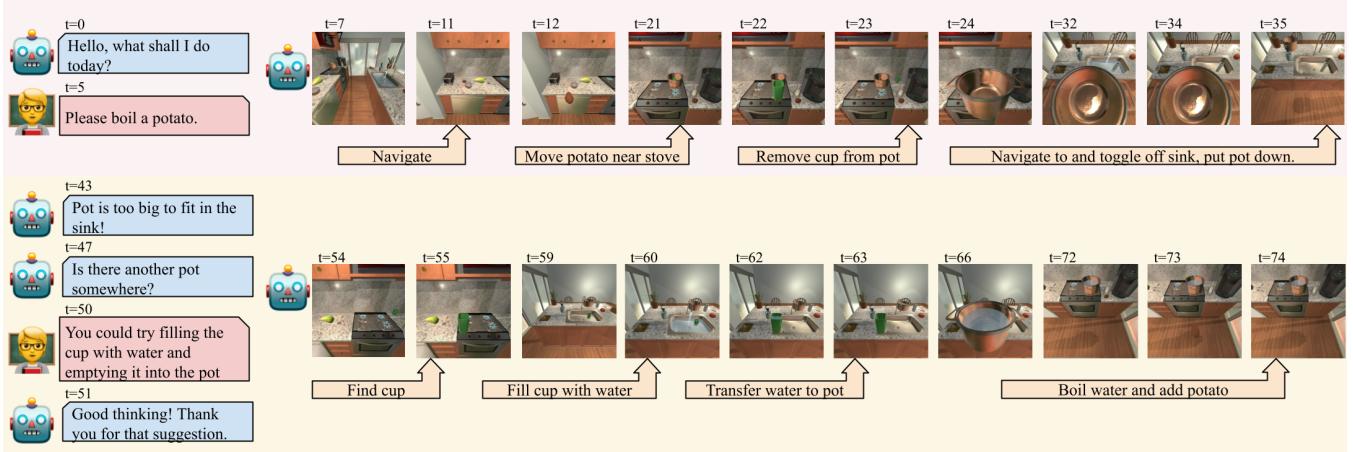


Figure 25: Sample session for the *Boil Potato* task. The *Follower* requires the task of boiling a potato where it needs to fill the pot with water before putting it on stove. The session also demonstrates example where the *Commander* helps the *Follower* to solve the issue of a pot not fitting into the sink.

