SEAGULL: An Embodied Agent for Instruction Following through Situated Dialog

Yichi Zhang, Jianing Yang, Keunwoo Yu, Yinpei Dai, Shane Storks
Yuwei Bao, Jiayi Pan, Nikhil Devraj, Ziqiao Ma, Joyce Chai

University of Michigan

{zhangyic, jianingy, kpyu, daiyp, sstorks, yuweibao, jiayipan, devrajn, marstin, chaijy}@umich.edu

Abstract

The growing demand for advanced AI necessitates the development of an intelligent agent capable of perceiving, reasoning, acting, and communicating within an embodied environment. We introduce SEAGULL, an interactive embodied agent designed for the inaugural Alexa Prize SimBot Challenge, which can complete complex tasks in the Arena simulation environment through dialog with users. SEAGULL is engineered to be efficient, user-centric, and continuously improving. To achieve these goals, we develop a modular system that combines neural and symbolic components. Our natural language understanding module employs a hierarchical pipeline to convert user utterances into logical symbolic representations of their intentions and semantics. Meanwhile, a neural vision module detects object classes, states, and spatial relations. These multi-sensory inputs are then processed by a state tracker to update the agent's beliefs regarding the world state, user intentions, and task progress. A central policy interprets the neuro-symbolic beliefs and selects one of several available skills, including navigation, planning, and dialog. We place particular emphasis on optimizing dialog flow and user experience, ensuring that users have a responsive, natural, informative, and engaging interaction with our bot. Furthermore, we have developed tools and pipelines to augment our vision and language data, continually enhancing our system's robustness and performance.

1 Introduction

We envision that the forthcoming generation of artificial intelligence (AI) will adopt an embodied paradigm [1, 9, 3, 12]: one that enables AI agents to operate in the physical realm, engage in object manipulation, interpret and process multimodal inputs, and learn from situated communication with humans. The potential impact of developing an efficacious embodied agent is tremendous, spanning from robots that serve as waiters in restaurants and assist elderly individuals to complete household chores, to the aspiration of artificial general intelligence (AGI). Recent advances in the field of computer vision and natural language processing have reached a level of maturity that facilitates the realization of embodied AI. The aim of the Alexa SimBot Challenge is to design a bot that can perceive, reason, act, and communicate in Arena simulation environment [5], while utilizing voice-based interactions with Alexa users to collaboratively accomplish tasks. Compared to earlier Alexa Prize challenges, the SimBot Challenge represents a crucial milestone in the integration of visual perception, situated reasoning, and task-driven decision-making in a physical environment.

In Phase 1 of the SimBot Challenge, we developed the Deliberative Agent for following Natural Language Instructions (DANLI) [13], the state-of-the-art approach to the TEACh dataset [11] that integrated persistent world representations with tiered neuro-symbolic planning to solve long-horizon tasks. For the second phase, building upon DANLI, we introduce the Situated and Embodied Agent with GroUnded Language Learning (SEAGULL) agent. SEAGULL is similar in spirit to DANLI, using a persistent world representation to reason over and execute Arena players' requests. We adapted several aspects of DANLI into streamlined, efficient, as well as continuously improvable, and expandable modules.

In this report, we introduce the SEAGULL agent, an AI system designed to effectively respond to verbal commands and accomplish tasks in the virtual Arena world. The primary objective of SEAGULL is to develop strong baseline models for language understanding, multimodal perception and reasoning, commonsense knowledge, and user-centered dialog policy.

1.1 Advantages and Key Innovations

The design of the SEAGULL system offers several advantages and introduces key innovations:

1. Modularity: The modular architecture permits easy addition, modification, and removal of skills and policies, fostering system flexibility and adaptability.
2. Comprehensive Skill Set: The extensive skill library allows the system to tackle a broad array of tasks and situations, improving its overall effectiveness.
3. Robust State Tracking: The state tracking module provides a rich understanding of the current situation, leading to more informed and transparent decision-making.
4. Dynamic Policy Management: The policy controller enables the system to adapt its behavior based on the current state, ensuring appropriate responses and actions across diverse situations.
5. Enhanced user Experience: The system is designed with user experience in mind, ensuring responsiveness, natural interaction, warmth, and informativeness for an optimized user experience.

In summary, the SEAGULL system represents an integrated AI solution designed for complex virtual environments. Its modular architecture, comprehensive skill set, robust state tracking, and dynamic policy management form the foundation for strong performance in language understanding, multi-modal reasoning, commonsense knowledge, and user-centered dialog policy.

2 System Overview

2.1 Core Components

The architecture of SEAGULL is a modular system, consisting of four main components:

Skill Library. SEAGULL is equipped with an extensive skill library that encompasses natural language understanding (NLU), vision, causal effect reasoning, navigation, planning, dialog, and natural language generation (NLG) skills. These skills correspond to the specific capabilities of the system, empowering it to parse user input, generate plans, and execute actions, all while providing coherent and contextually appropriate responses.

State Tracking Module. This core component represents the agent's complete understanding of the current situation. It is updated based on the output of specific skills and offers situational context for their optimal functionality. The state tracking module is composed of four types of states, namely bot state, world state, interaction history, and task state.

Knowledge Module. The knowledge module supplies domain-specific knowledge for all the skills and the state tracker. It integrates object affordances, probable object locations, and conventions for naming and referring to rare objects, thereby enhancing the system's overall understanding and performance.

Policy Module. The modularized policy module is tasked with determining which skill to employ based on the current state, in order to produce the most suitable response and actions. The system implements various policies and utilizes a policy controller to dynamically switch between them, adapting to a range of situations.

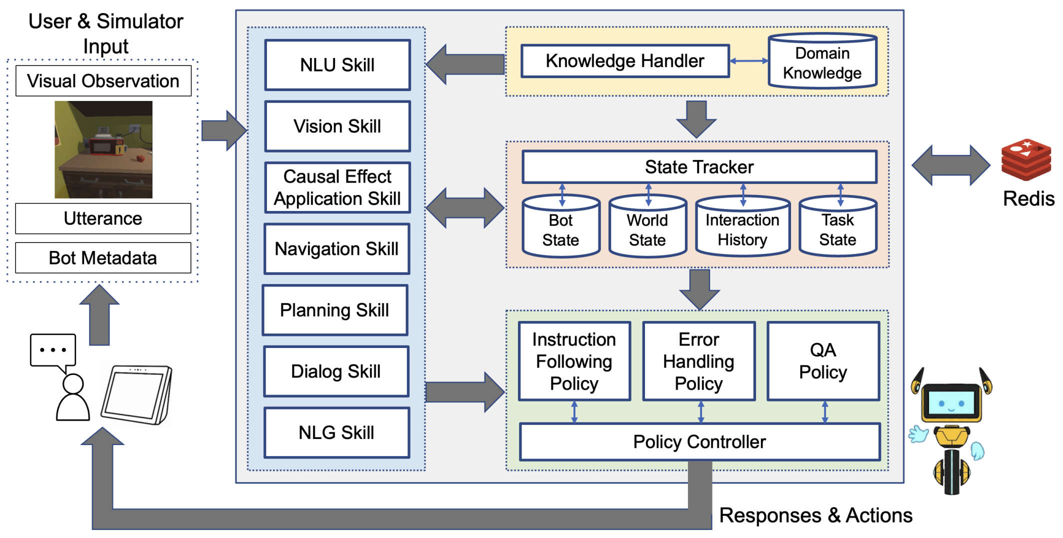


Figure 1: The overall architectural design of the SEAGULL system. As a modulated AI system, SEAGULL is composed of four main components: a skill library with various understanding, reasoning, planning, and interaction skills; a state tracker that updates upon the outcomes of skills; a policy module that determines which skill to employ; and a knowledge base that provides domain-specific knowledge.

Overall, the SEAGULL system offers a modular and versatile AI solution for effectively responding to verbal commands and accomplishing tasks in virtual environments. The four main components—Skill Library, State Tracking Module, Knowledge Module, and Policy Module—work synergistically to provide a robust and adaptable system.

2.2 System Engineering

Agent State Persistence The SimBot challenge adopts a stateless backend design where users' utterances in a game session come as separate HTTP requests. However, it is necessary to preserve the state of the agent during a game because an embodied agent often relies on the context of a session to understand user intents and execute actions. To bridge this gap, we implement an agent state persistence mechanism where at the end of each request, the entire state of the agent for this game (games are uniquely identified using session IDs) is serialized and saved to cache; at the beginning of the next request of the same session, the agent is deserialized and restored so that its full context is restored. The serialization and deserialization are cascading, meaning all fields in any child skills, states, knowledge, and policies mentioned in Section 2.1 will be included. This system-level design alleviates developers from the hassle of serialization and deserialization and thus enabling developers to implement complex logic easily by storing any information they need as if the backend is stateful. Specifically, we use Python packages `attrs` [1] and `cattrs` [2] for (de)serialization and AWS ElasticCache for caching.

Game Session Persistence and Logging To enable easy debugging and continual improvement, it is necessary to have a robust system of game session data persistence and logging. We use AWS DynamoDB to store all game interaction data including game session ID, timestamp, user/bot utterances, and bot actions. In addition, we connect our logging system to AWS CloudWatch to store all execution logs. These setups prepare us well for quickly identifying user pain points and continually improving system capabilities.

