TEACH: Task-driven Embodied Agents that Chat

Aishwarya Padmakumar*, 1, Jesse Thomason*, 1 2, Ayush Shrivastava3, 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, both understanding and executing instructions, and using conversation to resolve ambiguity and correct 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 an initial language instruction to actions assume no further language communication (Anderson et al. 2018; Shridhar et al. 2020). However, obtaining clarification via simulated interactions (Chi et al. 2020; Nguyen and Daumé III 2019) or learning from human-human dialogue (Thomason et al. 2019; Suhr et al. 2019) can improve embodied navigation. We hypothesize that dialogue has even more to offer for object-centric, hierarchical tasks.

We introduce Task-driven Embodied Agents that Chat (TEACH) 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.

*Authors contributed equally
1https://github.com/alexa/teach

Let’s put all the forks in the sink. Check the fridge to your left.

(a) (b)
(c)
(d)

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.

Because TEACH sessions are human, rather than planner-based (Ghallab et al. 1998), Follower trajectories include mistakes and corresponding, language-guided 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 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 dialog 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.

Vision & Dialogue 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 predication 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 policies for navigating and speaking (Roman et al. 2020; Shrivastava 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 Followor 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 information, utterances between the Commander and Followor, movement actions, and object interactions taken by the Followor. 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 im-
pлемент 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 incorporate 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).

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 $A = (a_1, a_2, \ldots)$ 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 10x10 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. Table 3 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.

---

2Using the spaCy tokenizer: https://pypi.org/project/spacy/
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 $A = (a_1, a_2, \ldots)$ 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).

4.1 **Execution from Dialogue History (EDH)**

We segment **TEACH** sessions into EDH instances. We construct EDH instances $(S^E, A_H, A^D_H, F^E)$ 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 $A^D_H$ reference interaction actions taken in the session as supervision.

---

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.

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

To evaluate inferred EDH action sequences, we compare the simulator state changes $E$ 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 $E$, else 0. We average over all trajectories.
- Goal-Condition Success (GC) $(0, 1)$: The fraction of expected state changes in $F^E$ present in $E$. We average over all trajectories.
- Trajectory Weighted Metrics: For a reference trajectory $A^I_H$ and inferred action sequence $A^I$, we calculate trajectory length weighted metric for metric value $m$ as $TLW_{m} = \frac{m * |A^I_H|}{max(|A^I_H|, |A^I|)}$.

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 $(S_i, A^D_H, A^I_H, S_f)$, where $A^D_H$ is all dialogue actions taken by both agents, and $A^I_H$ is all non-dialogue actions taken by the **Follower**. We append a **Stop** action to
A model can communicate back via language generation. The TEACH dataset provides. 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 $A_H^I$, as well as $A_H^R$, to adapt to the EDH and TFD benchmark tasks. Given an EDH or TFD instance, we extract all dialogue utterances from the action history $A_H^I$ and concatenate these with a separator between utterances to form the language input. The remaining actions $A_H^R$ are fed in order as the past action input with associated image observations. Consequently, our adapted E.T. 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 $A_H^R$, 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 predicted action sequence and the ground truth action sequence.

For EDH, we ablate a history loss (H) as cross entropy over predicted action sequence and the ground truth action sequence. The model does not have temporal alignment between dialogue actions and environment actions. Consequently, our adapted E.T. does not have temporal alignment between dialogue actions and environment actions.

### 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 and goal-conditioned success by comparing TEACH challenges 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 $S_f$.

| Parameter Unique Total Sessions | Utah Sessions | 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 |
| PREPARE S | 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 |
| FRED or TouchDown (Chen et al. 2019). We calculate success with a block of instructions, analogous to ALFRED benchmark. 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 $S_f$. |

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

| Fold | Split | # Sessions | # EDH Instances |
|------|-------|------------|-----------------|
| Train | Seen | 181 (6%) | 608 (5%) |
| Val | Seen | 181 (6%) | 612 (20%) |
| Test | Unseen | 589 (19%) | 2270 (21%) |

| Parameter Unique Total Sessions | Utah Sessions | 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 |
| PREPARE S | 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 |
| FRED or TouchDown (Chen et al. 2019). We calculate success with a block of instructions, analogous to ALFRED benchmark. 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 $S_f$. |

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

| Fold | Split | # Sessions | # EDH Instances |
|------|-------|------------|-----------------|
| Train | Seen | 181 (6%) | 608 (5%) |
| Val | Seen | 181 (6%) | 612 (20%) |
| Test | Unseen | 589 (19%) | 2270 (21%) |
in TfD, \( |A_{tr}^f| = 0 \).

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 \( A_{tr}^f \) 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 condition 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 (\( \sim 130 \)) than those in ALFRED (\( \sim 50 \)).

### 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).

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...
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.

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. Note that TATC performance is not directly comparable to EDH or TID due to two-agent modeling in TATC.

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 (TID) 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 TID 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.
We are excited to explore human-in-the-loop evaluation of Commander and Follower models developed for TATC.

