SimBot - Navigation and Interaction

Adhokshaja Madhwaraj
amadhwar@andrew.cmu.edu

Jessica Zhong
zhengzho@andrew.cmu.edu

Kushagra Mahajan
kmahajan@andrew.cmu.edu

Malaika Vijay
mvijay@andrew.cmu.edu

Sai Vishwas Padigi
spadigi@andrew.cmu.edu

Vineeth Reddy Vatti
vbv@andrew.cmu.edu

Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Embodied task completion agents are a class of intelligent agents that can perceive, navigate, and manipulate objects in an environment in which they are situated. Previous work on such agents has focused on building non-conversational agents that must complete a task given a single instruction. In this scenario, the agent cannot ask for help if it needs more information to complete a task. The lack of interaction can make it frustrating for users to complete tasks when the bot cannot provide context into what it doing, where it is stuck, or if the bot cannot prompt the user when there isn't enough information for the user to know how to interact with the environment. In this work, we develop a bot for the SimBot challenge in conversational embodied task completion, where users can guide a bot to complete tasks in a 3D environment by conversing with the user and understanding the environment.

1 Introduction

Embodied AI enables virtual agents to see, listen, interact, and learn from a virtual environment. The goal of SimBot is to build a conversational embodied agent to complete tasks given natural language instructions. Such agents find use as household assistive bots or in search and rescue operations where humans might not be able to navigate volatile and dangerous terrain. Previous datasets for embodied task completion, such as ALFRED [1], are designed such that the agent receives a single instruction. There is no opportunity for the agent to engage in dialog with the commander and ask clarifying questions when it needs more information. The Amazon SimBot challenge focuses on helping advance the development of virtual assistants that will assist humans in completing real-world tasks cooperatively.

This report documents the bot we built and deployed for the SimBot challenge in conversational embodied task completion. In this challenge, a bot is expected to follow instructions provided by human users to interactively complete tasks in a 3D simulated environment, specifically the Amazon Arena 3D environment. The bot must be competent at language understanding, image understanding and conversational dialog. Users rate their experience interacting with the bot after each game session. This score is equivalently referred to as a Customer Satisfaction (CSAT) Score or a Customer Experience (CX) Score. Over the course of the competition, our primary goal has been to upskill our

bot such that task completion is greatly simplified, even for users who have no prior experience with the platform.

The Navigation and Interaction thrust of the SimBot project focuses primarily on scene understanding and action planning to perform actions that satisfy a user's instruction. Our contributions lie in an Image Segmentation model to identify objects of interest and an Action Planning module capable of generating a logical sequence of actions to execute on the Arena Simulation Engine. In addition to these primary components, we also focus on environment mapping, game session handling, and generating dialog prompts to understand bot failures and to help the user understand how to interact with objects where it is not immediately apparent.

2 SimBot Challenge Description

The goal of the SimBot challenge is to build a conversational embodied task completion agent that can interact with users to cooperatively complete tasks in 3D simulated environment. SimBot runs in a gameified application on an Alexa Echo device where users can interact with the bot through voice commands.

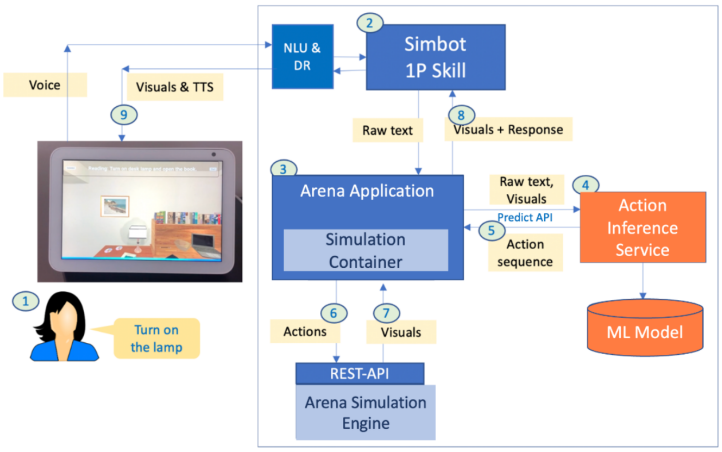


Figure 1: Arena Runtime Platform

Figure 1 describes the architecture of the Arena services that together create the game experience running on an Alexa device.

1. **Arena Skill**: The Arena Skill is responsible for receiving and parsing speech input into text. It is also responsible for invoking the Arena Application that processes the instruction.
2. **Arena Simulation Engine**: The Arena Simulation Engine is a Unity based simulation engine within which the robot navigates and completes tasks. It renders multiple indoor scene layouts and 3D assets with which the robot can interact. See the appendix for the full list of actions the robot can use to interact with these assets.
3. **Simulation Container Service**: This service is a wrapper around the Arena Simulation Engine to execute actions and stream visuals from the Arena Simulation Engine to upstream applications.
4. **Arena Application**: The Arena Application or the Arena Orchestrator orchestrates the flow for playing game sessions with an end user. It interacts with the Skill, the Action Inference Service and the Arena Simulation Container Service.
5. **Action Inference Service**: This component contains the logic for the embodied agent. Everything from dialog parsing to action planning is handled by the Action Inference Service.

