# Towards a Robust and Generalizable Embodied Agent

**Chan Hee Song[*], Jiaman Wu[*], Ju-Seung Byeon, Zexin Xu, Vardaan Pahuja, Goonmeet Bajaj, Samuel Stevens, Ziru Chen, Yu Su**

The Ohio State University

{song.1855, wu.5686, byun.83, xu.2460, pahuja.9, bajaj.32, stevens.994, chen.8336, su.809}@osu.edu

## Abstract

This paper introduces SalsaBot, an embodied agent designed for the first Alexa Prize Simbot Challenge. The primary aim of SalsaBot is to assist users in completing a game within a virtual environment by providing a consistent user-centric experience, which requires the agent to be capable of handling various types of user interactions. Individuals from various backgrounds interact with SalsaBot using Automatic Speech Recognition (ASR), and SalsaBot is designed with a user-centered approach to ensure a frustration-free experience. To ensure a great user experience, SalsaBot is equipped with robust macros, an explicit object memory, and a state-aware dialogue generation module. Additionally, in order to promptly fulfill user expectations, we have developed a web monitoring system that enables effortless collection, visualization, and analysis of user interactions. Our efforts and findings demonstrate that our SalsaBot design has contributed to a continuous enhancement in user experience throughout the competition.

## 1 Introduction

We present SalsaBot, an embodied agent developed for the first Amazon Alexa Prize Simbot Challenge. SalsaBot operates in complex partially-observable environments with only visual perception of the environment, and collaborate with human users to accomplish complex tasks while using natural language as the main means of communication. The users are real Alexa users with diverse backgrounds. The tasks are goal-oriented and formulated as games: At the beginning of each game session, the human users are given a high-level goal and a list of subgoals, and they engage with the agent in a multi-turn task-oriented dialogue via voice to command the agent to achieve the goal. The key objective metrics are *success rate*, *i.e.*, how often is the final goal achieved, and *speed*, *i.e.*, how many turns of interaction it takes to achieve the goal. These also lead to an overall rating from the user, which is more subjective.

This is a fairly complex and ambitious setting. No off-the-shelf method for embodied instruction following would work to a satisfactory degree or can be adapted with reasonable efforts. Novel methods rooted in a deep understanding of the unique and fundamental challenges of this advanced setting have to be developed. We believe the unique challenges, compared with model development on static academic datasets like ALFRED (Shridhar et al., 2020), include the following:

1. The diversity in users' mental models. Real users show a great diversity in their mental model of the game. Some power users are pretty familiar with the agents' capabilities and limitations, while many novice users may enter the game knowing little more than nothing.

[*]Team Leads. Equal Contribution.

1st Proceedings of Alexa Prize SimBot.

An robust agent should cater to users spanning this full spectrum. It should provide advanced capabilities to assist power users to accomplish tasks quickly, but also provide sufficient guidance so novice users do not easily get lost.

2. The diversity in user commands. For static datasets, the language patterns are usually consistent between training and test data. For an agent open to real users all over the world, however, users can truly say anything in any way. Not only does this manifest in the large number of different ways users may use to express the same thing, but it also manifests in the different levels of abstraction: Some users may prefer a *procedural* way and mostly use primitive actions, while other users may be more *declarative* and give high-level commands, e.g., *"Get milk from fridge"* entails a sequence of actions including going to the fridge, opening it, picking up milk, and closing the fridge.
3. The high stakes of planning errors. When developing models on static academic datasets, we may applaud for a planning accuracy of 78% on validation dataset from our advanced neural network models. However, that is not enough for a real user-facing agent. Planning errors, *e.g.*, pouring coffee without first picking up the coffee pot, or predicting an erroneous extra action, could surprise the user or sometimes lead to catastrophic, irreversible errors, compromising the trustworthiness of the agent.

We design SalsaBot to be *user-centric*, delivering a robust and consistent user experience regardless of their mental model about the agent or language usage patterns. So far we have focused on the following innovations:

1. We perform a systemic analysis of user interactions to understand the different levels of user needs and patterns. Based on the understanding, we propose a novel construct called *macros*, a high-level action that encompasses all the essential ingredients for achieving a certain goal, *e.g.*, picking up some object from a receptacle. It includes all the possible scenarios (e.g., does the receptacle have a door? Is the door closed? What if the object can not be found in the receptacle?), the corresponding action sequences for each scenario, exception handling, and appropriate agent response generation. Macros provide a powerful and flexible tool for robust and generalizable agent planning that caters to different levels of abstraction.
2. Finding objects quickly and accurately is central to the success of SalsaBot. We develop a powerful and user-friendly *object localization* method that includes a *memory* module to allow going back to seen objects and a *find* module to systematically search for unseen objects.
3. Human-agent collaboration is not a one-way street. Agent responses are crucial in communicating its understanding of both the user intents and the current situation, as well as providing proactive guidance to the user when needed. We design our dialogue response module to generate state-aware dialog responses and proactively suggest or execute appropriate actions when necessary. We also diversity our responses to make it more engaging.
4. A unique aspect of the Alexa Prize challenge is we are able to constantly receive timely user feedback. Learning from user feedback is crucial to user-centric designs. We devote significant efforts in this, including developing a sophisticated web monitor for monitoring and analyzing real-time user traffic.