Canary Tests Sometimes a critical error or a rogue code push can cause system downtime or significant regression. Our ratings are usually severely impacted under such circumstances. To avoid

[1]`https://www.attrs.org`
[2]`https://catt.rs`

catastrophic situations like these, we implemented canary tests to emulate user requests. These canary tests are run every 5 minutes and will notify us once anything abnormal happens. Canary tests have saved us from several cases that could have caused severe issues to our system. We used AWS CloudWatch Synthetic Canaries and AWS Simple Notification Service to implement this pipeline.

2.3 Development Pipeline

As described in 2, SEAGULL is a modular system with many components working in conjunction. Therefore, the development effort has to be a team-wide effort with multiple team members making changes to the system in parallel. At the same time, we need to ensure the correctness of the system while preserving development speed and fast iteration time. Given these requirements, it is imperative that we follow the best software engineering practices and establish a robust development pipeline that supports continuous integration to the production environment.

Continuous Integration The main goal of our development pipeline is continuous integration to the production environment to support rapid iteration. Once a new piece of code is merged into the main branch, it is automatically deployed to the staging environment. When the team is ready to make a deployment to the production environment, we create a tag on the repository, and the tagged code is automatically deployed to the production environment. The tag naming convention is the date of the deployment in order to quickly and accurately keep track of which code is running on the production environment. The continuous integration pipeline is implemented using GitLab CI/CD[3].

Code Quality For every commit, we run a set of automated code quality tools to enforce a uniform coding style. This includes autoformatting tools such as black[4], as well as linters like flake8[5]. This is to discourage bike-shedding[6] and encourage discussions about substantive matters regarding the system and code. These tools are run locally and as part of the continuous integration pipeline using pre-commit[7].

Type Annotations We require developers to use Python type annotations and run a static type checker called mypy[8] on every commit using pre-commit, both locally and on the continuous integration pipeline. Python type annotations may not be as beneficial for small scripts for research projects, but it is absolutely vital for a big, collaborative project like SEAGULL. It catches many small mistakes and bugs early on, thereby greatly increasing the robustness of the system. Furthermore, it serves as a secondary documentation, lowering the cost of information propagation within the team.

Testing We enforce that no piece of code can be merged to the main branch without passing a suite of automated unit and integration tests. These tests range from simple input-output pairs for our NLU module to heavy integration tests with Amazon DynamoDB. In an ideal world, all of our code should have accompanying tests, but in the real world, we have to make trade-offs between writing tests and development speed. As a result, we sometimes allow code to be merged into the main branch without tests, but we make sure to pay off the tech debt incurred here at a later time by refactoring the untested code to be more testable and adding appropriate tests. This effort is actually aided by the fact that the rest of our code base is well-tested, as we can perform large refactors without the fear of introducing regressions. Overall, the time spent writing testable code and adding tests is well-compensated by the time saved from debugging.

Code Reviews Once a piece of code passes the suite of automated tests, it goes through a code review process before it is merged into the main branch. Relevant code reviewers ensure that the overall design of the code is sound and fits well with the rest of the system. Automated code quality tools and tests play a crucial role here as they help reviewers focus on the substantive matter rather than trivial matters like coding style. We use GitLab's built-in code review tools to implement the process.

[3]`https://docs.gitlab.com/ee/ci/`

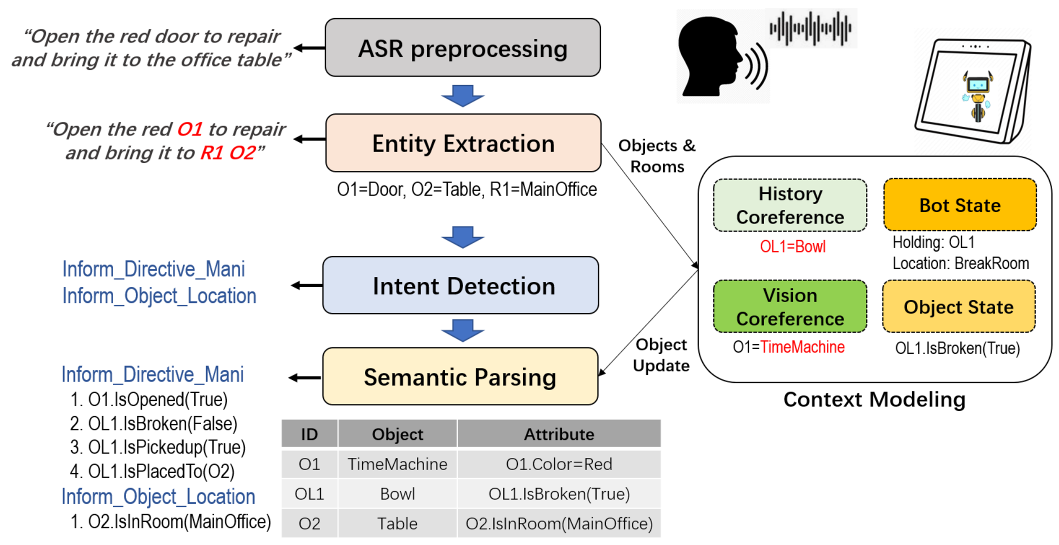
[4]`https://github.com/psf/black`

[5]`https://flake8.pycqa.org/`

[6]`https://en.wikipedia.org/wiki/Law_of_triviality/`

[7]`https://pre-commit.com/`

[8]`https://www.mypy-lang.org/`



ID	Object	Attribute
O1	TimeMachine	O1.Color=Red
OL1	Bowl	OL1.IsBroken(True)
O2	Table	O2.IsInRoom(MainOffice)

Figure 2: The Natural Language Understanding (NLU) pipeline. NLU can combine the information from dialog history, vision input, bot state, and object state to make predictions. The goal states are inferred as a sequence of triples like `X.IsPlacedTo(Y)` that the agent needs to accomplish. In the given scenario, the user intends to ask the agent to perform a series of actions: open the red door of the time machine, repair the broken bowl the agent is holding, and subsequently take the fixed bowl to a table located in the main office. However, the user ends up giving an utterance like "open the red door to repair and bring it to the office table", which requires complex context modeling and commonsense reasoning to accurately predict the intended goal state. Therefore, the NLU pipeline plays a vital role in understanding such hard utterances and ensuring that the agent can comprehend and execute the desired actions. The example user utterance used in this figure comes from a SEAGULL team member's interaction with the bot rather than from a real customer's interaction.

Documentation Another challenge in managing large system development is the efficient distribution of information about different parts of the system. This is crucial in terms of not only collaboration and development speed but also ensuring the overall correctness of the system. This problem is especially acute in a system like SEAGULL which consists of specialized modules developed by different members. Code review addresses this problem to a certain level, but it cannot be the only solution due to its scalability (developers cannot review every single piece of code). To address this issue, we encourage developers to document their code and use automated documentation generation tools like Sphinx AutoAPI[9] to keep the documentation up-to-date. Other important processes are manually documented in the same documentation and tracked in the git repository alongside the code.

3 Natural Language Understanding

Natural Language Understanding (NLU) is one of the crucial components of spoken dialogue systems, which is responsible for analyzing and interpreting users' inputs and extracting the relevant information for downstream modules like the dialogue manager. Typically, NLU involves several sub-tasks, including identifying the intent behind the user's query, extracting named entities, and parsing the syntax and semantics of the input. In our SimBot, we build a pipeline-based NLU framework to understand and respond to user queries.

3.1 NLU pipelines

Figure 2 illustrates the pipeline in our SEAGULL NLU module, which consists of several sub-modules executed sequentially to ensure an accurate understanding of user utterances.

[9]`https://sphinx-autoapi.readthedocs.io/`

Entity Extraction. Entity extraction is the process of identifying pertinent entity information within the user's input, such as objects, rooms, and object attributes (e.g., color and spatial location). For instance, in Figure 2, the utterance "open the red door to repair and bring it to the office table" is processed and delexicalized into "open the red `O1` to repair and bring it to `R1 O2`", where the detected entities are `O1=door`, `R1=office`, and `O2=table`. This information is essential for parsing the intricate semantic structures present in the sentence. The delexicalized utterance and the extracted entities are then forwarded to the next submodule for further analysis.

Intent Detection. We have defined a hierarchical user intention taxonomy in our schema, consisting of six main categories: `Inform`, `Reply`, `Ask`, `Social`, `Profanity`, and `Others`. Each category contains several fine-grained user intent classes in the following. A comprehensive description can be found in Appendix A.1.

- `Inform` is a category in which the user informs some instructions or descriptions to the bot.
  - `Inform_Directive_Mani` indicates a goal state the bot needs to complete.
  - `Inform_Directive_Navi` indicates where the bot should navigate.
  - `Inform_Directive_CommonTask` indicates a common task the bot should perform.
  - `Inform_Task_Goal` specifies the main goal of a task.
  - `Inform_Object_Location` provides the spatial location of an object.
  - `Inform_Object_State` provides the current state of an object.
- `Reply` is a category in which the user replies to the bot's previous message.
  - `Reply_No` is a negative response.
  - `Reply_Yes` is an affirmative response.
  - `Reply_Notsure` is an uncertain response.
- `Ask` is a category in which the user asks a question.
  - `Ask_Agent_Location` asks about the bot's location.
  - `Ask_Agent_Inventory` asks about the what's holding in the agent.
  - `Ask_Agent_Capability` asks what the bot can do in the challenge.
- `Social` is a category for social and chit-chat interactions.
- `Profanity` is a category for profane or inappropriate content.
- `Others` is a category for miscellaneous intents.