The goal of our work is to build the Action Inference Service that is responsible for understanding and responding to user instructions. The remainder of this section describes how the Action Inference Service is setup.

Each time a user provides an instruction to the bot, it is translated to text by an Alexa Speech Recognition system residing in the Arena Skill. The Arena Application pulls the image of the what the bot is currently seeing from the Simulation Container Service and sends a request to the Action Inference service with the image and the text instruction. The bot then uses these inputs to chart out a sequence of actions to complete the task. The Inference Service then responds to the Arena Application with a set of instructions to execute on the Arena Simulation Engine. For each user utterance, the Arena Application will recursively invoke our Inference Service 10 times to complete the set of actions corresponding to the utterance. The Arena Simulation Engine responds with a fresh set of images and the execution status of each set of actions is executed. The turn corresponding to one utterance ends when we have exhausted the 10 requests or when the agent sends a dialog action back to the user indicating success, an error, a prompt, or a clarification question as shown in Figure 2.

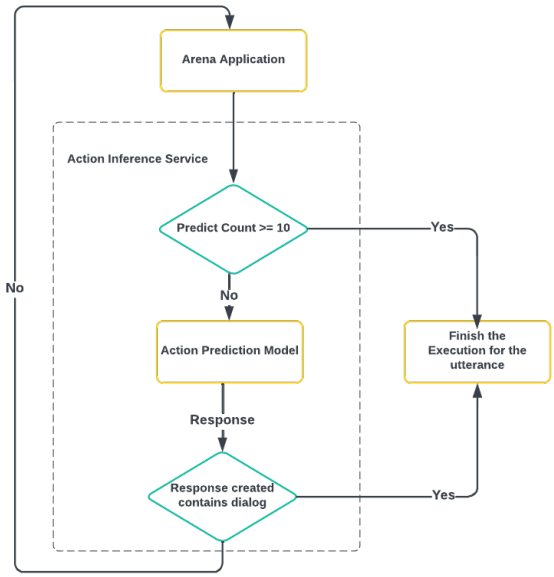


Figure 2: Action Inference Service's Execution

3 Hypothesis/Project Goal

The goal of our work is to develop a conversational embodied bot that is capable of completing a task in a 3D environment given human instructions. The primary hypothesis of our work is that conferring the embodied agent with the ability to converse with human commanders will lead to improved success rates on task completion as compared to the situations where the agent cannot generate dialog to ask questions and prompt the user with hints. The agent not only has the ability to understand user instructions and perform commonsense reasoning, but it also guides the user through intelligent dialog so that they are able to interact with the environment to successfully complete tasks. The bot can provide prompts, and clarifications to the user to help them complete the tasks by taking into account the scene and what was the last action that was completed.

We propose a modular system where each module performs a specific task and can be improved without affecting the rest of the system. The various modules in our system are: instance segmentation, object attribute identifier, action queue, path planning, session and state handler, mapping,

clarifications, error handler, and prompt generator. The functions of each of these modules has been discussed in detail in the subsequent sections. This modular structure allows us to model complex portions of the task-completion pipeline.

Some of the challenges encountered during the project are highlighted below:

- **Action batching**: The Predict API in the Arena Application can only be called 10 times for each user utterance as per the limit set by Amazon. In order to complete user instructions in these 10 calls, we needed to group actions together such that actions that did not require new state images were batched with other actions for efficient execution.
- **Clarifications**: These help the agent to better understand the user's instructions. So far, we have included 3 types of clarifications: 1) *Color Disambiguation*: To resolve ambiguities in several objects such as computers, shelves etc. which have instances of multiple colors within the same room. 2) *Instance disambiguation*: If there are multiple similar looking object instances in the same room, the agent would not know which instance the user wants to interact with. 3) *Location disambiguation*: If the agent is not able to find the object that the user wants to interact with, it asks the user for assistance in finding the object that he/she wants to interact with.
- **Handling multiple objects**: There are several instances where there are multiple instances of the same object class within the same room and the user is required to interact with specifically one of them to successfully complete the mission. This can be seen for Computer and Cartridge object classes.
- **CX score**: CX score refers to the customer experience score that is assigned by the users you play the missions with our bot. At present this score is being assigned by Amazon's internal beta testers. Our goal is not only to understand user instructions, but also to systematically guide and train the users to successfully complete the missions.
- **Thresholds for segmentation**: Determining the appropriate IoU thresholds for the different object classes was a challenging task, and required us to determine the suitable values empirically.