Finally, we take generalizability (to new objects/environments/games/users) to our heart. We never use any ad-hoc design that is specific to a certain game, even though the short-term benefit on user ratings could be tempting. Instead, we always focus on developing generalizable solutions such as macros. As a result, every time we introduce a major feature, our rating reliably increases (Section 7). This is most evident on the new games after the code release. Due to our generalizable design, *our daily ratings are consistently in the top 5 and sometimes at the top 1.* Due to time constraints, there are still many features that have been planned but have not been implemented yet. We plan to continue our development to deliver a truly collaborative, intelligent, and robust agent.

## 2 System Overview

We develop SalsaBot (Figure 1) on the Arena framework (Gao et al., 2023) provided by Amazon Alexa. A turn begins with user input to the Amazon Echo Show or Fire TV device and consists of a

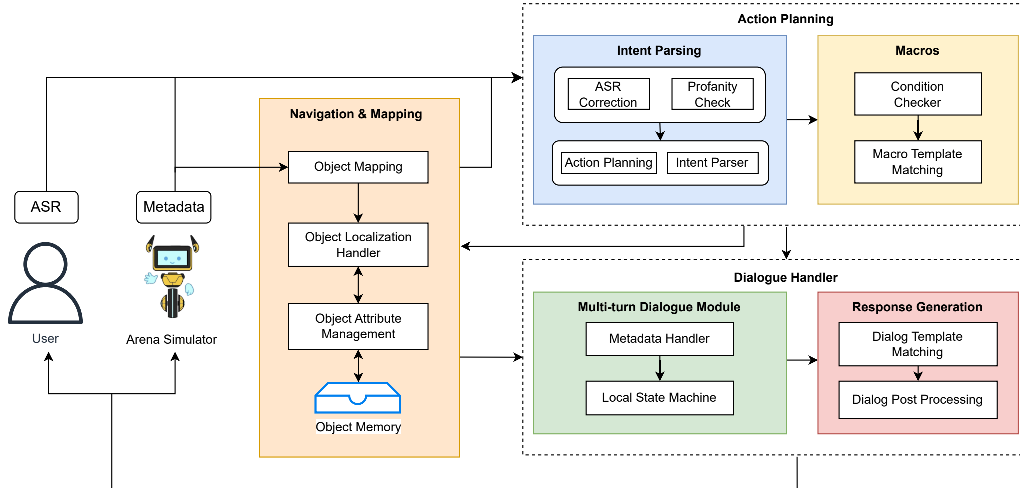


Figure 1: Overall System Design

series of interactions between SalsaBot and the Arena simulator, which concludes with a dialogue and an activation of the microphone to accept additional user input. During each turn, the user can communicate with SalsaBot using their voice commands. Alexa's Automatic Speech Recognition (ASR) transcribes the user's speech into a text utterance.

When user input is received, our SalsaBot performs initial processing through our natural language understanding (NLU) module, which contains three components: *ASR correction*, *profanity checking*, and *intent classification*. Following correction by the ASR module and verification by the profanity checker, the intent classification module identifies the responsible module to execute the user's intent.

SalsaBot categorizes intents into two types: *macros* (Section 4), which abstract a sequence of actions, and *primitive actions*, which allow users to directly control the agent's actions. Because a single user utterance can contain a mixture of multiple intents, SalsaBot can detect and execute multiple intents simultaneously.

Each intent is handled by our action planning module (Section 3). Depending on the intent type, the action planning module invokes different sub-modules to execute the intent. Some intents require an action sequence (*e.g.*, macros) or dialogue with the users (*e.g.*, object disambiguation). In all cases, our response generation module (Section 5) generates appropriate responses. Furthermore, we use the response generation module to provide users with constructive feedback on failed actions and suggest suitable actions based on the environmental context.

Finally, we present our approach to rapidly learn from user feedback in Section 6.

## 3 Action Planning

This section focuses on the non-primitive action planning, such as object localization and navigation to specific rooms. Such actions serve as the foundation for performing various tasks. It is crucial to bridge the gap between user expectations and our agent's abilities. Users issue commands to the embodied agent in various ways, even when executing these basic actions. Section 3.1 discusses the scenario in which a user asks the bot to go to an object in the current scene or has previously encountered it, while Section 3.2 addresses instances in which the robot needs to search for the target object. Section 3.3 outlines our approach to resolving situations in which multiple objects create ambiguity. The entire flow of action planning is in Figure 2. In addition, Section 3.4 describes our approach in handling primitive object interaction actions such as *Pick up* or *Toggle*.