Given the last system dialog act and the current delexicalized user utterance as input, we employ a regex parser to determine possible user intents. For instance, if the last system dialog act confirms an action and the user responds with "yes," the parser will output an intent `Reply_Yes`. Our experiments have shown that the most frequent user intents are `Inform_Directive_Manipulation`, `Inform_Directive_Navigation`, and `Inform_Directive_CommonTask`, which are closely related to semantic parsing.

Semantic Parsing. Semantic parsing involves analyzing the syntax and semantics of the user's input to generate structured representations. This step typically involves converting the input text into goal states, which are denoted as triples, such as `X.IsPickUped(True)` and `X.IsPlacedTo(Y)`. Additionally, there are triples that describe conditions that must be met when the planner (sec. 6) reaches the goal states, for example, the `O2.IsInRoom(MainOffice)` for the intent `Inform_Object_Location` in Figure 2, which implies that the bot should attempt to place the bowl onto a table in the main office.

Context Modeling Context modeling resolves ambiguities and maintains a stack coherently to record the objects mentioned by the user or detected by vision models. The stack is then used to update the extracted entities by previous sub-modules, as shown in Figure 2:

- In the stack, we first add all mentioned objects in the dialogue history (e.g., the 'bowl' in the last turn) so that when a co-reference expression appears in the current sentence (e.g., 'it') and the parsed objects are insufficient for predicting goal states, we will systematically examine mentioned objects to determine a suitable object to complete the goal states.
- To integrate visual information, we also add objects detected by the vision model to the stack to address the vision-text co-reference issue. For example, in Figure 2, the utterance "open the red door" actually refers to "open the time machine" because the time machine has a red door. A goal state like `Door.IsOpened(True)` is not contextually coherent; thus, our NLU replaces the original entity `door` with the detected object `time machine`.

- We consider the inventory information, i.e., the objects currently held by the bot, from the bot state (ref to sec. 5) and add it to the stack for matching. If the current utterance cannot be parsed into complete goal states.
- We also utilize information about the existing object state for mentioned objects from the world state (ref to sec. 5), which helps filter out false positive predictions.

In this way, we can combine information from both visual and linguistic inputs to create a contextualized understanding of human-robot conversations. Since there is no adequate amount of annotated data to train robust neural models from scratch, the NLU module was implemented based on parsing rules and achieved 98% overall semantic parsing accuracy in a small clean evaluation set (1,000 utterances) extracted from the noisy trajectory data provided by Amazon.

3.2 Data Augmentation

Data augmentation is a technique used to expand and diversify the training dataset, which can improve the performance and generalizability of the dialogue system. In the context of NLU, data augmentation can involve generating variations of existing data points or creating entirely new instances to cover a wider range of possible user inputs.

Data Augmentation with LLMs Large Language Models (LLMs), such as GPT-3 [2] and ChatGPT[10], are used to generate synthetic data for augmenting the existing testing dataset. We first manually generate the semantic labels such as `Apple.IsPickedUp` and the corresponding simulated user utterances like "pick up the apple", then using the prompting technique to ask LLMs to generate more diverse expressions. For reliability considerations, we leverage the additional data in a rule-based manner, i.e., we continually improve the rules in our NLU parser by covering more unexpected corner cases (around 3,000 utterances) provided by text-davinci-003 in the OpenAI playground. We provide detailed templates and examples in Appendix A.3. After the improvement, NLU parser performance is able to boost from 83% to 92% on the additional augmented dataset.

4 Visual Perception

Visual perception is another important component for the bot to understand the situation. In SEAGULL, the vision skill of consists three models: a hierarchical object recognition model, an object state detector, and a supporting relation detection model. The goal of these models is to process the raw visual inputs and output symbolic object-centric representations to be used in downstream world state tracking and planning.

4.1 Hierarchical Object Recognition

The hierarchical object recognition system allows SEAGULL to recognize objects with base categories, and subsequently distinguish finer-grained categories upon closer inspection or when viewed from a better angle. For example, in the Arena environment, four different types of cartridges (mug, hammer, action figure, and lever) can be used to 3D print specific objects. The bot may be able to detect that an object is a cartridge from a distance, which is useful when searching for a cartridge in the room. However, to determine the specific type of cartridge, the bot must examine the icon on the cartridge, requiring a closer view or a better angle. The hierarchical object recognition algorithm empowers the bot to take advantage of both coarse and fine-grained object categorization, adapting to the available visual input and task requirements.

We train a Mask2Former model [4] to detect objects with 96 distinct object classes (referred to as the coarse object types). We first train the model on the vision dataset provided by the Alexa team, which consists of 514,321 images collected by traversing the agent in the room to face various objects. However, we find that the model trained on this data performs badly under the following scenarios: (1) when there is a state change of the objects, such as a cake becomes frozen; (2) when the illumination condition changes, such as the room light is turned off or the object is placed inside a container like a microwave; (3) when the object is small (e.g. bread slice, floppy disk, etc).

[10]`https://openai.com/blog/chatgpt`

To boost the performance, we developed a robust, high-performance pipeline that encompasses a data collection engine, an effective training recipe, and a state-of-the-art detection architecture. We collect a custom dataset as a complementary of the original vision dataset. We modify the Arena CDF files to add objects with customized states into the scene and drive the agent to look at them from different angles. To ensure data coverage, we enumerate all the possible state changes according to the object affordance and manually identify objects that the vision model struggles to detect. We iteratively update our model and data several times. The final dataset has 50,858 images in total. To improve model training, we also applied a comprehensive set of data augmentation techniques, including CopyPaste [6], PhotoMetricDistortion, and RandomCrop. We utilized the per-class balancing algorithm from LVIS [7] with an oversampling threshold of 0.015 to make the data distribution less biased.

Based on the object detections of coarse categories, we train another classifier to further predict 219 fine-grained object classes, which can distinguish different types of objects of the same coarse type, such as different cartridges. This model is based on ResNet-50 [8] with several layers of MLP layers on top. We evaluate the fine-grained type classification model on the validation vision data provided by the Alexa team. The model achieves an accuracy of 93.8% in identifying fine-grained object classes.

4.2 Object State Estimation

Beyond object types, we also train another set of classifiers to predict essential object states from observations, such as `isBroken`, `isOpened`, etc. Due to efficiency concerns, in practice, this model was implemented as another 2 layers of MLP on top of the ResNet-50 backbone of the fine-grained object-class classifier. Our model was able to correctly infer object state with an accuracy of approximately 95%.

Besides the object's physical state, we also designed a pixel-based color recognition algorithm to identify object colors. Given an object bounding box, we compute the nearest neighbor of each pixel in the box to a set of base colors, where the distance is computed as the Euclidean distance in the `Lab` color space. We keep the colors that have a proportion of 15% or higher as the predicted object colors.

4.3 Object Relation Detection

Finally, we identify spatial relationships between objects to enable SEAGULL to reason about object positions and interactions. These relationships can be represented as object tuples (*base-object*, *supported-object*), where the *base-object* supports the *supported-object*, like a table supporting a cup.

Upon detecting object classes using the object recognition system, objects are filtered into two categories: potential *base-objects* and potential *supported-objects*. The system then enumerates all possible combinations within the detected objects and employs bounding box information to determine if the relationship holds. To verify the relationship, simple yet robust heuristics are used, such as ensuring the centroid of the *supported-object* lies within the *base-object* and that the *supported-object* is positioned higher than the *base-object*.

5 State Tracking

The state tracking module serves as the core component of the SEAGULL system, representing the bot's comprehensive understanding of the current situation. This understanding is crucial for the policy module, as it leverages this information to determine the next actions. The state is updated based on the output of various skills, such as the parsing result from the natural language understanding (NLU) skill and the object detection and state estimation result from the vision skill. Additionally, the state tracking module provides situational context, allowing these skills to function more effectively. The module comprises four distinct types of states:

- Bot State: The bot state encompasses various aspects of the bot's current status, including its pose, location, the room it is in, and its inventory (items held by the bot). This information enables the system to maintain an accurate representation of the bot's position and capabilities in the virtual environment.

- World State: The world state captures the bot's beliefs about all objects, their physical states (e.g., a `Bowl` has `isBroken=True`), and their relationships (e.g., `Plate.isPlacedTo(Table)`). These beliefs can be updated from multiple information sources, such as the bot's visual observations, the user's language descriptions or the causal effects of actions. This comprehensive representation of the environment provides a rich context for the bot to make informed decisions.
- Interaction History: The interaction history records all exchanges between the bot and the user. Coreference resolution, the process of identifying which noun phrases refer to the same entity, is essential for accurately interpreting user input based on the interaction history. For example, if a user says "pick it up" after the bot approaches an apple, "it" refers to the apple. By maintaining a complete interaction history, the bot can better understand user intentions, especially when the instructions are partially specified.
- Task State: The task state corresponds to the current progress of a task. As tasks can be complex and comprise multiple sub-goals, tracking progress is vital for understanding the situation and deciding on the next steps. This state allows the bot to effectively manage tasks by breaking them down into manageable sub-goals and updating its progress as it proceeds.

The state tracking module, with its four types of states, enables the SEAGULL system to maintain an accurate and dynamic understanding of the bot's status, the environment, the interaction history, and the progress of tasks. This comprehensive representation empowers the system to make informed decisions, adapt to various situations, and provide contextually appropriate responses, contributing to the overall effectiveness and adaptability of the SEAGULL system.