4 Relationship to Prior Work

Vision and Language-based navigation and task completion involve an agent performing a set of tasks based on language input with visual observations grounded in natural language. Earliest iterations of the capabilities of such autonomous agents were detailed in [2]. Recent work in this space is based on datasets that do not involve human-like conversation, rather a set of instructions based on which the agent must complete a task [1]. These instruction sets do not resemble conversations in a natural setting where the agent has the ability to resolve ambiguities through questions and clarifications. The earlier considered TEACh dataset [3] (which we do not use now, due to the nature of the competition) includes a conversation between two agents in a simulated environment. Our work is now based on a newer version of the dataset, which is based on the Arena simulator, which provides us with sessions with the users, along with visual images, and interactions.

Historically, navigation and task completion problems have been solved using Heuristic Action Selection, or more recently through Reinforcement Learning (RL) solutions. While end-to-end Deep RL solutions have performed well on navigation tasks [4], they are heavily reliant on structured state representation. Introducing the possibility of interjecting dialogue implies a variable state space. Recent research in the domain [5] inspires a modular approach to this problem, with proven improvements over other end-to-end solutions [6]. However, in the redesign of the problem statement by Amazon, navigation towards rooms and objects has been simplified as explained in appendix A.2.

5 Development Goals

The details of the milestone descriptions and due dates are mentioned in Table 1.

Milestone Description	Due Date
SimBot bootcamp	9/6
Arena Orchestrator Setup and Exploration	9/11
Template Data Generation	9/20
Dummy Data Creation	9/20
Static Mapping of Receptacle Objects	9/20
Instance Segmentation	9/28
Template Engineering	9/28
Arena Request Generation	9/28
Implement scottybot's arena model workflow	9/28
Basic Path Planning	9/28
DynamoDB Schema Design and API	9/28
Integrate request generation, path planning, and orchestrator	10/2
APIs to Communicate with Dialog Team	10/2
Integrate Navigation Modules	10/2
Integrate with Dialog Team	10/9
Package Integrated Model as Alexa Skill for the Skill Certification	10/16
Mid Term Check-in	10/26
Address Skill Certification Issues + Error Analysis	10/30
Iterate on Models and Design	11/13
Draft Report	11/20
Incorporate Amazon Feedback and Iterate on Pipeline	11/27
Amazon Internal Beta Testing	11/29
Final Report and Presentation	12/14

Table 1: Milestone Plan with descriptions and due dates

We have been able to successfully complete all milestones set till mid-November. Our bot is currently in the Amazon internal beta testing phase during which we will be receiving continuous feedback from testers. During the coming weeks, we will continue to iterate on our models and algorithms to further improve the user experience provided by our bot.

6 Project Requirements

6.1 Intended Users

SimBot as a concept has a wide range of intended users, which are not limited to just the domestic setting in which it is being developed and trained at the moment. In this section however, we present two example use cases in which a full-fledged SimBot, equipped with the right hardware could be deployed to assist humans.

1. **Household Assistive Bot**: The primary use-case of SimBot is to assist in general household tasks. It can also be leveraged by people with physical challenges who require help in day-to-day activities as well as in circumstances where they need external assistance.
2. **Search and Rescue**: SimBot can be used in search-and-rescue settings where the user can provide the instruction and the bot can navigate and perform critical tasks.

Currently, since we are working towards the Amazon Alexa SimBot Challenge, we are catering to a new simulation environment where agent models are deployed on Alexa devices across multiple users in a gamified setting. Trained models are hosted on Alexa devices in an interactive game where users provide language input and guide the bot toward task completion. Metrics for success are task completion from a pre-defined set of missions, as well as customer experience and satisfaction. Apart from being a metric to assess the progress of teams, it also serves as an important data collection and augmentation tool, which can be used for further fine-tuning and improvement. Teams also receive feedback from users which can be integrated through iterations of improvement of the agent.

6.2 System Functionality

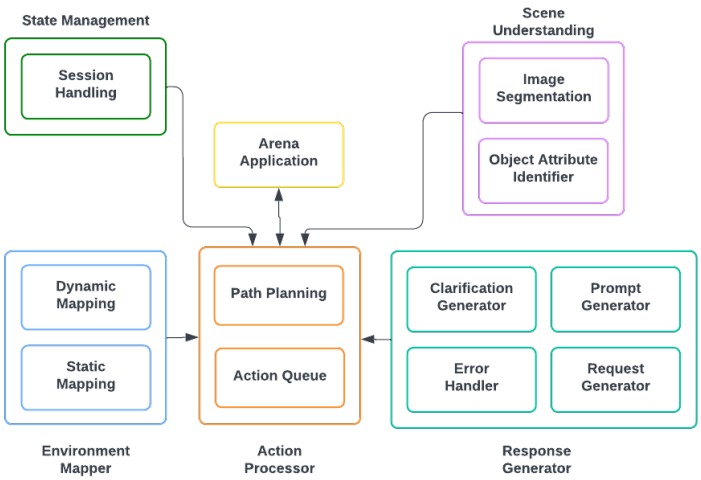


Figure 3: Arena Runtime Platform

Figure 3 illustrates the main components of the SimBot Navigation and Interaction pipeline. This section describes the functional requirements of each of these components.

6.2.1 Environment Mapper

The configuration of the layouts of the Arena environment change across each mission. The location of objects in these layouts is not static either. The mapping module of the Navigation and Interaction pipeline is responsible for maintaining information about where objects have been seen in the environment as the bot moves through it.