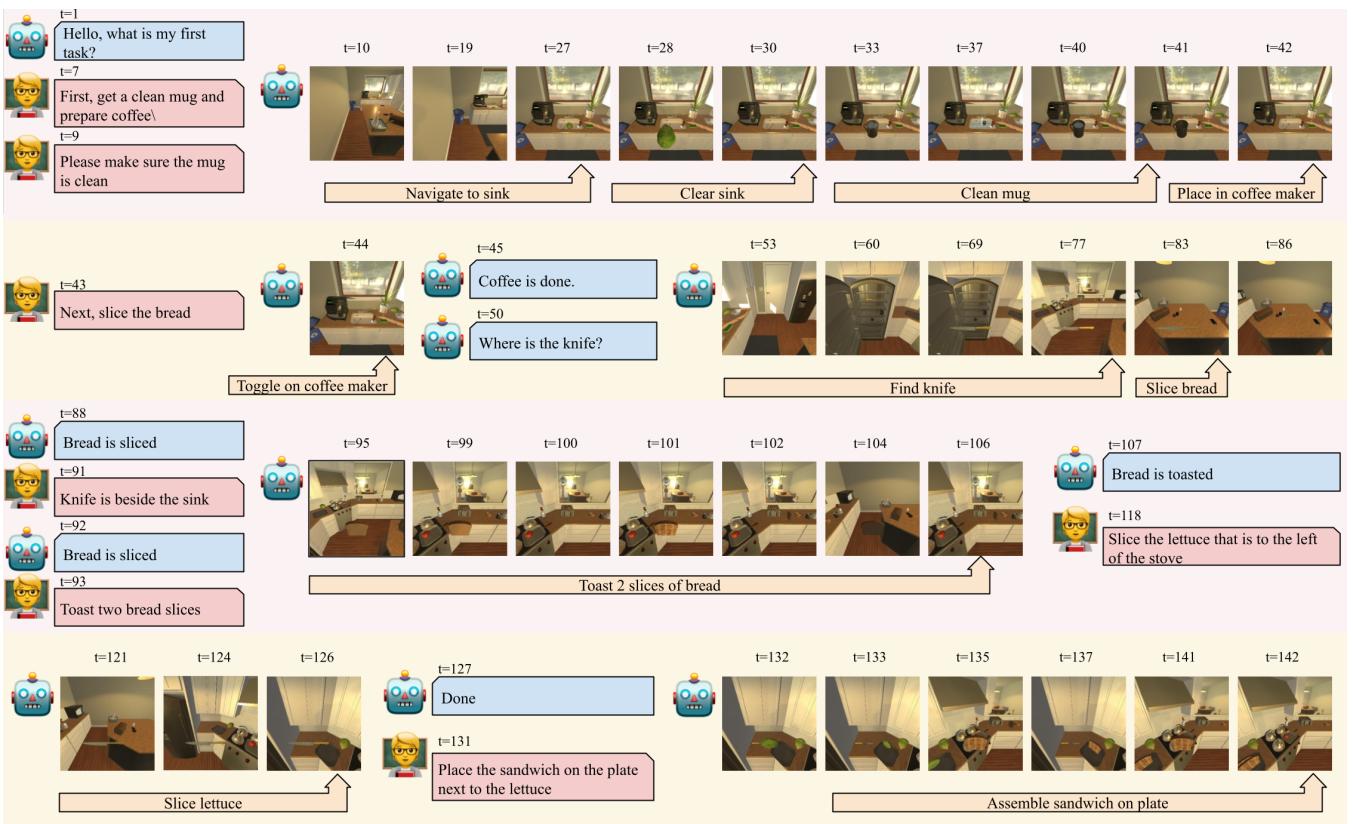


Figure 26: Sample session for the *Breakfast* task where the *Follower* has to make coffee and a sandwich with lettuce. The *Commander* provides step by step instructions but occasionally provides the next step, for example slicing bread, before the *Follower* is done with the previous step, and is sometimes late with help. For example, the *Follower* finds the knife alone because the *Commander* does not provide its location.