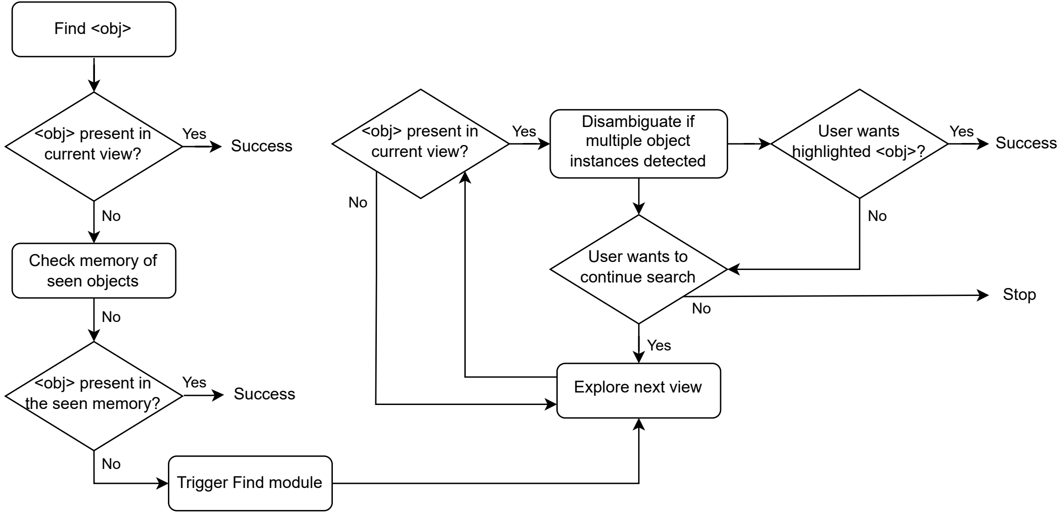


Figure 2: Workflow of the *Find* module

### 3.1 Situation-aware Action Planning — *Go* Module

This section deals with planning when our agent is aware of the current situation. This case can be categorized into two sub-cases. The first sub-case is when the target object is within the current view, while the second sub-case is when our bot has seen the object in the past while the user progressed with the task. Other than normal action planning, the macro module in Section 4 serves as a complementary component to action planning.

The first sub-case is relatively simple. Since the target object is in the current view, we can use the *GoTo* system command to move to the target object. However, the second case is more complicated. For example, suppose that the user wants the robot to go back to a cake placed on a shelf before. Our robot requires to have the location information to execute the command. But what if the robot has seen the cake in a different location? Telling the robot to *"Go back to the cake"* might land you in the wrong place. Based on user feedback, we have devised a rule to mitigate the issue of ambiguity. Unless explicitly directed by the user to go to the target object within a specific room, our robot basically goes to the target object closest to its present location. It is important to note that this simple rule has limitations and several counterexamples exist. In certain cases, our robot may approach a different object located in the another room. To address this challenge more comprehensively, we plan to adopt a more intricate approach by leveraging detailed information like the depth values of detected objects in the image.

Upon generating a planned action, the pair comprising the action and its relevant objects, as outlined in Section 4.1, is transmitted to the macro module for refinement. The macro module employs information regarding the user's intent and the current state of the environment to generate actions that augment the planned actions. A comprehensive description of this process can be found in Section 4.

### 3.2 Object Goal Navigation — *Find* Module

Successfully completing the game task depends significantly on accurately localizing salient objects. For example, in a cereal-making game, the user needs to locate the bowl, cereal, and milk. To make swift progress, users often find it more convenient to automatically locate these objects, rather than searching for them using primitive actions. With this motivation, the *Find* module assists the user in localizing these objects.

To provide a seamless user experience, the *Find* module employs a user-centric approach for object localization. Initially, the *Find* module checks if the object is present in the current scene. If the object is not present in the current scene, the *Find* module attempts to locate it in its memory and directly

navigates to it. The agent's memory is enriched as the users progress through the game and the agent acquires an implicit memory of the different objects it encounters, along with their respective attributes such as color. The memory stores the objects in the form of a lookup table with keys as canonical object names and their attributes. It stores the robot's position and rotation co-ordinates in which the said object was visible to utilize *goToPoint*. To be specific, the bot extracts *rotation* and *position* from *metaData*, so when the bot needs to go to a seen object, it uses those information and detects the seen object. Together with the atomic action to go to that object when present in the scene, our robot navigates to such objects. If the object is not in the current scene or in its memory, the robot begins active exploration of various viewpoints in the room to search for the object. The robot prioritizes active interaction with users by continuously prompting them to verify if the highlighted object conforms to their specifications.

Additionally, our team closely monitored feedback logs from users and discovered that some users desire the ability to pause *Find* module mid-operation, particularly as search times grew longer. In response, we introduce a user-friendly stop barrier triggered when running *Find* module, but another command comes in. Our robot halts *Find* module as quickly as possible, and it executes the new command.

### 3.3 Disambiguating Multiple Objects

Although the methodologies presented in Section 3.1 and 3.2 enable users to identify desired objects in numerous instances, ambiguity persists when multiple objects are present within a scene. To address this concern related to object multiplicity, Our *Color* module is utilized to differentiate the same class objects based on the color information. However, certain cases may render color-based distinction infeasible. *i.e.*, the multiple target objects have the same color. In such scenarios, the robot sequentially highlights each potential target object, prompting the user to verify the intended object. Upon confirmation, the system proceeds to the selected object; otherwise, it continues the verification process with the next object.

### 3.4 Handling Primitive Actions

To ensure effective handling of primitive actions, we place great emphasis on a crucial aspect that is often overlooked in existing embodied AI benchmarks - fatal failures. Currently, embodied instruction following models do not impose specific penalties for fatal and irreversible action predictions. However, as SalsaBot interacts with real users in the simulated physical world, such fatal error predictions may have severe consequences for the user experience. Our design approach, as mentioned in the introduction, is centered around this consideration, with macros and action suggestions being incorporated into the dialogue module. Similarly, for action planning, we have implemented various rule-based safety checks on top of our neural-symbolic model. This ensures that the model does not deviate from the user's intent, avoiding erroneous actions that may cause irreversible state changes or lead to significant user frustration if they need to undo the action. Specifically, we pass the instructions through a collection of hand crafted rules to filter out simpler instructions that can be reliably be parsed into an (action, object) sequence. For more complex instructions, we use a tuned version of Pashevich et al. (2021) to predict an (action, object) sequence.