- **Static Mapping:** Some large objects such as the refrigerator or the laser are guaranteed to be located in the same room across layouts and games. The purpose of this module is to construct and maintain a mapping between such object classes and their corresponding rooms. We later use this information to decide which room to explore to find an object of interest.
- **Dynamic Mapping:** As the agent explores the environment, it encounters objects that might not be of immediate interest to the current subtask, but might come of use later in the mission. The purpose of the dynamic mapping module is to dynamically track the location of objects the bot has encountered as it moves through the environment.

6.2.2 Scene Understanding

Every typical interaction with the environment results in the bot receiving one 300×300 color image, while a `look around` results in receiving 4 color images. In order to navigate to and interact with objects of interest, SimBot must be able to identify and localize objects in these images, as well as be able to distinguish multiple instances of the same object type by attributes like color and spatial location for downstream decision making. The scene understanding module of SimBot is responsible for:

- **Image Segmentation:** The function of the Image Segmentation Module is to identify and localize objects in a frame of the environment. This involves mapping each pixel in an image to its corresponding object class. It must be capable of identifying multiple masks for each object class and associating each mask with a score and label.

- **Object Attribute Identifier:** Some instructions require the bot to find objects with certain properties, such as a red bowl or a blue monitor. The Object Attribute Identifier must be capable of determining whether an instance of an object identified by the Image Segmentation module satisfies the specified attribute constraints. It must be able to handle color, shape and location. This functionality is also useful when the bot must determine which, among a set of identified instances of an object, the bot must interact with.

6.2.3 Session Handling

Multiple users can play games on SimBot simultaneously. Each game is associated with its own dialog history, action history, and state and it is necessary to isolate the state of each game to ensure independent execution of actions across games. The session handler is responsible for maintaining dialog and action history as well as state by routinely updating and retrieving session metadata stored in DynamoDB.

6.2.4 Response Generator

The bot must communicate with the Arena Application as well as with the user to complete tasks. The components of the Response Generator are responsible for constructing either dialog responses or structured responses describing actions to be executed on the Arena Simulation Environment. A.2

- **Request Generator:** Each user utterance is translated into a set of actions that bot must execute on the Arena Simulation Environment. These actions are specified by a structured format supported by the Arena Simulation Environment. Given an action type and its arguments, for example (turn left, 90°), the request generator must construct the corresponding JSON specifying the action. The example below illustrates the format of a such a request.

```
request = {
            "id": str(uuid.uuid1()),
            "type": "Turn",
            "turn": {
                "direction": "left",
                "magnitude": 90
             }
          }
```

- **Error Handler:** There are several situations that can cause the bot to be unable to complete an action. For example, it may be too far away from an object to interact with it. The Arena Simulation Environment provides status codes for each action that it executes. The Error Handler is responsible for parsing these status codes and constructing dialog responses to let the user know what went wrong and what instructions they can give the bot to help rectify the problem. Refer to A.1 for a complete list of error types and responses the bot provides.
- **Prompt Generator:** Several objects in the Arena environment are objects that do not exist in the real world. Some examples of such objects are freeze rays, time machines and color changer machines. Users who are not familiar with these objects will not immediately know how to operate these devices. Since the user is also not primed with the full set of actions it can ask the bot to perform, it can be difficult to figure out how to interact with these objects. The prompt generator module is responsible for generating prompts on how to interact with objects the bot is near when it is not obvious. For example, if the user has navigated the bot to the laser machine, the bot should prompt the user with a dialog indicating that they must navigate to the red monitor and turn it on to fire the laser.
- **Clarification Generator:** The instructions provided by the user sometimes might not be sufficient to correctly identify the object that is required for the mission completion. For example, if the user asks the bot to turn on a monitor and multiple monitors are visible to the bot, the bot must be able to ask a clarifying question to determine which monitor it must

interact with. The role of the clarification module is to identify attributes (such as color, shape or location) of the objects that can be used to distinguish them, and then use these attributes to ask a clarifying question.

6.2.5 Action Processor

The Action Processor is the central coordinator of the Navigation and Interaction pipeline. It invokes the previously described modules to generate a sequence of actions in response to user utterances to compete sub-tasks of a game mission. The input to this module is a set of primitive (action, object) pairs generated by the Language Parser developed by the SimBot Dialog team. These actions may not be sufficient to complete a task. For example an instruction that reads "Pickup the bowl" may require the bot to first navigate to the bowl. The Action Processor is responsible for filling is missing actions, logically grouping actions that can be executed as a batch, and generating the requests that cause actions to be executed on the Arena Simulation Engine.

- **Action Queue:** The Navigation and Interaction pipeline must keep track of the sequence of actions to be executed to compete a sub-task. The primary responsibility of the Action Queue is to maintain this sequence of actions and serve up the next action to be executed on each turn. Actions can either be executed one at a time or as a batch of actions on the Arena Simulation Engine. The Action Queue is also responsible for performing this logical batching of instructions where appropriate. Section 7.1.5 describes how this grouping is accomplished.