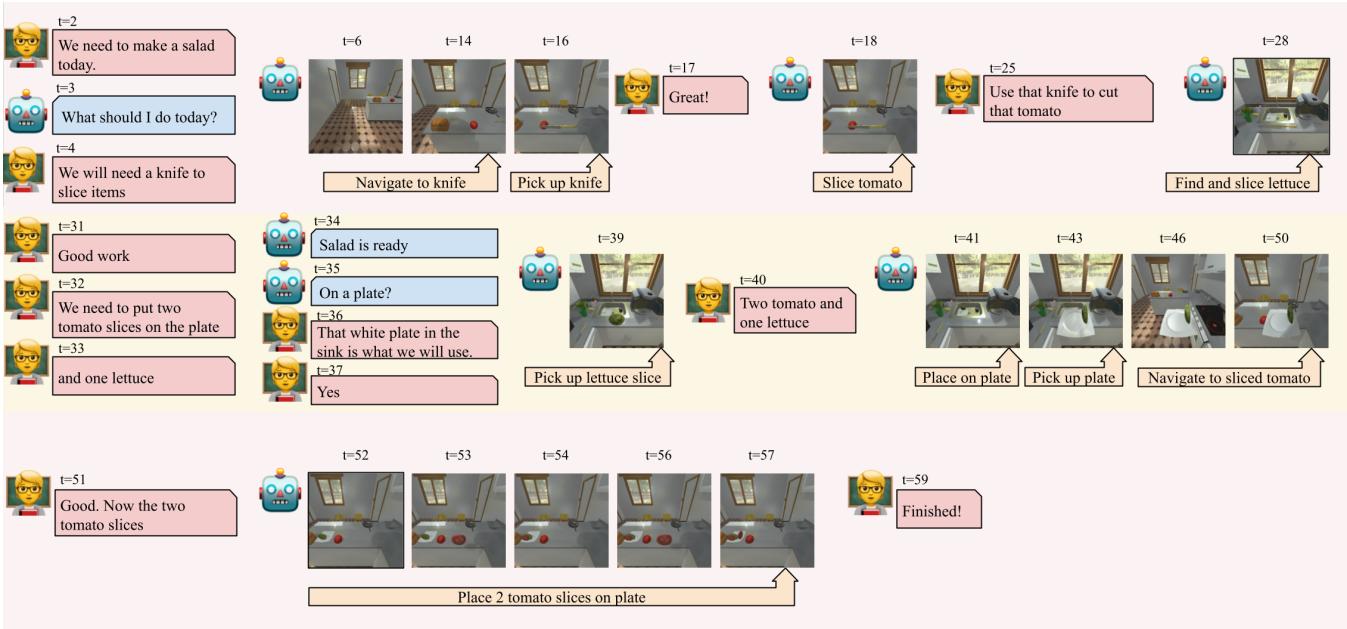


Figure 27: Sample session for the `Salad` task. The *Follower* anticipates the *Commander*'s directions, slicing the tomato and lettuce before it is asked, but forgets to plate the salad until directed to do so by the *Commander*.