5.1 Belief Update from Multiple Sources

Estimating the belief state is of paramount importance for decision-making within SEAGULL. A comprehensive and accurate belief state enables the bot to make informed decisions, react appropriately to user commands, and adapt to changes in the environment. In SEAGULL, the belief state is updated from multiple sources, including visual observations, natural language descriptions, and action causal effects. This section elaborates on the approaches used in SEAGULL to update the belief state and the benefits of such design.

Belief Update from Visual Observations. In SEAGULL, the belief state is updated based on the bot's visual observations. When an object is detected, it is added to the belief state along with its estimated physical state. This approach ensures that the belief state continuously incorporates new information as the bot navigates the environment, enhancing its decision-making capabilities. For example, if the bot observes a bowl of milk on a table, it will update its belief state with this information, allowing it to respond accurately to user commands involving the bowl.

Belief Update from Natural Language Descriptions. The belief state is also updated using information obtained from natural language descriptions provided by the user. When the user mentions specific details about the environment, such as the presence of an apple in the fridge, the system incorporates this fact into the belief state. This feature allows the bot to integrate user-provided knowledge and more effectively address the task.

Belief Update from Action Causal Effects. SEAGULL maintains a human-constructed knowledge base about action cause-effects. Given an action, SEAGULL updates its belief about objects' physical states using cause-effect knowledge. This method enables the bot to maintain an up-to-date belief state that accurately reflects the consequences of its actions. For instance, if the bot opens a fridge, the belief state will be updated to indicate that the fridge is now open (`Fridge.isOpened=True`). This awareness of the causal effects of actions reduces the uncertainty of object state estimations from noisy vision and NLU outputs, which significantly improves the accuracy of the bot's belief.

The final belief state is a combination of beliefs derived from these three sources. The integration of information from visual observations, natural language descriptions, and action causal effects enables the SEAGULL system to maintain a comprehensive and dynamic belief state. Whenever there is uncertainty, especially a conflict from multiple sources of information, the policy module would take

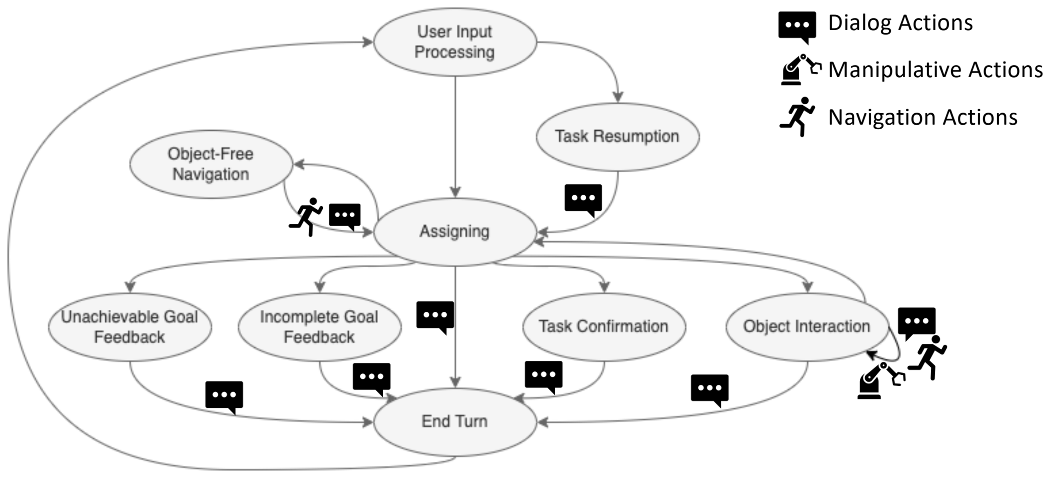


Figure 3: Diagram of the finite state machine for SEAGULL's instruction following policy. Different types of actions (i.e. dialog actions, manipulative actions and navigation actions) are added to the bot's execution queue as a consequence of state transitions.

the initiative to ask for clarification and confirmation from users. This design promotes more proper decision-making, improved responsiveness to user commands, and increased adaptability to various situations. For example, if a user asks the bot to find an apple in the fridge, and the bot has previously observed a closed fridge, it can utilize its belief state to deduce that it must first open the fridge to locate the apple. This multi-source belief update approach ensures that the bot can effectively combine and utilize all available information, resulting in a more robust and efficient system.

6 Policy

The policy module in SEAGULL is designed to handle various types of user interactions and decide an appropriate course of action based on the current situation. The module consists of a policy controller and three distinct policies: an instruction-following policy, an error-handling policy, and a question-answering policy.

6.1 Policy Controller

The policy controller is the central component responsible for determining which policy should be used and managing the transitions between different policies. It examines the current state, user input, and interaction history to decide the most suitable policy to handle the given situation. The policy controller ensures that the bot can dynamically adapt to the user's commands and seamlessly switch between policies as needed.

6.2 User-Engaged Instruction Following

As instruction following is the core of the Arena games, the instruction following policy is the crucial component of the SEAGULL system. This policy will be activated when a user provides a directive to the bot. This policy is implemented as a state machine, as illustrated in Figure 3. The policy consists of several interconnected states, designed to handle different types of goals in a flexible and efficient manner.

In the **User Input Processing** state, the user's input utterance is processed by the NLU module (Section 3.1) and transformed into a sequence of goal states representing the user's directive. Based on the identified user intentions, the policy proceeds to the **Assigning** state, where it routes the goal request to an appropriate subsequent state to handle different types of goals.

For example, the **Object-Free Navigation** sub-policy is employed for simple navigation commands that do not involve object interaction, such as "move forward" or "go to the robotics lab." In these cases, the bot focuses on navigating through the environment without the need for grounding or object manipulation.

On the other hand, the **Object Interaction** sub-policy is activated for goals involving object manipulation, which encompasses locating objects, moving close enough to render them interactable, and executing actions upon them. Consequently, most directives necessitate the bot's interaction with at least one object, while some high-level goals may demand interactions with multiple objects (e.g., "fix the bowl" requires locating and picking up the bowl, finding a time machine, placing the bowl into the time machine, and turning it on). Since performing actions necessitates fulfilling certain preconditions (e.g., opening the door of the time machine before placing an object inside), it is crucial for the bot to plan its actions. The planning algorithm we employed will be discussed in greater detail in Section 6.2.1.

The **Task Confirmation** and **Task Resumption** sub-policies come into play when handling high-level task goals that require multiple actions to complete. These sub-policies enable the bot to divide complex tasks into smaller, manageable steps by asking the user whether they wish to continue or not. For example, if the user requests the bot to "repair the bowl," the bot may first confirm if the user wants to proceed with the repair, and then execute the necessary actions in a step-by-step manner.

In situations where the bot identifies an unachievable goal, the **Unachievable Goal Feedback** sub-policy is triggered. This sub-policy leverages the bot's built-in domain knowledge to identify and reject goals that are not feasible due to mismatched object affordances and required actions. For instance, the bot will recognize that it cannot "pick up a refrigerator" as it is too heavy to be lifted.

Lastly, the **Incomplete Goal Feedback** sub-policy enables the bot to detect incomplete directives, such as when an action is identified without a specified target object. In such cases, the bot may ask the user for clarification before proceeding.

In summary, the instruction following policy combines these states and sub-policies to create a versatile and adaptive approach to handling user directives. By covering a wide range of goal types and addressing various scenarios, the policy ensures that the SEAGULL system effectively follows user instructions while maintaining a seamless and interactive user experience.

6.2.1 PDDL-Based Action Planning

Users sometimes provide multi-step, high-level goals, such as "make me a cup of coffee". This type of goal is not necessarily easily handled by one predetermined series of actions; for example, in some cases the bot may need to empty a coffee mug before using it, or in some cases the coffee mug isn't visible and needs to be found. In fact, there is a large number of cases for this one high-level goal that would be tedious to explicitly account for in a decision tree. In these cases, it is useful to have a way to specify goals without specifying how they should be solved, and to let an optimizer solve them for us. We employ PDDL [10] planning to accomplish this.

PDDL is a planning language that lets us specify a set of objects, associated predicates, and possible actions as a *domain*. PDDL planners are heuristic-based search algorithms that use these domains, along with specified goals and their relevant objects (called *problems*) to come up with optimal-length trajectories of symbolic actions to take. PDDL is a good fit for the Arena setting since Arena actions and object states are designed symbolically. Since PDDL lets us specify arbitrary actions as pairs of preconditions and post-conditions, we can specify both low-level actions that coincide with Arena-compatible actions and high-level actions that encode more complex goals. We continually update our PDDL domain's action set based on interaction data from users.

The PDDL planner is incorporated into our instruction following policy. Because PDDL planning cannot handle epistemic uncertainty particularly well and requires full observability, we do not rely entirely on it for execution: if the PDDL planner is able to produce a solution based on our specification, we execute its plan. If not, we execute our one-step reactive execution policy, where we apply heuristics to predict executable actions given the current state. During the execution of the planned sequence of actions, if the action fails or there is a mismatch in the expected state, we make use of our error-handling policy, as described below.