- **Path Planning:** Objects that the bot must interact with to complete a subtask of the mission might not be visible in the current frame. In these situations, the bot must explore the room to localize the required object. The Path Planning module is responsible for charting out the sequence of navigation actions the bot must perform to explore the room and find the object. Each room in the layout of a game is associated with a set of viewpoints, or locations in the room the bot can directly navigate to with a goto command. The path planning module determines the sequence of viewpoints to visit and populates a queue of navigation actions the bot must perform to find the object.

6.3 Non Functional Requirements

Considering quality attributes while designing intelligent systems is of utmost importance, especially when the model will be deployed in the wild. As our team builds solutions that will directly interact and communicate with humans, we lay down these expectations for our system:

6.3.1 Availability

Throughout the SimBot competition, testers from Amazon interact with and rate our bot. These ratings are crucial to our success in the compeition. If our Inference Service is down during such testing, our bot will receive poor ratings. It is important for our service to be up during active testing hours (9AM - 8PM PST). Our service runs on an AWS EC2 instance and has controls for automatic restarts in the event of a failure.

6.3.2 Profanity

There are no regulations on the kinds of utterances that can be used to interact with SimBot. In the event that a user uses profane or inappropriate language, the bot must not entertain such requests. The SimBot dialog team has established profanity checks based on standards set by Amazon. When an utterance is identified as inappropriate, the bot simply responds with a dialog indicating it did not understand the instruction.

6.3.3 Interoperability

The SimBot project has several moving parts across both the Dialog and Navigation and Interaction components. Several dialog components communicate with components in the Navigation and Interaction space, as a result of which they must maintain high levels of interoperability. This is accomplished by setting up well defined interfaces based API requirements for the interacting modules.

6.3.4 Usability

Our bot is regularly tested by testers who are not familiar with the Arena Environment or the game itself. For first time users, it can be hard to accomplish tasks without knowing the full action space of the bot as well as the full set of permissible interactions with all object types in the environment. If not prompted, users will need to spend a great deal of time experimenting with the bot to see what it is capable of and how to interact with various objects. Since the score we receive on game completion is time-weighted, this could result in low scores. Usability is therefore a crucial requirement for our bot. Specifically, the bot should provide appropriate dialogs to the user that make it easy for them to understand how to play the game.

6.3.5 Scalability and Speed

Latency is an important factor that influences customer experience with completing missions on SimBot. If the bot takes too long to execute an action or respond to the user with a dialog, the user could get frustrated looking at a static screen while they wait. We must ensure that the bot responds to user instructions with minimal latency. This constrains the types and sizes of models we can use in our language and scene understanding pipelines.

Currently, a single EC2 instance handles all requests from each game running on the bot. While this is acceptable in the current scenario where testing is infrequent and often not concurrent, when the bot is open to interaction with the general public, our service must be able to handle a high volume of requests. While this is beyond the scope of the challenge, load-balancing requests across multiple instances should be considered.

6.4 Resource Requirements

There are 2 primary resource requirements for the SimBot challenge. Our data requirements are satisfied by the SimBot dataset provided by Amazon. This dataset consists of vision data, bot trajectory data and game description files. These datasets are described further in section 8.1. Our hardware requirements are satisfied by two AWS G4dn instances with NVIDIA T4 GPUs. One instance runs the Action Inference Service which responds to user requests. The second instance runs the ML Toolbox which is our development environment.

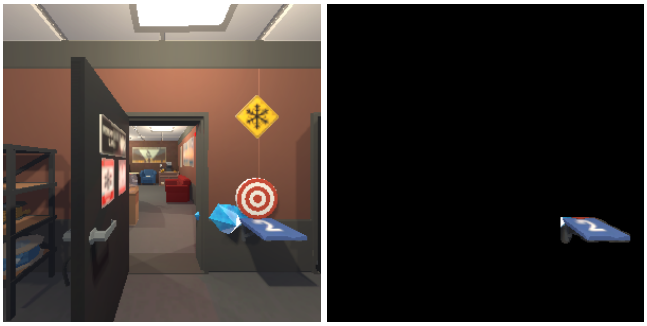
7 Experiment/System Design Overview

7.1 System Architecture

This section describes the architecture of the Arena Runtime Platform, which is the set of services that make up SimBot. It also describes the architecture of the Navigation and Interaction pipeline of the SimBot Action Inference Service.