## 4 Macro Module

This section will primarily focus on the design and implementation of *Macro* module, a pivotal module in SalsaBot aimed at enhancing robustness, generalizability, and user experience. This module plays a critical role in bolstering the robustness of our action planning by leveraging pre-defined action sequences to guarantee that the requisite conditions for action execution are fulfilled. Moreover, it offers considerable potential for future development, thereby affording a high degree of extendibility.

As illustrated in Figure 3, the planning model employs an ASR system and metadata processing to generate a sequence of actions and user intents that are subsequently passed on to the macro module. This complementary component serves to verify whether the current environment and user intents trigger macro generation. In the context of our example, the action of *Repair* is considered a valid macro action. Subsequently, the condition checker examines whether *Holding* condition is satisfied. If the condition is not sufficiently met, *Macro* module generates a series of actions to fulfill

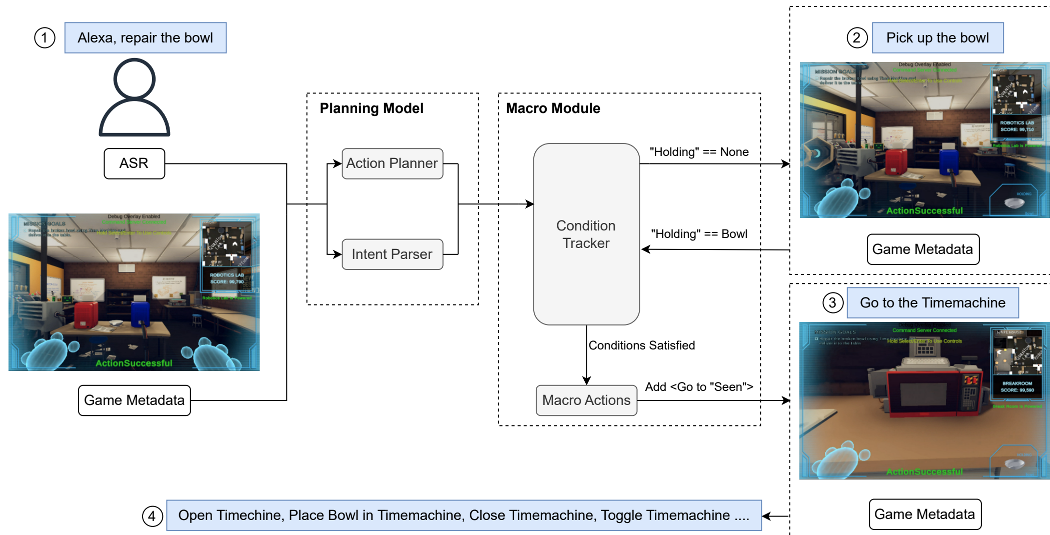


Figure 3: An example of macros module workflow

the condition. Thereafter, the new metadata is submitted to the condition checker for reassessment. Once the *Holding* condition is met, our *Macro* module generates a pre-defined series of actions to replace the previously planned actions. In the event that a user prompts the presence of a *Seen* object, a Goto action is appended to the action sequence. Finally, with the aid of the condition checker, a comprehensive action query is sent to the arena to enable the user to complete all necessary steps required to execute the *Repair* action.

As previously elucidated, our *Macro* module functions as a multifaceted component, serving as a condition checker and an action executor for the planning model. It is a complementary component to the planning model for complex action generation. Before processing user utterances and planned actions, the planning module parses user intent and generates an action pair. Subsequently, *Macro* module checks the environmental conditions and prompts users for necessary steps to accomplish specific actions. Alternatively, it outputs actions to automatically fulfill the requisite conditions. For action completion, *Macro* module leverages pre-defined macro actions and recompose the planned action to achieve more complex actions. Section 4.1 delves into the structure of pre-defined macros, whereas Section 4.2 elucidates the logic of utterance pre-processing. Furthermore, Section 4.3 demonstrates the macro's capability to adapt to the situational context. Lastly, Section 4.4 outlines the workflow of macro-based action planning and the logic of action completion, which was mentioned earlier.

### 4.1 Pre-defined Macro

The macros are structured in JSON format, as depicted below:

**PredictedObjIndex**: This field specifies the object index in the action pair, which is generated by the planning model. Its value indicates the index position in the pair, and in some cases, it can also represent the object's name. For instance, the action *"Repair the bowl"* would have the *Seen* field set to *Timemachine* since this object is the only one capable of executing the *Repair* action.

**Conditions**: *Conditions* outlines the necessary conditions for executing a given macro. If the *Holding* condition is present, the module will

```
{
    "MacroAction": {
        "PredictedObjIndex": {
            "Holding": "Index or Object Name",
            "Seen"   : "Index or Object Name"
        },
        "Conditions": {
            "Holding": "Index",
            "Seen"   : "Index"
        },
        "Actions": ["Action1", "Action2", "Action3", "..."],
        "ExtraActions": {
            "ObjectName1": ["Action4", "Action5", "Action6", "..."],
            "ObjectName2": ["Action7", "Action8", "Action9", "..."]
        }
    }
}
```

Figure 4: JSON template for macro definitions

expect the user to mention the object they are holding in their utterance. Subsequently, *Macro* module will generate a series of actions to enable the user to pick up the required object before proceeding with other actions. On the other hand, if the condition *Seen* is present, *Macro* module will prompt *Find* module to locate the object in the user's view before executing any other action. It's important to note that *Conditions* can consist of single attributes, which enables the macro to handle single (or no) object commands.