6.2.2 User-Engaged Action Execution

To enhance user experience, the instruction following policy also incorporates user-engaged action execution. This feature ensures that the bot keeps the user informed about the current goal, upcoming actions, and progress throughout the execution process. Maintaining transparent communication with the user is crucial for building trust and enhancing user satisfaction. For example, while executing a high-level task such as repairing the bowl, the bot will first acknowledge the goal to the user by saying "Okay, I will repair the bowl". Then at each step, it will report the upcoming action by stating "I am now going to the time machine." or "I am placing the bowl to the time machine" etc. Finally, after the bowl is repaired, the bot will inform the user about the completion of the task by saying "The bowl has been fixed!" and ask about the next step. This level of communication allows the user to have a clear understanding of the bot's actions and progress, ultimately improving the overall user experience.

6.3 Next-Step-Suggestion Policy

To further boost user experience, we develop a next-step-suggestion policy that can automatically suggest the next possible action based on the last physical action taken by the bot and the knowledge about object affordance. For example, if the user asks the bot to navigate to a `pickable` object without giving further instructions, the bot will suggest if the user wants to pick up that object. Therefore, the user can simply say "yes" instead of giving a full command to make the bot perform the pick-up action. We implement such suggestions based on object affordance, which means the bot will never suggest anything that is not doable in Arena. Besides suggesting low-lever actions, we also implement several high-level tasks that will be triggered when interacting with certain special objects. For example, when something is placed on a color changer, our bot will ask the user whether they want to change the color of that object, and mention all the options (red, green, and blue) that can be chosen from. We observe from the user feedback that many users are impressed by the next-step-suggestion policy, which is said to greatly improve the game experience.

6.4 Error Handling Policy

The error handling policy is invoked when an action execution error occurs. In such cases, the error handling policy informs the user about the error and assists them in addressing the problem. For instance, if the bot tries to turn on the robot arm that is not powered up, the error handling policy will notify the user of the issue and may suggest turning on the power by saying "Alexa, turn on the power" before attempting to operate the robot arm on again.

6.5 Question-Answering Policy

As the user may not always give direct commands to SEAGULL, the question-answering (QA) policy is activated when the user asks a question or tries to chat with the bot. This policy is designed to answer user inquiries about the Arena game and bot capabilities, and also give thoughtful responses for general chit-chat from the user. For example, if the user asks, "What does the time machine do?" where the target object is a rare artifact, the policy will provide a description of the artifact and propose to find one or point one out if in view. If a user inquires about the bot's capabilities, the policy will describe SEAGULL's capabilities by informing the user how to communicate with it. Similarly, if the user asks how to play or what to do next, the policy will provide a quick game overview or a thoughtful suggestion, e.g., to read the next subgoal or a sticky note. This policy also enables the bot to respond to general chit-chat from the user to greet the user, thank the user, and more while directing their focus back to the task at hand. For example, when the user thanks SEAGULL, it may respond "You're welcome! What should I do next?". Note that the QA policy does not directly output an utterance, but only determines the system dialog act to take. The realization of the dialog act, i.e. the translation from a system dialog act to the system response in the natural language form, is conducted in our Natural Language Generation module which will be introduced in the next section.

7 Natural Language Generation

Natural language generation (NLG) enables SEAGULL to communicate with users. As reliability and speed are important when interacting with users in an online setting, our NLG module is

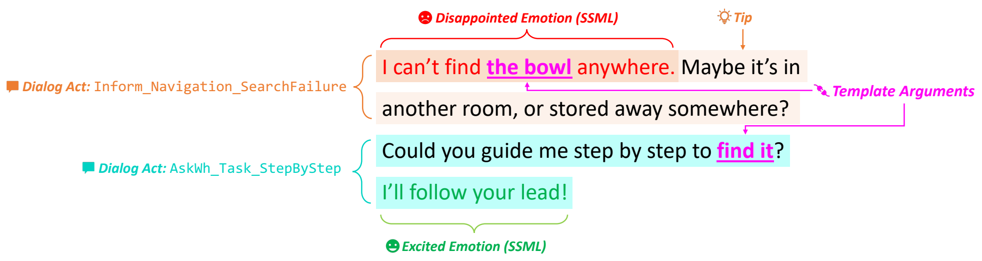

Disappointed Emotion (SSML)
Tip
Dialog Act: Inform_Navigation_SearchFailure
I can't find the bowl anywhere. Maybe it's in another room, or stored away somewhere?
Template Arguments
Dialog Act: AskWh_Task_StepByStep
Could you guide me step by step to find it?
I'll follow your lead!
Excited Emotion (SSML)

Figure 4: Components of an utterance generated by SEAGULL's NLG module, which consists of *templates*, one for each *dialog act* in the utterance, filled with one or more *arguments*, e.g., objects (*the bowl*) or activities (*find it*). Each dialog act can provide additional special *tips* to provide extra information and/or guide the user's response, and Speech Synthesis Markup Language (*SSML*) is used to add emotional tones to templates to make them sound more natural.

entirely template-based. Its flexibility and easy modification enables developers to easily add dialog communication within policies and skills, and add to and refine SEAGULL's utterances where needed.

7.1 System Utterance Components

As illustrated in Figure 4, when SEAGULL says an utterance, it consists of several components which have been defined by us, and can be easily expanded and modified based on user needs or different application areas.

7.1.1 Dialog Acts

System dialog acts (DAs) are the base unit of communicative intents SEAGULL may use to interact with the user. It currently has 112 possible DAs to communicate with the user through,[11] and these are used in 148 contexts throughout SEAGULL's policies and skills to enable communication with the user. Thanks to the NLG module's flexible design, it is simple to add new system DAs where new user needs emerge, requiring only a few lines of code.

DAs are organized through a three-tier system by their *intent*, *context*, and optionally a *sub-context*. There are 9 DA intents, each of which represent a high-level communicative goal toward one of several possible contexts and/or sub-contexts.

`AskWh` intents are used to ask the user a *wh*-question to gather information or complete its understanding of their command, such as where is something, what actions should be applied to which objects, or what is the user's current goal. `AskSelect` DAs are used to ask a user to choose between several options. In some cases, there may be multiple possibilities for SEAGULL to address their request, and we may need to ask them choose. In Arena, such DAs can be interleaved with visual cues, e.g., highlighting an object in front of the user, to improve the clarity of communication. `Confirm` DAs are used to ask the user to confirm some information provided. These DAs are used when SEAGULL has an idea of what to do next, but would like to confirm in case the system has interpreted the user's request incorrectly. `Inform` DAs are used to provide information to the user. `Propose` DAs are used for SEAGULL to take initiative and propose what to do next, either by proposing commands the user can say to SEAGULL, or actions that SEAGULL suggests to take next. `Reply` DAs can offer a quick response to a user's request to answer a question, or indicate that the request has been understood, begun, or completed. `Fallback` DAs are used when SEAGULL fails to successfully execute a user's request. In these cases, SEAGULL may ask the user to repeat themselves, rephrase their request, or try something else, or inform them it has misheard their request or missed part of it. `Social` DAs are used in traditional social and colloquial dialog, e.g., greetings, thanks, encouragements, and interjections; these make its utterances sound more natural and warm to the user. `Others` DAs cover other intents, primarily to indicate the user's request is out of domain.

[11]System DAs and example templates listed in full in Appendix A.2.

7.1.2 Templates

Each DA has several possible *templates*, which are manually added and maintained in SEAGULL. All templates for a specific DA should have about the same meaning, but phrased in different ways. For example, for the DA `Social_Interjection_Apologetic`, which is used when SEAGULL encounters an error or makes a mistake, templates include utterances like "Oops," "Whoops," and "Oh dear." For another example, in the DA `Inform_Object_Room`, templates include "I think {`object_type`} is in {`room`}," "I remember {`object_type`} is in {`room`}," and "{`object_type`}'s in {`room`}," where template *arguments* (described below in more detail) are surrounded by curly brackets.

To enhance the user experience, we focused on creating templates that were natural, diverse, warm, and informative. We achieved this by manually testing each utterance with Alexa's voice and tone, and adding Speech Synthesis Markup Language (SSML)[12] to convey emotions such as excitement and disappointment. Further, we commonly used feedback from users to fine-tune our templates, e.g., in cases where users found the bot too excited or it was talking too much. The streamlined design of NLG makes it easy for anyone on the research team to add or modify templates, and our thorough tests ensure templates are properly formed and convey the expected information before deployment.

7.1.3 Arguments

Some templates require one or more string arguments to form complete utterances. For example, in the templates listed above for `Inform_Object_Room`, we require object type (e.g., *the bowl*) and room (e.g., *the break room*) arguments to fill the template. Based on the Arena environment and SEAGULL's system DAs, there are currently 15 types of arguments supported in the NLG templates, including but not limited to object type, object instance (e.g., "the bowl on the left"), object property (e.g., "broken"), location description (e.g., "in the time machine"), action type (e.g., "grab"), activity description (e.g., "find the time machine"), and room (e.g., "break room"). We use a mapping process, described in Section 7.2, to convert symbolic information from SEAGULL's tracked world state into strings to fill these template arguments. While these template argument types are specifically geared toward the Arena environment, it is simple to add or remove argument types to adapt to other tasks.

7.1.4 Tips

Based on our understanding of the situation and interaction history with the user, we can also append or prepend *tips* to SEAGULL's utterances to provide extra context as needed or guide the user's response. SEAGULL currently supports three types of tips stored along with template data for each DA.

Pre-tips are prepended before an utterance for a specific DA to provide some extra context. For example, when SEAGULL fails to find an object (e.g., "I can't seem to find the bowl here."), we may prepend a pre-tip to tell the user why that may be: "Sorry, I'm still learning my way around here."