7.1.1 Environment Mapper

We construct two kinds of maps, stored as dictionaries, of the environment - static and dynamic. For each object that the bot must interact with, we first query these states to determine whether the bot has already seen the required object. If it has been encountered, we simply navigate to the location saved in the mapping



(a) Example image obtained from Arena Simulator (b) The mask generated for a *Blue Shelf*

Figure 4: Image Segmentation Input and Output

- **Static Mapping:** Static maps construct mappings between large objects, whose locations are guaranteed to remain the same across layouts, and the rooms in which they appear. To construct these mappings, we parsed the set of game description files to fetch the locations of all objects with static locations 8.1.
- **Dynamic Mapping:** As the location of the smaller objects change between different layouts and games we need to create a dynamic mapping for these objects, Dynamic maps are constructed on the fly when the bot moves across the rooms. The Image Segmentation model is triggered at every location the bot is navigated to, and all the objects that the bot identifies are used to update the map. Additionally, image segmentation can be done at points that are not viewpoints. In these situations, we save the location of the object as the nearest viewpoint from the current bot location. This mapping is also saved as part of the session state in DynamoDB.

7.1.2 Scene Understanding

In order to navigate to and interact with objects or interests, SimBot must be able to identify and localize objects in the scene, as well as be able to distinguish multiple instances of the same object type by attributes like color and spatial location. Both the modules that comprise scene understanding are called when we need to identify the mask for an object with the option of selecting them based on the attributes.

- **Image Segmentation:** The image segmentation module takes as input a set of color images, attributes, and classes, and outputs a set of masks obtained across the images. When each image is received, it is passed through a MaskRCNN instance segmentation model. The model requires a configurable `probability_threshold` parameter, based on which a mask is selected or rejected. Outputs are a set of scores, labels, and predicted masks for each of the selected instances. If a color attribute is received from the dialogue team, we consider a threshold-based system to obtain the main color of the object present in the mask. If there is disambiguation or further information required to complete the task, it triggers the clarification module to process it further.
- **Object Attribute Identifier:** Currently the object attribute identifier is able to only obtain the color information of the objects from an image. The Object Attribute Identifier uses the masks that are provided by the Image Segmentation module to identify the objects of interest. The module uses the binary mask and superimposes it on the original image to get the object of interest and then finds the number of pixels that belong to the predefined ranges of red, green, and blue. The particular object is then either identified as one of the three primary colors or not assigned a color itself if it is below a certain threshold to avoid assigning colors to say a white computer.

7.1.3 Session Handler

Multiple users can play SimBot missions at the same time, necessitating session handling. We use DynamoDB to store and retrieve the state for independent sessions. Each game session is associated with a session identifier used to key state information in DynamoDB. At the beginning of each turn, DynamoDB is queried for the state associated with the current session identifier. If a key for the current session exists, we initialize the state for this run with the state retrieved from DynamoDB. Else, an empty session state is created. Each turn comprises 10 calls to the function that plans actions to execute on the simulator. Once the turn is completed, the state is written back to DynamoDB. This approach avoids having to read and write to DynamoDB 10 times each turn.

7.1.4 Response Generator

The bot must communicate with the Arena Application as well as with the user to complete tasks. The components of the Response Generator are responsible for constructing either dialog responses or structured responses describing actions to be executed on the Arena Simulation Environment. A

- **Request Generator:** The Actions that need to be executed are specified by a structured format supported by the Arena Simulation Environment. The request generator's task is to construct the corresponding JSON specifying the actions. The request generator gets a list of primitive instructions as an input 8.1 and it identifies the type of instruction as Goto, Navigation, or Interaction. The final response to the Arena Application is created by using the templates for each action type.
- **Error Handler:** The bot can have multiple reasons to not complete the task, either because the Arean Application is unable to complete the required action because of an error in the response sent to it, or because the path planning fails and the bot is not able to locate the object. The Error handling module handles these situations in different ways -
  - Errors from the Arena Application: The error handling module handles all the different errors with its own corresponding message to help the user understand the problem, and what information is required by the bot to get around the error. The different responses to the errors from the arena application are documented in A.1
  - Errors from path planning: When the bot is not able to find the object via both the Image Segmentation module and the Path planning module, then we provide messages like "Sorry, I couldn't finish executing my tasks. Can you give me a simpler task?" or "I couldn't locate the object. Are you sure it's in the current room". The user would then be able to identify the problem and then either give a simpler new task or navigate to a different room.
- **Prompt Generator:** The prompt generator module is responsible for generating prompts on how to interact with objects the bot is near when it is not obvious. The Action processor module creates a response map to store all the responses that were sent to the Arena Application. The Prompt generator module uses this response map to find the last action that was successfully completed by the bot. The prompt generator then creates appropriate prompts after the specific action is completed. For example, after we goto the freeze ray, we give a prompt "To use the freeze ray, you can tell me to turn on the blue computer". This prompt is then sent to the Request generator to create the dialog to send back to the Arena Application.
- **Clarification Generator:** When the user provides either a Goto or an Interaction action that requires a mask as input, the image segmentation module provides multiple masks as the output. The clarification module is used in the scenario where we cannot disambiguate which mask we need to select.

  Currently, the bot only disambiguates based on color. It gets the masks and identifies what are the set of primary colors that are seen in all the masks. If it encounters more than one, and it needs to distinguish between which one it needs to select, it creates a dialog response

Figure 5: Scenario for Clarification

by asking the user to select a color between the set it has identified. For example, if we identify that the masks for identifying a monitor have both red and blue primary colors, then the bot sends back the dialog response "Are you looking for a blue or red monitor?".