**Actions**: The macro's successful execution is contingent upon fulfilling all the specified conditions. Once all the conditions are met, *Macro* module will generate the appropriate action(s).

**ExtraActions**: *ExtraActions* field specifies special cases when an object name is present in the action pair. In such instances, the *Actions* field will be replaced by the action list defined within the *ExtraActions* field.

### 4.2 User Utterance Pre-prossessing

As an additional component to the planning model, *Macro* module will leverage all the outputs generated by the planning module, which include both planned actions and utterance-parsed results. In addition to generating planned actions, the planning model parses user utterances into pairs (Action, Object1, Object2) to facilitate future macro matching.

**Action**: The first key component of parsed intent is the action. This approach is adopted to overcome the limitation of training data, where certain actions may not be present. For instance, the *Repair* action is a macro action that the planning model does not process. In such cases, the macros module is designed to identify and match the macro-defined action and, subsequently, generate macro actions based on the match. Other macros that follow a similar pattern include *Take*, *Bring*, *Heat*, *Deliver*, and others. This match-making process relies on the pair generated by the planning model, enabling the macros module to identify and capture the occurrence of user intents.

**Object**: Objects are also crucial in the pre-processing process. Users usually specify two or more objects in their utterances in most complex actions. However, it can be challenging to parse the meaning and function of the non-receptacle object given the complexity of the language structure and context. For instance, in the utterance, *"Break the bowl with a hammer"*, the object *bowl* is the action receptacle, while the object *hammer* determines an action condition. To execute this command successfully, Simbot will first need to *"Pick up the Hammer"* and have that object toggled. In contrast, in another case, such as *"Pick up the cake in the fridge"*, with the *fridge* being the action receptacle, the *cake* is an implicit action receptacle as well. To break down this command, Simbot will first need to open the fridge and then pick up the cake from it.

**Coreference Resolution**: To improve user experience, we implemented a coreference resolution model that allows users to instruct SalsaBot in a more natural way. We first use a list of keywords that frequently appear in observed conversations, such as *"it"* to detect if an utterance refers to previous instructions. Then, we use a fine-tuned SpanBERT-Large (Joshi et al., 2020) from AllenNLP[2] to identify possible coreferences in the detected utterance. Finally, we use a rule-based filter to exclude irrelevant results and replace each detected keyword with the span it refers to. We also store the resolved utterance instead of the original user utterance in the instruction history to make the context more informative for future model predictions.

The planning model plays a crucial role in both generating planning actions and parsing user intent into (Action, Object1, Object2) pairs. This is important because the same object mentioned in different contexts or sentence structures may have different meanings within the game environment. Therefore, the planning model uses specific language rules to accurately parse the user's intent and create pairs that the macros module can match against. This ensures that the macros module can effectively identify user intents and take the appropriate actions in response.

**ASR Error Correction**: The efficacy of our framework is contingent upon the accuracy of Automatic Speech Recognition (ASR); should ASR contain errors, the consequences may be severe. For instance, a user may have intended to convey the commands (*"Find a freezer in the breakroom"*, *"Pour the*

[2] https://github.com/allenai/allennlp-models/blob/main/training_config/coref/coref_spanbert_large.jsonnet

*cereal into the bowl"*); however, ASR inaccuracies resulted in (*"Find a freezer in the rec room"*, *"Poor the cereal into the bowl"*). Consequently, instead of locating the freezer in the breakroom as intended, the search is erroneously conducted in an unanticipated location (Default room is the current room). To mitigate such issues, we conducted a comprehensive analysis of user interaction logs and rectified ASR errors for specific words (e.g., *pour, bowl*) utilizing a rule-based methodology.

### 4.3 Situation-aware Macro

In addition to *Holding* and *Seen*, there can be other key values specified in the *Conditions* field depending on the specific macro. When a macro is triggered, *Macro* module will either generate a series of actions to fulfill the conditions specified or prompt the user for necessary steps to be taken before executing the macro-defined action. For instance, if the *Holding* key value is present in the *Conditions* field, the module will generate a sequence of actions to Pickup the required object before executing other actions. On the other hand, if *Seen* is present, *Macro* module will trigger the *Find* module to locate the specified object in the environment before proceeding with further actions. It's worth noting that a macro can have single or no attributes, making it flexible enough to handle simple commands as well.

**Holding**: In order to satisfy the conditions specified in the *Holding* attribute, the macro module has been designed to examine whether the user is holding any relevant objects before executing the macro actions. If the user is not already holding the required object, the module will trigger the *Find* module to locate and pick up the object. However, in the event that the object held by the user does not match the predicted holding object, the module will inform the user and prompt them to go to the required holding object. These mechanisms ensure that the *Holding* condition is satisfied before executing the macro actions.

**Seen**: *Seen* attribute addresses the receptacle of the user's command. After identifying the receptacle based on the planning model pair, the macro module will insert a Goto Seen action into the action list to ensure that the object is in the current view before executing macro actions. This helps to prevent unsuccessful action errors. If the object is not in view, the module will trigger *Find* module to locate the object before executing the user command.

**Conditions Value Field**: This field consists of key-value pairs and serves to identify actions that can be omitted when executing generated macro actions. When the value is not 0, it indicates that when the condition is satisfied, *Macro* module should execute actions starting from that value instead of from the beginning. This design helps to reduce action generation redundancy if some actions have already been executed by the user.