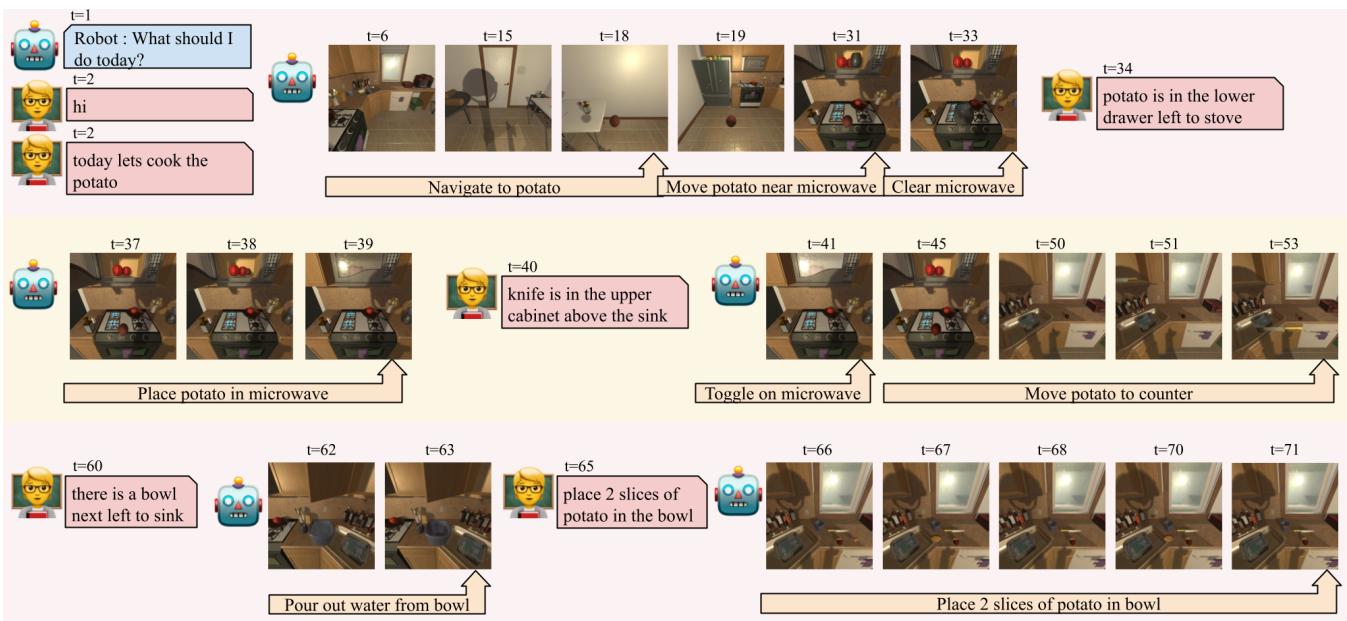


Figure 28: Sample session for the `N Cooked Slices Of X In Y` task. The *Follower* finds a potato before the *Commander* directs it to one.

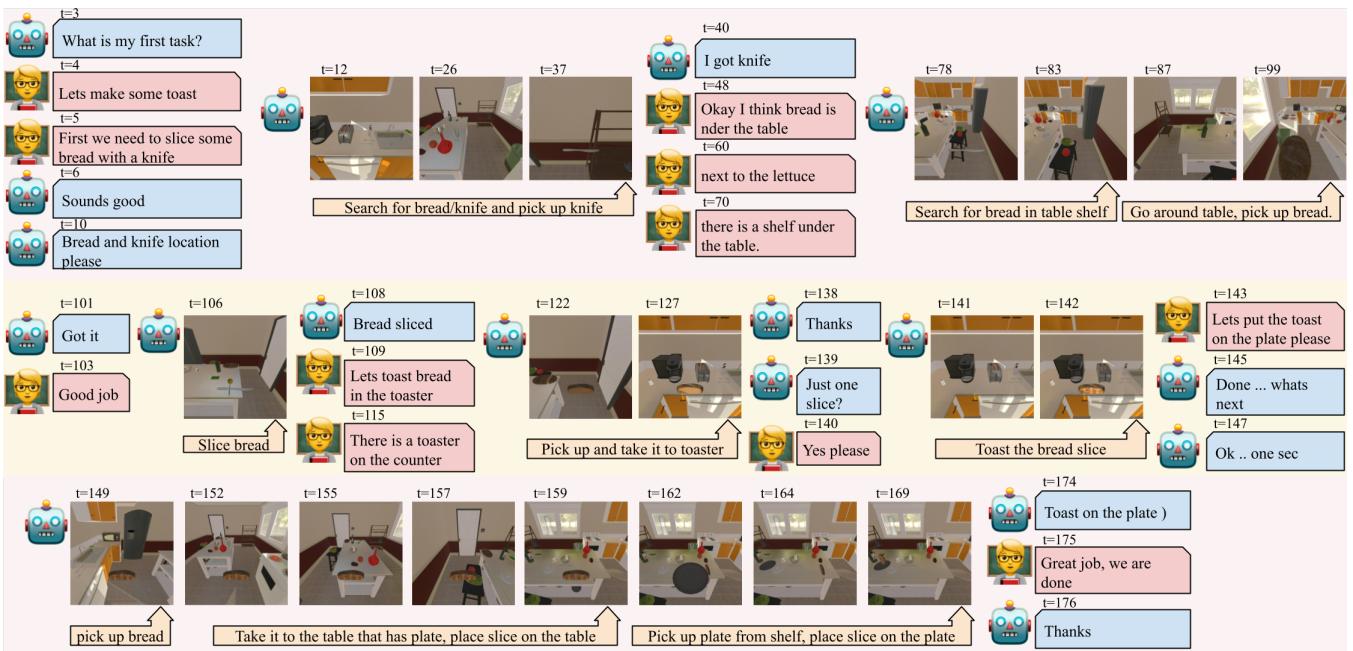


Figure 29: Sample session for the Plate Of Toast task. This session demonstrates interleaving chat messages, referring expressions and *Commander* proving feedback to the *Follower* for sub-tasks.