In the future, the clarification module would also handle other attributes like shape and location, where if we are not able to distinguish the masks by color, we will use these additional attributes to disambiguate.

7.1.5 Action Processor

Natural language instructions given to the bot by the commander are first parsed into a sequence of primitive actions along with their arguments such as the source and target object of an action, and attributes of the object such as the color the bot must interact with. These raw actions serve as input to the navigation and interaction modules of the pipeline. The Action Processor is the brain of the Navigation and Interaction pipeline. It coordinates with the rest of the modules to decide the next sequence of actions the bot must execute in response to a user utterance.

- **Action Queue:** The action queue is responsible for maintaining the sequence of these primitive actions the bot must perform to complete a task. This queue is architected as a standard queue with some exceptions described later in this section.

  The enqueue operation first translates an action into a format that the rest of the navigation and interaction modules are designed to expect. These translated actions are later pushed into the queue, internally stored as a list of dictionaries, each of which represents an action.

  The pop operation involves additional logic over a simple queue. Actions can either be executed one at a time on the Arena platform or can be executed as a batch of instructions. Actions that do not require intermediate responses from the simulator are grouped into a batch. Those that require inputs from the simulator before executing must be executed individually. For example, some actions such as (goto, object) require the next set of images from which the object mask is identified before it can be executed. Other actions such as (move, forward) do not require image input and can be grouped with other actions that similarly do not require image input. The pop operation constructs and yields this batch of actions.
- **Path Planning:** The path planning module is responsible for charting out the sequence of navigation actions the bot must perform to explore the room and find an object. It handles both receptacle and non-receptacle in different ways. For receptacle objects the room that the bot should search is retrieved from the environment mapping, as their location does not change across missions. For the smaller objects, we search the bot's current room. We populate a path planning queue with actions that navigate the bot to each of the 8 viewpoints

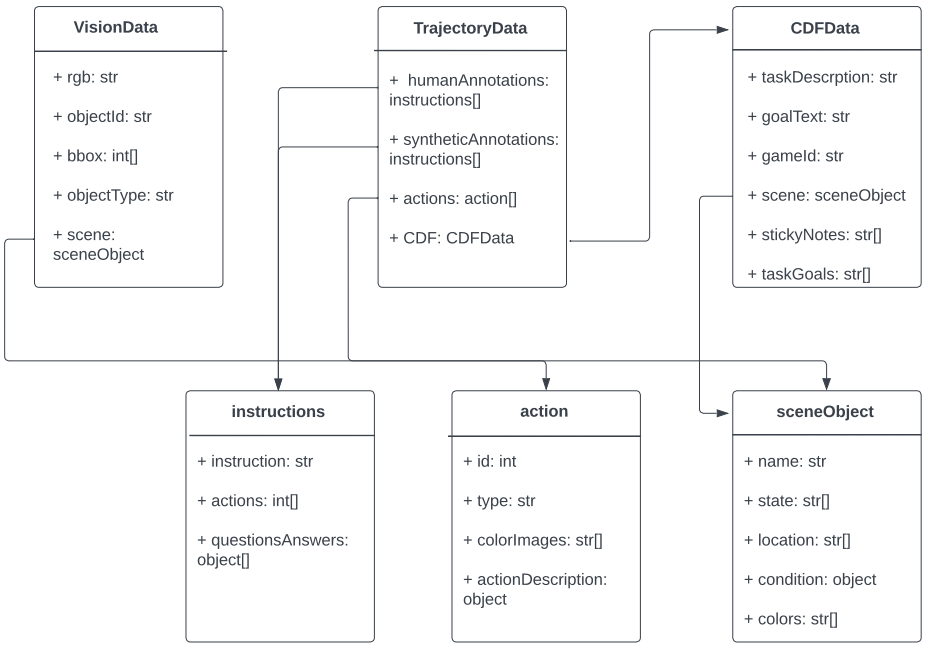


Figure 6: Class Diagram of Dataset

of the room. At each viewpoint, we also add an action to look around so that the bot gets a 360°view of the current location.

7.2 Data Design

The dataset provided by Amazon comprises of the vision data, action trajectory data (containing the ground truth action trajectories of the game missions), and CDF data (containing the game mission information such as initial state, goal state, and game text data). Images contain RGB ego-centric views of the environment from the follower's perspective. Figure 6 shows the relationship between the different components of the data.

The vision data contains annotations of the RGB image and the associated ground truth image. Each image has an associated object id and the bounding box annotations of the ground truth boxes for the various objects in the image. For each object, we have the associated object class and the scene information corresponding to the RGB image. The trajectory data consists of human-annotated instructions and template-generated synthetic instructions. Each instruction consists of the instruction text, the action indices corresponding to the instruction, and the set of question-and-answer pairs between the user and the bot when the bot needs clarifications. The trajectory data provides the list of actions that the bot takes to execute the instruction provided by the user. It also contains the CDF data.