In a special case, for example, when the user prompts *"Pick up the cake from the fridge"*, where the objects in the user command do not match the definition of *Holding* and *Seen*, *Macro* module will still follow the same template as before. The first object, *cake* in this case, will be treated as the action receptacle while the second object, *fridge*, will be treated as the second receptacle of the following action. However, to avoid confusion, *Macro* module can be modified to include an additional field in the macro template called *Secondary Receptacle* to explicitly address those kind of special cases.

### 4.4 Macro-based Action Planning

*Macro* module is an integral part of our pre-processing system, which complements the action planning process by providing a more complex action planning option. *Macro* module detects macro-defined actions and examines the current conditions to generate a more sophisticated planned action. This behavior not only improves the recognizability of actions but also enables the handling of unpredictable actions.

*Macro* module is highly extendable and can accommodate any input given in the form of a pair (Action, Object1, Object2). As described in Section 4, once the user defines a macro action and specifies its object's positional information and condition, the module automatically satisfies these conditions and generates actions accordingly.

In summary, *Macro* module plays a crucial role in enabling the efficient execution of complex actions by providing a flexible and dynamic approach to action planning. The implementation of *Macro* module represents a significant contribution to the overall effectiveness of our language-based game control system.

## 5 Response Generation

Section 5.1 elucidates the process of generating a template-based response, Section 5.2 summarizes the construction of the multi-turn dialog systems which handle the interactions in other modules, and Section 5.3 showcases the examples in details to prove the efficacy of our design.

### 5.1 Template-Based Generation

SalsaBot generates responses by selecting the pre-written templates and filling the slots in the templates. These templates explain the problems our bot meets, provide hints to play the game, and list the actions users can make. To achieve those purposes, we create different types of templates under the handcrafted conditions. There are 24 types of templates based on error signals ( *i.e.*, UnsupportedAction, AlreadyHoldingObject) received from Arena. One template introduces sticky notes to users and we append the template only to the first response for every user. Other types of templates are based on the raw utterances or the outputs from the planning module. The response module also considers the previous status and what the bot is holding to choose an appropriate template type. For each type of template, we select the response sentences randomly. After the selection, we fill the slots in templates with values of action types, target objects, available actions, or original utterances.

To enhance user-friendliness, we study user feedback and adjust the responses by shortening the sentences and decreasing the frequency of verbose ones. We balance the informativeness and conciseness carefully.

### 5.2 Multi-turn Dialog System

We use a multi-turn dialog system to fulfill the requirements of confirmation and clarification in other modules (*e.g.*, *Macro-based Planning*, *Find*, and *Disambiguation*).

**Macro-based Planning.** The macro module is designed to facilitate multi-turn dialogues with the user to assist them in fulfilling macro conditions. This is achieved through the use of prompts that are triggered when the user's holding object does not match their intended object. These prompts guide the user to find and replace the object to satisfy their intended action. Details on this functionality can be found in Section 4.3.

**Find Module.** Every time finding the target object, our bot highlights the object and consults users if the agent should go to the object. To avoid long waiting time during exploration, SalsaBot will inquire users if they would like to continue the searching after going through several viewpoints. After checking all possible viewpoints in the current room but still failing to find the target, SalsaBot will notify users the failure and advise users to search the object in other rooms. Besides the dialog part, Section 3.2 introduces the mechanism of *Find* module.

**Disambiguation.** When there are multiple similar objects in the view, our bot initially informs users the number of similar objects it has seen. Then, SalsaBot highlights each object and requests confirmation from users in turn until users affirm the highlighted object. If none of the seen objects are the intended one, our robot will tell users there are no more other objects it has recognized and suggest users to try other commands. Other details related to Disambiguation module are in Section 3.3.

### 5.3 Case Studies

In this section, we represent and highlight several examples of how our design greatly improves the user experience.

#### 5.3.1 Possible Actions

The game-like simulator causes two difficulties for users to play the games: 1) Users are unfamiliar with manipulating futuristic devices. For instance, users do not know how to operate a time machine or a laser. 2) It is hard to tell the which background or decorative objects are available to interact

with. For two similar desks, one could be a receptacle to place objects on and the other one could be just for a decoration.

In order to reveal the properties of objects in the simulator, simbot will identify any object mentioned by the user without valid actions and present a list of the most probable executable actions, provided the object is interactive. This is based on the Arena User Manual and error signals.

**Algorithm 1** Provide Possible Action Lists and Construct Templates

$O \leftarrow$ Object name ▷ Object name mentioned by users
$A \leftarrow$ Dict$[O]$ ▷ Retrieve the list of executable actions from a dictionary
**if** Receive UnsupportedAction **then**
  $A$.remove(prev_action) ▷ Delete the unsupported action based on the Arena error signal.
  **if** $A$ is empty **then**
    $R \leftarrow$ template(UnsupportedActionForNow, O) ▷ Tell users the object is not executable with for now
  **else**
    $R \leftarrow$ template(UnsupportedAction, A, O, prev_action) ▷ Notify the error and provide other actions
  **end if**
**else if** Receive a new plan without actions **then**
  **if** $A$ is empty **then**
    $R \leftarrow$ template(DecorObject, O) ▷ Tell users the object is a decoration
  **else**
    $R \leftarrow$ template(ActIsNull, A, O) ▷ Provide the possible actions
  **end if**
**end if**

### 5.3.2 Unspecified Receptacles

It is possible that for Place command, the utterance received does not include the receptacle where the holding object should be put. It usually results from natural dialogue from the user omitting the receptacle. However, the receptacle is necessary to construct a valid Place command.