Second, *post-tips* give some followup guidance on how to respond to an utterance for a specific DA. For example, SEAGULL often proposes actions it could perform next, e.g., interacting with the color changer: "Shall I change the color of the bowl?" In this case, we may append a post-tip to give the user an idea how to respond to SEAGULL's proposal: "We can choose from red, blue, or green."

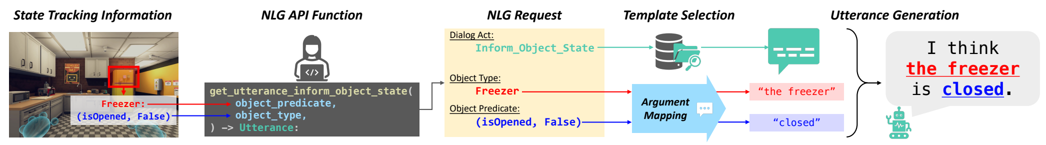
Lastly, *newbie-tips* are written specifically for users who may be new to the game. For example, when SEAGULL mentions the Arena game's subgoals, we may want to provide information on where to find them.

These tips can be manually written for each DA, or instead point to another DA to avoid redundant writing of tips. If other types of informative dialog are needed, it is straightforward to add more types of tips into the NLG template data.

7.2 System Utterance Generation

In order to generate an utterance from these components, we use an NLG *request*, which contains the target DA for the utterance and relevant information to inform the filling of its template arguments. We then perform an automatic many-to-many mapping from the information in the request to the

[12]`https://developer.amazon.com/en-US/docs/alexa/custom-skills/speech-synthesis-markup-language-ssml-reference.html`



```
get_utterance_inform_object_state(
    object_predicate,
    object_type,
) -> Utterance:
```

Figure 5: Overview of how SEAGULL's natural language generation (NLG) module generates an utterance, e.g., for `Inform_Object_State`. A consumer skill or policy will call an NLG API function, passing in tracked state information or other relevant details to generate the utterance. This information will be used to create an NLG request, which is used to randomly select a template (and tips, if needed) and fill its arguments with strings.

template arguments, then select a random template to fill for the target DA. This mapping can be triggered through a user-friendly, automatically generated *API*.

7.2.1 NLG Requests and Mapping

Within an NLG request, several types of information may be provided to map into template arguments. Object-related template arguments can be mapped from object types, specific object instances, object predicates (i.e., physical states), or manually specified strings. Action-related template arguments can be mapped from action types, fully-specified actions (i.e., action type with direct and/or indirect objects), one of several common task classes (e.g., *find object* or *fire the laser*), a goal state (i.e., to mention the action that could achieve that state), or manually specified strings. Room-related template arguments are mapped from a set of room names in the Arena environment.

To convert object, action, and room classes into strings, we use manually defined names for each class, and generate comma-delimited lists of these names when template arguments require a list of them. When SEAGULL needs to describe a list of objects, it must distinguish them to avoid ambiguity. This is done by identifying a unique feature of each object: an object's type, relative position in the visual frame, or physical state predicates can be used to identify it.

7.2.2 NLG API

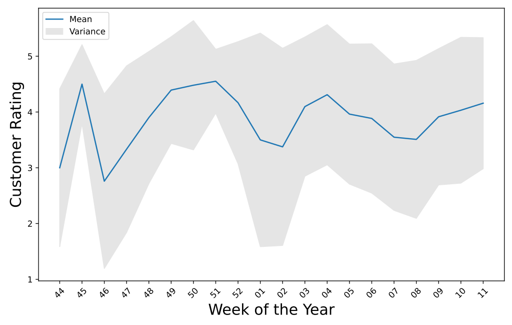
To enable easy generation of utterances from policies or skills, we automatically generate a user-friendly API from the template data described in Section 7.1. Within each API function, we define the required request parameters to generate an utterance for each possible DA, with all possible data types and thorough documentation. API functions return special class objects for utterances which can be automatically concatenated to each other, and track the mentioned objects, states, actions, directions, rooms, and subgoals within utterances. This enables developers to easily call the NLG API from other different policies and skills to build SEAGULL's engaging dialog communications. In turn, this helps streamline the development of new features for SEAGULL to continually improve its capabilities.

8 Evaluation

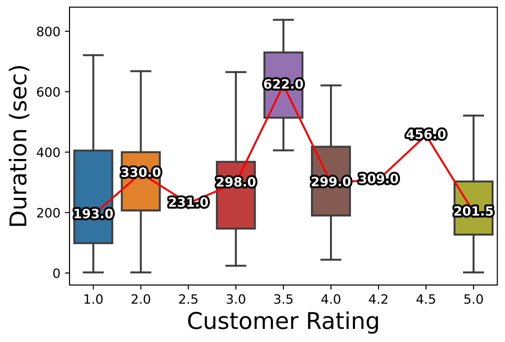
The evaluation for SimBot is a complex task. While traditional embodied AI tasks usually adopt automatic and non-interactive evaluation pipelines that focus on task success rate, SimBot uses interactive, human-in-the-loop evaluation that focuses more on the holistic user experience. Specifically, after each SimBot session, the user rates the experience from 1.0 to 5.0. This rating, although not perfect, is a direct and holistic reflection of the user's impression as a system need to be capable, stable, responsive, natural, and smart to win user satisfaction. Below, we share a quantitative analysis of the overall user rating, an evaluation of task success rate in the offline environment and qualitative observations made from user feedback.

8.1 Quantitative Analysis on User Ratings

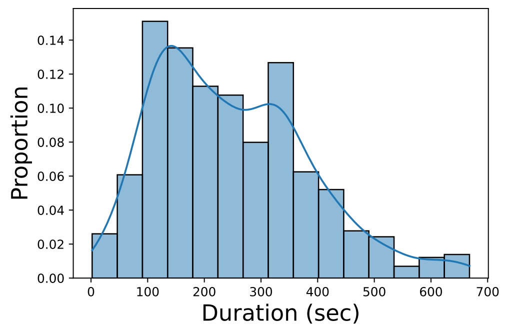
Figure 6a gives an overall trend of ratings we received from users. We observe an overall increasing but fluctuating trend. There are many factors contributing to the fluctuating scores. We share some observations in the section below. Figure 6c and 6d show the distribution of conversation duration



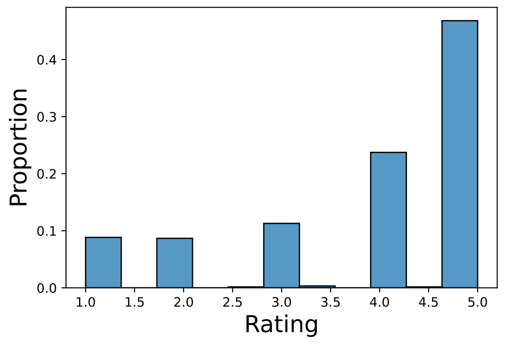
(a) Ratings over time



(b) Rating distribution (numbers are medians).



(c) Conversation duration (in seconds) distribution.



(d) user rating distribution.

Figure 6: Rating distribution from October 31, 2022 to March 20, 2023.

and ratings. As we can see, most users end the conversation between 100 to 300 seconds and many gave 4.0 to 5.0. From Figure 6b users who gave 5.0 tend to have shorter and quicker conversations, suggesting finishing the game quickly is a favorable behavior. For users who gave 1.0, the lowest score, the median duration is 193 seconds, shorter than user conversations of 5.0 sessions, but exhibiting larger variance, suggesting that users may be frustrated early in the game and just want to quit, or have been stuck for very long and eventually give up unhappily. The sessions received 3.5s are also interesting: They have the longest duration of 622 seconds - more than 10 minutes of interactions. This usually happens when users use primitive commands (likely because higher-level commands like find failed) to control the bot to complete long-horizon tasks - they are still able to complete the task but the experience is not great.

8.2 Trends Observed from User Feedback

Here are some hypotheses we qualitatively summarized from reading users' textual feedback.

- Users tend to speak highly about bot's capability of completing the task fast and stable.
- Users tend to give high ratings when our bot was able to understand every command.
- Users appreciated the more natural and fun tones we added using Alexa SSML.
- Users appreciated that our bot gave hints on how to get unstuck.
- Users seem to get frustrated when they were unable to make progress on the game and were not provided useful hints to move forward.
- Users seem to get frustrated when some seemingly simple tasks, e.g. toggling a light switch when the switch is right in front of the bot, were unable to be fulfilled.
- Users noticed when our bot carried a long latency and appreciated shortened latency.

Method	MSR (%)
Neural-Symbolic [5]	18.19%
VL Model [5]	22.80%
SEAGULL	30.98%

Table 1: Experimental results of offline instruction following the validation split. MSR stands for mission success rate.

8.3 Offline Instruction Following Success Rates

Table 1 displays SEAGULL's offline instruction-following performance on the validation split of the Arena benchmark[13]. SEAGULL achieves a mission success rate of 30.98%, which is 8.18% higher than the baseline VL model from [5].

To better understand the performance, we make a closer examination of the model predictions and identified several typical failure cases. First, our bot performs poorly in distinguishing between objects of the same type, which poses challenges when the bot needs to deliver an item to a specific desk or locate something within a particular cabinet. It also struggles in understanding object references with fine-grained spatial indicators, such as "the desk by the door". Secondly, our bot has some problems detecting and finding small objects, where it misses some key object to complete the task. Thirdly, there is a domain gap between online interaction which SEAGULL is designed for, and the instruction-following formulation in the offline evaluation. For example, sometimes our bot will pause and ask the user whether to proceed, which cannot be provided during offline evaluation.