7 Acknowledgements
We would like to thank Ron Rezac, Shui Hu, Lucy Hu, Hangjie Shi for their assistance with the data and code release, and Siija Liu for assistance with data cleaning. We would also like to thank Nicole Chartier, Savanna Stuff, Ana Sanchez, Ben Kelk, Joel Sachar, Govind Thattai, Gaurav Sukhatme, Joel Chengottusseryi, Tony Bissell, Qiaozi Gao, Kaixiang Lin, Karthik Gopalakrishnan, Alexandros Papan gelis, Yang Liu, Mahdi Namazifar, Behnam Hedayatnia, 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:2107.05612
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.; 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).
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.; Snively, N.; and Artzi, Y. 2019. Touchdown: Natural Language Navigation and Spatio-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).
Ghaffari, 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.; Bhamari, 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. ArmaMon: 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.; Weils, 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.; Shatkin, 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 (EMNLP).
Narayan-Chen, A.; Jayannavar, P.; and Hockenmaier, J. 2019. Collaborative Dialogue in Minecraft. In Association for Computational Linguistics (ACL).

Nguyen, K.; and Daume 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 Everyday Tasks. 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.; Schlager, 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.; Yedidson, 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.

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 (§C), additional details about EDH and TfD benchmark experiments (§D), additional details about the rule-based agents we implemented for the TTAC benchmark (§E), an explanation of the task definition language that guides the **TEACh** (§F), and representative examples and qualitative analysis of **TEACh** sessions (§G).

### 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 5.

| Task                        | train | val | seen | 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).

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 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 time-outs 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 spaCy\(^5\)). We also include the distribution of the frequency with which 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 affordances, 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 Figure 10. Our sessions are quite long, often involving several hundred actions per session.

### B Data Cleaning

In addition to original utterances entered by annotators in gameplay sessions, we also release cleaned versions of the utterances. Utterances were cleaned to remove spelling errors using SymSpell\(^6\) followed by manual checking to avoid spurious changes, as well as expand contractions commonly used in chat (for example expanding “nvm” to “never mind”). We also removed utterances that only referred to aspects of the annotation interface (for example the Commander mentioning that they did not understand what object was highlighted). Utterances that involved a mix of references to the interface and task relevant information were modified to retain task relevant information while removing references to the interface. All manual annotation was done by an expert annotator. While we did not use the cleaned utterances for modeling, we believe their availability will enable better generalization of trained models to utterances received in the form of speech, and to new utterances not collected via our interface.

### C 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.*

---

\(^5\)https://pypi.org/project/spacy/

\(^6\)https://pypi.org/project/symspellpy/
Figure 6: Object Distribution: Frequency with which objects are interacted with by the Follower across all sessions. Log scale.

(a) All

(b) Verbs

(c) Nouns

(d) Adjectives

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.
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.
- 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.

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 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.

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.

D Additional EDH and Tfd Experiment Details

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 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 $1 e^{-4}$ for the first 10 epochs and $1 e^{-5}$ 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.

F 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

E 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, 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.

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, 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.
| Task                        | 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| 15.79| 10.53| 21.05| 5.26 |
| Make Plate Of Toast         | 0.00 | 2.38 | 0.00    | 7.14 | 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|
| N Slices Of X In Y          | 2.08 | 1.04 | 7.29    | 14.58| 8.33 | 10.42| 5.75 | 10.34|
| N Cooked Slices Of X In Y   | 1.15 | 4.60 | 4.60    | 8.05 | 4.60 | 6.90 | 5.75 | 10.34|
| Clean All X                 | 0.00 | 2.00 | 8.00    | 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 | 5.26 |
| Put All X On Y              | 0.00 | 0.00 | 6.25    | 10.42| 8.33 | 10.42| 6.25 | 6.25 |
| Prepare Salad               | 1.02 | 1.02 | 4.08    | 6.12 | 5.10 | 5.10 | 9.18 | 8.16 |
| Prepare Breakfast           | 0.00 | 0.56 | 4.52    | 4.52 | 4.52 | 4.52 | 3.95 | 4.52 |

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

(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 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 referenced 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 language 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 preconditions objects required to complete the task such as toast 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 2), 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 3), 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 MAKE A PLATE OF TOAST.)
Listing 1: Sample Progress Check response for MAKE A PLATE OF TOAST

```json
{
  "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.",
        }
      ]
    },
    {
      "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."
        }
      ]
    }
  ]
}
```

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 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 ex-
ample 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. 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
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 type X (x₁ and x₂) and two objects of type Y (y₁ and y₂) in the scene. If x₁ is placed in/on y₁ and x₂ is placed in/on y₂, 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 x₁ and x₂ 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 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.

G  **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 egocentric 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 §C. 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 EmbBERT [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 plan-
Figure 17: Sample session for 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 for Make Coffee. In this session, the Commander provides step by step instructions and feedback to the Follower despite the Follower not asking for the next instruction or help.

er, 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 TdF tasks, an agent will likely need to learn to ignore these feedback steps. However, in future, these self-reported completions could be 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 ex-
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 the sink. This session also demonstrates examples of how utterances can 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.

amp, 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 successfully 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, 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 occasionally 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 but anticipates a few steps, finding the bread and knife before being explicitly asked to.
Figure 25: Sample session for the Boil Potato task. The session demonstrates an 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 *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.