Figure 5: Both table and 3D printer are valid receptacles for a cartridge.

One solution could be assuming the receptacle is in the current view. However, when multiple receptacles are in the views, and our bot predicts a wrong receptacle, it is frustrating for the user to correct our assumption (*e.g.* if we choose to ask the user to confirm the assumption) or undo the action of place (*e.g.* if we decide to execute the assumption directly). Instead of guessing the intended receptacle, we help the users realize the necessity of mentioning the receptacles so that they can play future games efficiently.

As the example shown in Figure 5, an user picks up a cartridge and tries to put down the cartridge into the 3D printer. There are two possible receptacles, a 3D printer, and a table, in front of the user. When the user asks to put down the cartridge, we check what objects the agent holds. If the holding object is a cartridge, we query the intended receptacle and give an example to teach the user to give a

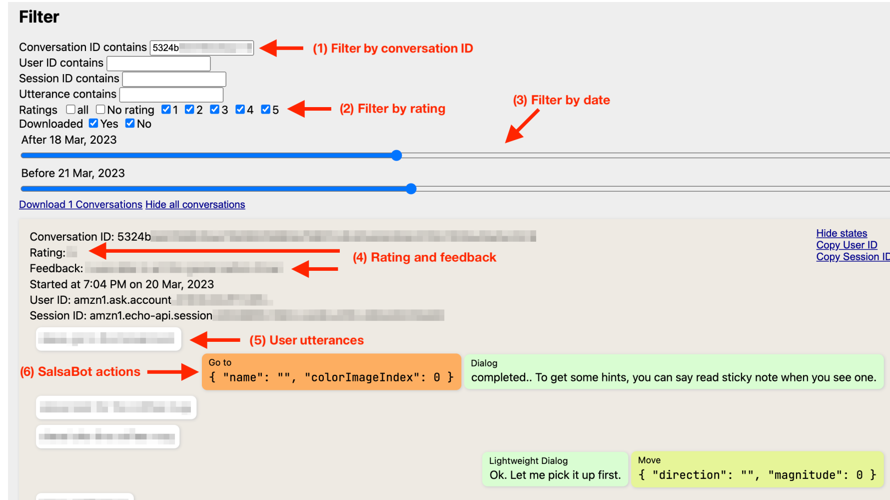


Figure 6: SalsaMonitor, a custom web-based tool for monitoring user interactions with SalsaBot. (1) Team members can filter by session and conversation ID. (2) Team members can filter by rating to find poorly-rated interactions. (3) Team members can filter by date to find interactions that correspond to a specific version of SalsaBot. (4) Each conversation's rating and feedback (if any). (5) User utterances, as seen after ASR. (6) The actions SalsaBot takes in response to an utterance. Note that IDs, feedback, and user utterances are blurred in this figure to protect user privacy.

more useful command. Otherwise, SalsaBot guides the user to look for the cartridge first. To do so, our bot needs to record the status of what it is holding. If we send a Pickup action with an object and receive a success signal of execution, we store the object as holding in hands. Next, after we send a Place action and receive a success signal, we mark the agent holding nothing.

#### 5.3.3 Error Signal of TargetOutOfRange

To interact with the objects, the agent is supposed to face the object in a range. If a user tries to interact with an object out of the range, we will receive an error signal, TargetOutOfRange. Instead of only sending the dialog to notify users of the problem, we directly insert a Goto action. The response to Arena becomes a list: [(Goto, <Object>), (LightweightDialog, <template(TargetOutOfRange)>), (<Action>, <Obj>)]

## 6 Learning from User Feedback

User feedback played a critical role in the development of our bot. We are highly driven to improve user experience by focusing on understanding how users interact with our bot and what features they desire. Our user feedback team developed a custom monitoring tool with a web UI (see Figure 6) where team members can search by session and user rating and easily view complete user interactions. We also integrate the monitor with our team's *Microsoft Teams* channel and receive notifications in the channel whenever a low rating ($\leq 3$) is received, so that we can take timely actions to investigate and fix potential issues. We also share daily summaries with the entire team so development focuses on the most critical issues. The summaries enabled us to find trends in user interaction patterns and identify our bot's weak points. Specific examples include the need for improved image processing, a *Find* module to look for objects, and a *Go* module.

**Improved image processing**: Our early analysis of user gameplays showed that our bot could not distinguish between objects of different colors. Our team quickly added a component to the image-processing pipeline, including a color detection module. This improved user play by eliminating any issues with detecting different colored objects.