Our analysis has highlighted the need to enhance SEAGULL's object reference understanding capabilities and keep improving the vision model, in order to improve its overall performance. As part of our future work, we will focus on refining the natural language and vision models, addressing the issues identified in our evaluation, and working towards a more robust and accurate system for instruction-following tasks in the Arena benchmark.

9 Conclusions

In summary, SEAGULL takes a step towards embodied AI with Arena by building and integrating several substantial components:

1. Speech-to-symbol natural language understanding (NLU) pipeline from ASR preprocessing to entity extraction, intent detection, context modeling, and semantic parsing.
2. Data augmentation processes powered by large language models, enriching the language diversity in training data.
3. Multi-faceted, persistent state tracking to enable SEAGULL to remember important details of the interaction history, task progress, and current states of the agent and objects, continuously updated as the agent performs actions, observes the world visually, and communicates with the user through language.
4. Robust policies for instruction following, error handling, and questions answering to control the agent's execution and dialog, allowing continuous improvement and augmentation with new skill development.
5. Scalable and versatile template-based natural language generation (NLG) database and user-friendly API enabling diverse yet controllable language communication.

The Amazon Simbot Challenge provided us a unique, collaborative, and engaging experience to develop SEAGULL - an embodied AI agent that strives to listen, talk, perceive, reason, plan, and act in the environment. SEAGULL connects lower-level perception and action with rich symbolic structures at various levels of processing. Such transparency plays an important role in bringing

[13]We use the new Arena built to run the evaluation since SEAGULL requires correct depth information in its perception skill.

humans and the agent to a common understanding of tasks, goals, and situations. SEAGULL will serve as an infrastructure for our future work that explores continual learning and adaptation for embodied human-AI collaboration and communication.

References

[1] Yonatan Bisk, Ari Holtzman, Jesse Thomason, Jacob Andreas, Yoshua Bengio, Joyce Chai, Mirella Lapata, Angeliki Lazaridou, Jonathan May, Aleksandr Nisnevich, Nicolas Pinto, and Joseph Turian. Experience grounds language. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8718–8735, Online, November 2020. Association for Computational Linguistics.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Joyce Y Chai, Qiaozi Gao, Lanbo She, Shaohua Yang, Sari Saba-Sadiya, and Guangyue Xu. Language to action: Towards interactive task learning with physical agents. In *IJCAI*, pages 2–9, 2018.

[4] Bowen Cheng, Ishan Misra, Alexander G Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention mask transformer for universal image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1290–1299, 2022.

[5] Qiaozi Gao, Govind Thattai, Xiaofeng Gao, Suhaila Shakiah, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zheng, et al. Alexa arena: A user-centric interactive platform for embodied ai. *arXiv preprint arXiv:2303.01586*, 2023.

[6] Golnaz Ghiasi, Yin Cui, Aravind Srinivas, Rui Qian, Tsung-Yi Lin, Ekin D Cubuk, Quoc V Le, and Barret Zoph. Simple copy-paste is a strong data augmentation method for instance segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2918–2928, 2021.

[7] Agrim Gupta, Piotr Dollar, and Ross Girshick. Lvis: A dataset for large vocabulary instance segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5356–5364, 2019.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.

[10] Drew McDermott, Malik Ghallab, Adele E. Howe, Craig A. Knoblock, Ashwin Ram, Manuela M. Veloso, Daniel S. Weld, and David E. Wilkins. Pddl-the planning domain definition language. 1998.

[11] Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramithu, Gokhan Tur, and Dilek Hakkani-Tur. Teach: Task-driven embodied agents that chat. *arXiv*, 2021.

[12] Rolf Pfeifer and Fumiya Iida. Embodied artificial intelligence: Trends and challenges. In *Embodied Artificial Intelligence: International Seminar, Dagstuhl Castle, Germany, July 7-11, 2003. Revised Papers*, pages 1–26. Springer, 2004.

[13] Yichi Zhang, Jianing Yang, Jiayi Pan, Shane Storks, Nikhil Devraj, Ziqiao Ma, Keunwoo Yu, Yuwei Bao, and Joyce Chai. DANLI: Deliberative agent for following natural language instructions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1280–1298, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

A Appendix

A.1 Full Description of User Dialog Acts

- `Inform` is a category in which the user informs some instructions or descriptions to the bot.
  - *Inform_Directive_Mani* indicates a goal state the bot needs to complete.
  - *Inform_Directive_Navi* indicates where the bot should navigate.
  - *Inform_Directive_CommonTask* indicates a common task the bot should perform.

  - *Inform_Task_Goal* specifies the main goal of a task.
  - *Inform_Task_Subgoal* specifies a subgoal within a larger task.
  - *Inform_Task_Done* informs that a specific task has been completed.
  - *Inform_Task_Notdone* informs that a specific task has not been completed.
  - *Inform_Task_Numsg* specifies the number of messages related to a task.
  - *Inform_Task_Numsgdone* specifies the number of messages related to a task that have been completed.
  - *Inform_Task_Numsgleft* specifies the number of messages related to a task that are left to complete.
  - *Inform_Task_Tool* provides information about a tool needed for a task.
  - *Inform_Object_Location* provides the spatial location of an object.
  - *Inform_Object_State* provides the current state of an object.
  - *Inform_Object_Reference* provides a reference to an object.
  - *Inform_Object_Type* provides the predefined type of an object.
- `Reply` is a category in which the user replies to the bot's previous message.
  - *Reply_No* is a negative response.
  - *Reply_Yes* is an affirmative response.
  - *Reply_Notsure* is an uncertain response.
  - *Reply_Select* is a selection from given options.
  - *Reply_None* indicates none of the options apply.
  - *Reply_Deny* is a denial or contradiction.
  - *Reply_Acknowledge* acknowledges a statement or action.
- `Ask` is a category in which the user asks a question.
  - *Ask_Agent_Location* asks about the bot's location.
  - *Ask_Agent_Inventory* asks about the bot's inventory.
  - *Ask_Agent_Capability* asks what the bot can do in the challenge.
  - *Ask_Current_Task* asks about the current task the bot is performing.
  - *Ask_Check_Objectstate* asks the bot to check the state of an object.
  - *Ask_Object_Location* asks about the location of an object.
  - *Ask_How_Task* asks how a specific task should be completed.
  - *Ask_Knowledge_Object* asks for information about a specific object.
  - *Ask_Prompt* asks the bot to provide a prompt or suggestion.
- `Social` is a category for social and chit-chat interactions.
  - *Social_Greeting_Opening* is an opening greeting.
  - *Social_Greeting_Closing* is a closing greeting.
  - *Social_Thanks_Initiate* is an expression of gratitude.
  - *Social_Thanks_Response* is a response to an expression of gratitude.
  - *Social_Apology_Initiate* is an expression of apology.
  - *Social_Apology_Response* is a response to an expression of apology.
  - *Social_Praise* is an expression of praise or compliment.
  - *Social_Complaint* is an expression of dissatisfaction or complaint.
- `Profinity` is a category for profane or inappropriate content.
  - *Profinity_Sentive* is for sensitive content.
  - *Profinity_OutOfDomain* is for out-of-domain content.
  - *Profinity_Offensive* is for offensive content.
  - *Profinity_EndOfGame* is for content signaling the end of the game or interaction.
  - *Profinity_HarmfulToBot* is for content that may be harmful to the bot.
- `Others` is a category for miscellaneous intents.
  - *Request_Repeat* asks the bot to repeat a previous message.
  - *Incomplete* indicates an incomplete message or thought.
  - *Nonsense* indicates a message that doesn't make sense or is unrelated.
  - *Others* is a catch-all for other intents not covered by the previous categories.

A.2 Full Description of System Dialog Acts

Below we list all system dialog acts (DAs) and example utterances for each, excluding SSML tags.

AskWh: DAs used to ask the customer a *wh*-question to gather information or complete its understanding of their command, such as where is something, what actions should be applied to which objects, or what is the customer's current goal.

- AskWh_Object_Receptacle: "Where can I find {object_type}?"
- AskWh_Object_Room: "In which room can I find {object_type}?"
- AskWh_Action_Target: "{action_type} what?"
- AskWh_Action_Receptacle: "Where should I {action_type} {object_type}?"
- AskWh_Action_Tool: "What can I use to {action}?"
- AskWh_Action_TurnDirection: "Which direction should I turn?"
- AskWh_Action_TurnDirectionTowardObject: "Which direction should I turn to find {object_type}?"
- AskWh_Object_Type: "What do you want me to do with {object_type}?"
- AskWh_Navigation_Target: "Where did you want me to go?"
- AskWh_Task_Goal: "What is our mission goal?"
- AskWh_Task_FirstSubgoal: "What is our first subgoal?"
- AskWh_Task_NextSubgoal: "What is our next subgoal?"
- AskWh_Task_FirstStep: "What's the first step?"
- AskWh_Task_FirstStepTowardGoal: "What is the first step to {action}?"
- AskWh_Task_NextStep: "What should I do next?"
- AskWh_Task_NextStepTowardGoal: "What is the next step to {action}?"
- AskWh_Task_StepByStep: "<prosody rate='115
- AskWh_Task_AlternatePlan: "Any other ideas?"
- AskWh_Task_NumSubgoals: "How many subgoals do we have to complete?"
- AskWh_Task_NumSubgoalsDone: "How many subgoals have we completed?"
- AskWh_StickyNote_Text: "What does the sticky note say?"