***Find* module**: Another example of user-interaction-driven features is our *Find*module. By monitoring user interactions and feedback, we noticed that users repeatedly asked our bot to find objects. This

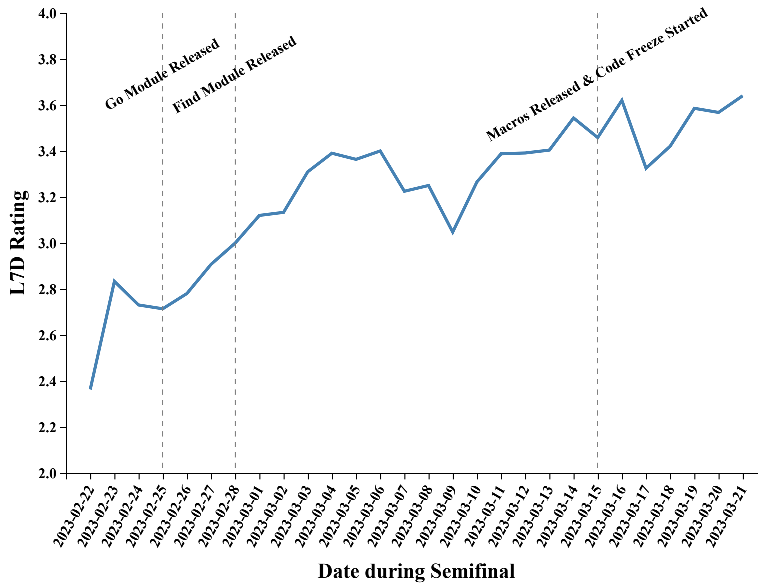


Figure 7: The L7D ratings since semifinal. Our ratings have been steadily increasing even on the new games after the code freeze. This is evidently driven by the major features we introduced, showing the effectiveness of the features.

realization led to quickly developing the *Find* module. Moreover, while we were developing our module, we wanted to ensure that we kept the user experience in mind. There we also added a dialogue response to let users know how they can interact with objects and to ask them to navigate our bot to an object if it is not in view.

***Go* module**: Our *Go* module is another example of a feature driven by user interactions. Our analysis of user interactions allow led us to discover another desired feature. We noticed users trying to retrace their steps to an object they had previously interacted with. This new module allowed our users to return to objects without the need for manual navigation or *Find* module.

## 7 Results and Analysis

**User Ratings:** We calculated the average of ratings in the past seven days to evaluate the efficacy of new features. From Figure 7, we notice the obvious increase of scores after the deployment of *Go* module, *Find* module, and *Macro* module. The upward trend validates our approach to this task and we hope to continue the trend to the end of the competition.

**Results on the Arena Dataset:** We report the success rate of our neural model based on the vision model Gao et al. (2023) and multimodal transformer Pashevich et al. (2021) as 18.97 on the validation split of the Arena dataset. However we want to mention that for SalsaBot, we perform significant modification and substitution to the model as described in this paper that do not translate to a higher success rate for the Arena dataset.

We also want to report the result from additional language models that we have trained, even though we do not use it in our system. Finetuning Flan-T5-base (Chung et al., 2022) with the training dataset in Gao et al. (2023), we achieved 78% planning accuracy on the all tasks in the validation dataset. We report planning accuracy as percentage of (action, object) pair that the model successfully generated compared to the ground truth. We consider the model to have correctly parsed an instruction when its prediction exactly matches all the sub-plans assigned to the instruction.

However, we also find that there are no primitive actions annotated in the dataset, such as move forward and rotate right. To enable the ability of parsing the instructions including navigation commands,

we augmented the dataset by asking GPT-3 (Brown et al., 2020) rephrasing the simple navigation commands we wrote. Moreover, we re-annotated Gao et al. (2023) by the following steps: (1) We masked Goto actions in the labels and asked GPT-3 (Brown et al., 2020) to fill the blank with navigation actions, including Move, Rotate, Look and Goto. (2) We only replace the actions of Goto and keep other parts of the annotation as original. (3) If the replacement is still Goto action, we revoke the replacement (to use the original object name). We plan to replace our current neural-symbolic model with T5 further augmented with trajectory data generated from large language models.

# 8 Discussion and Future Work

After introduction of each components of our design, SalsaBot has over 50% relative improvement in L7D ratings from 2.36 (on February 22, 2023) to 3.64 (on March 21, 2023). This validates that our direction leads to a better user satisfaction. However, we envision SalsaBot to be much more than its current form.

We plan to extend our macros to handle all non-primitive actions, enabling a layer of abstraction for users so that primitive actions are only used in occasional circumstances. We envision macros as a building block to handle much more complex tasks. To achieve this, we want to make them data-driven and learning-based so that macros can learn to perform more complex tasks and adapt to different environments.

Furthermore, we want SalsaBot to have more human-like capabilities, including the ability to retain a detailed memory of the environment and actions it has seen/done so far. Being environmentally aware can also be used to suggest macros and preemptively execute an action that the user has intended. With this capability, SalsaBot can drastically improve its performance in partially observable environments and meet the user's expectations that the bot will have capabilities that is similar to or exceed those of a human player.

Finally, we plan to augment our planning models with more data synthesized from large language models (LLMs) such as GPT-3 (Brown et al., 2020) or ChatGPT. This will enable us to address a wider range of utterances from real users. We believe using LLMs will enable a cost-efficient way of annotating a large amount of data. Additionally, due to the difficulty of gathering user traffic to evaluate our overall system, we are testing LLMs' capability to be used as an artificial human player that can be used to test the bot.

# Acknowledgments

We thank Amazon for financial and technical support and colleagues in the OSU NLP group for their valuable comments.

## References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc. 13

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*. 12

Qiaozi Gao, Govind Thattai, Xiaofeng Gao, Suhaila Shakiah, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zheng, et al. 2023. Alexa arena: A user-centric interactive platform for embodied ai. *arXiv preprint arXiv:2303.01586*. 2, 12, 13

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. 2020. SpanBERT: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77. 7

Alexander Pashevich, Cordelia Schmid, and Chen Sun. 2021. Episodic Transformer for Vision-and-Language Navigation. In *ICCV*. 5, 12

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1