AskSelect: DAs used to ask a customer to choose between several options. In some cases, there may be multiple possibilities for SEAGULL to address their request, and we may need to ask them choose. In Arena, such DAs can be interleaved with visual cues, e.g., highlighting an object in front of the customer, to improve the clarity of communication.

- AskSelect_Object: "Do you mean {objects}?"
- AskSelect_Object_Receptacle: "Is {object_type} {receptacle_objects}?"
- AskSelect_Object_Room: "Is {object_type} in {rooms}, or somewhere else?"
- AskSelect_Room: "Do you mean {rooms}?"
- AskSelect_Action_Target: "Should I {action_type} {objects}?"
- AskSelect_Action_Receptacle: "Should I {action_type} it {receptacle_objects}?"
- AskSelect_Action_TurnDirection: "Should I turn {directions}?"
- AskSelect_action_type: "Should I {action_types} {object_type}?"
- AskSelect_Task_Ongoing: "Are we working on {subgoals}?"

Confirm: DAs used to ask the customer to confirm some information provided. These DAs are used when SEAGULL has an idea of what to do next, but would like to confirm in case the system has interpreted the customer's request incorrectly.

- Confirm_Object: "Do you mean {object_type}?"
- Confirm_Object_State: "Is {object_type} {object_predicate}?"
- Confirm_Object_Room: "Is {object_type} in {room}?"
- Confirm_Object_Type: "Is this {object_type}?"
- Confirm_Action: "Did you want me to {action}?"
- Confirm_Action_Target: "Is this the right {object_type} to {action_type}?"
- Confirm_Action_Receptacle: "Should I {action_type} it {receptacle_object}?"
- Confirm_Task_Goal: "Let me double check, you asked me to {action} - right?"
- Confirm_Task_NextStep: "Is the next step to {action}?"
- Confirm_Task_GoalUnchanged: "We're still trying to {action}, right?"
- Confirm_Task_GoalDone: "Is our mission goal marked as complete?"
- Confirm_Task_SubgoalDone: "Is {subgoal} marked as complete?"

Inform: DAs used to provide information to the customer.

- Inform_CurrentRoom: "I think we're in {room}."
- Inform_Inventory_Object: "I'm holding {object_type}."
- Inform_Inventory_Empty: "I'm not holding anything."
- Inform_Object_Type: "That's {object_type}."
- Inform_Object_Receptacle: "I think {object_type} is {receptacle_object}."
- Inform_Object_Room: "I think {object_type} is in {room}."
- Inform_Object_State: "I think {object_type} is {object_predicate}."
- Inform_Object_Purpose: "{object_type} is used to {action}."
- Inform_Navigation_Room: "Heading to {room}."
- Inform_Navigation_SearchTarget: "Let me go and find {object_type}."
- Inform_Navigation_SearchUpdate: "Checking over here!"
- Inform_Navigation_SearchSuccess: "There it is!"
- Inform_Navigation_SearchFailure: "I can't seem to find {object_type}."
- Inform_Task_Ongoing: "I am {action}."
- Inform_Task_Done: "We {action}."
- Inform_Task_Plan: "I will {actions}."
- Inform_Task_PlanNotFound: "I'm not sure how to {action}."
- Inform_Action_Incapable: "Looks like I'm not able to {action}."
- Inform_Exception_UnsupportedAction: "I'm not able to do that."
- Inform_Exception_UnsupportedNavigation: "I can't move that way."
- Inform_Exception_AlreadyHoldingObject: "I can only hold one thing at a time."
- Inform_Exception_OutOfRange: "I can't reach {object_type}."
- Inform_Exception_OutOfSight: "I can't see {object_type} now."
- Inform_Exception_KilledByHazard: "We should keep an eye out for dangerous hazards like that."
- Inform_Exception_Other: "That didn't work."
- Inform_GameInfo_Goal: "The mission goal can be found in the first line of the upper left corner."
- Inform_GameInfo_Subgoal: "The subgoals are listed in the upper left corner, under the mission goal."
- Inform_GameInfo_GoalStatus: "You can see the status of a goal or subgoal by looking at the checkbox next to it."
- Inform_GameInfo_MiniMap: "There is a mini-map of the lab in the upper right corner of the screen."
- Inform_GameInfo_StickyNote: "Just say: Alexa, read the sticky note."
- Inform_GameInfo_RoomName: "You can see the names of rooms on the minimap."
- Inform_GameInfo_Score: "Pay attention to the game score under your mini-map."
- Inform_GameInfo_Background: "<prosody rate='115
- Inform_Game_Success: "Yay! We completed the mission. Great job."
- Inform_Game_Failure: "Bummer. We couldn't complete the mission. Let's try again soon."
- Inform_Tutorial_Communicate: "<prosody rate='110
- Inform_Tutorial_StickyNote: "To find other sticky notes, look for shining green dots on themini-map in the upper right."
- Inform_Tutorial_Hazard: "By the way, some parts of this lab are dangerous. Please help me avoid any hazards so I don't get hurt."

Propose: DAs used for SEAGULL to take initiative and propose what to do next.

- Propose_Action: "Shall we {action}?"
- Propose_NextStep: "Do you want me to {action}?"
- Propose_NextStep_FindObject: "<prosody rate='100
- Propose_NextStep_OperateColorChanger: "Wanna change the color of {object_type}?"
- Propose_Command: "Just say: Alexa, {actions}."
- Propose_ReadSubgoal: "You can always try to tell me one of the subgoals in the upper left."
- Propose_Tutorial_Communicate: "<prosody rate='115
- Propose_NextGame: "I think we're hitting a wall here. Wanna try another task?"

Reply: DAs that can offer a quick response to a customer's request to answer a question, or indicate that the request has been understood, begun, or completed.

- `Reply_Yes`: "Yes!"
- `Reply_No`: "No."
- `Reply_NotSure`: "Not sure."
- `Reply_Acknowledge`: "Got it - "
- `Reply_Doing`: "I'm on it!"
- `Reply_NotDoing`: "We won't do that."
- `Reply_Done`: "Done!"
- `Reply_Going`: "On my way!"

Fallback: DAs used when SEAGULL fails to successfully execute a user's request. In these cases, SEAGULL may ask the customer to repeat themselves, rephrase their request, or try something else, or inform them it has misheard their request or missed part of it.

- `Fallback_AskRepeat`: "Could you repeat that?"
- `Fallback_NotUnderstand`: "I don't get it."
- `Fallback_AskRephrase`: "Could you say that in a different way?"
- `Fallback_TrySomethingElse`: "Maybe we can try something else."
- `Fallback_Mishear`: "I missed that."
- `Fallback_PartialMishear`: "I missed something."

Social: DAs used in traditional social and colloquial dialog. Such utterances are important when interacting with humans, and can help the customer better trust SEAGULL. SEAGULL can make social utterances for greetings, thanks, apologies, pausing its speech (e.g., "Umm..."), encouragement (i.e., to praise the customer when successes occur), or more colloquial interjections (e.g., "Wow!") to make its utterances sound more natural to the customer.

- `Social_Greeting_Opening`: "Hello!"
- `Social_Greeting_Closing`: "Goodbye."
- `Social_Thanks_Initiate`: "Thanks!"
- `Social_Thanks_Response`: "You're welcome!"
- `Social_Apology_Initiate`: "I'm sorry."
- `Social_Apology_Response`: "That's okay."
- `Social_SpeechPause`: "Umm..."
- `Social_Encourage`: "Great job!"
- `Social_Interjection_Apologetic`: "Oops!"
- `Social_Interjection_Confused`: "Hmm - "
- `Social_Interjection_Informative`: "Hey - "
- `Social_Interjection_Hurt`: "Ouch - "
- `Social_Interjection_Excited`: "Awesome!"
- `Social_Interjection_Operating`: "Here we go!"
- `Social_Interjection_Attention`: "Hey - "

Others: DAs for other intents, primarily to indicate the customer's request is out of domain.

- `Others_Sensitive_Response`: "I don't understand."
- `Others_OutOfDomain_Response`: "I don't understand."
- `Others_Offensive_Response`: "I don't understand. Please try something else."
- `Others_EndOfGame_Response`: "Thanks for your help!"
- `Others_HarmfulToBot_Response`: "I don't understand."

A.3 Full Description of LLM Augmentation

We use a prompt template as follows to collect synthetic user instructions from large language models.

```
You are collaborating with a robot to complete a task. Your goal is to
instruct the robot to [GOAL STATE] with natural language commands. Write
[NUMBER] other possible commands of the same purpose without introducing
additional context. Use various forms of sentences and everyday spoken
language. An example command is "[EXAMPLE]".
```

Some examples towards a goal state `Apple.IsPickedUp` are presented as follows:

```
Can you pick up the apple, please?
Please pick up the apple for me.
The apple needs to be picked up. Can you do it?
Pick up the apple and bring it here.
To complete this task, please pick up the apple.
I need you to pick up the apple for me.
The apple must be picked up. Can you help me?
Pick the apple up and place it on the table.
If you don’t mind, please pick up the apple.
Could you please pick up the apple and give it to me?
```