Each action object includes the action id, the type of the action such as 'look', 'move' etc., the set of color images provided to determine the state for action execution, and the action-specific description. This would include things like magnitude and direction for navigation actions. This description would include other attributes depending on the action type.

The CDF data includes a description of the task based on the instruction provided by the user, in addition to the overall mission goal. The scene information is also stored for a mission CDF. There is additional information provided to the user through sticky notes, which explain to the user the individual steps involved in achieving the goal.

7.3 Implementation Overview

7.3.1 Navigation and Interaction Sequence Diagram

This section describes the implementation details of the modules described in the System Architecture. Figure 7 exhibits a sequence diagram illustrating the sequence of events involved in translating an instruction provided by the user to a set of actions executed on the Arena Runtime Platform.

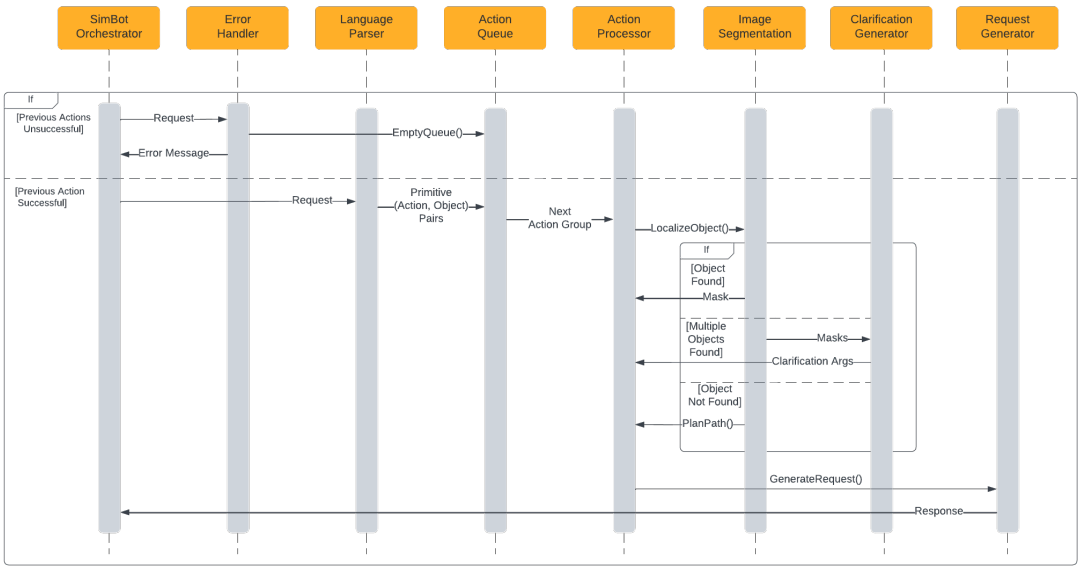


Figure 7: Sequence Diagram

Once a request reaches the Action Inference Service, it is first parsed into a set of (action, object) pairs by the Language Parser. The actions and objects both come from a closed set. These primitive actions are then added to the Action Queue. In each turn, a group of actions that can be executed as a batch is aggregated and dequeued from the Action Queue. These actions are then sent to an action processor that decides how to handle them based on their type. They can be one of Navigation or Interaction. If an action requires the bot to interact with an object, the Action Processor calls the Image Segmentation Module to localize the object in the scene. If multiple instances of the object are found in the current frame, the Clarification Generator is invoked to find a distinguishing factor amongst the masks identified and generate a question to determine which instance of the object the bot must interact with. Once the Action Processor has identified values for all the arguments required for the given action type to successfully execute on the Arena Simulation Engine, the Request Generator is invoked to construct the request describing the actions to be executed. This response is finally sent back to the Arena Application, which then forwards it to the Arena Simulation Engine where actions are executed.

After each batch of actions is executed on the Arena Simulation Engine, it responds with the execution status of either the first action that failed to execute or the status of the last action in the batch (if every action executes successfully). The Action Inference Service first checks the status of the last batch of actions and first responds to failed actions before processing the next batch of instructions in the Action Queue.

7.3.2 Navigation and Interaction Workflow

Figure 8 illustrates the detailed workflow of the Navigation and Interaction pipeline. When a request enters the Action Inference Service, SimBot first checks if it has exceeded the limit of 10 sets of actions it can execute per language utterance from the user. If we are out of turns the error handler is

Figure 8: Action Processor

invoked to let the user know that SimBot couldn't complete the action. Next, we parse the response corresponding to the previous set of actions executed on the Arena Simulation Engine to determine if there were action execution failures. In the event of action execution failures, the error handler is invoked to deliver the appropriate error message. Else, we begin the sequence of action prediction.

We begin by checking whether the bot is currently executing a sequence of navigation actions to find an object of interest i.e. we check if a path planning sequence is in progress. If yes, we perform image segmentation on the current set of images received from the Arena Simulation Engine. If the object has been found, we invoke the request generator to construct the request for which the object is required. If the bot sees multiple instances of the object and it is not clear which one it should interact with, the clarification generator is invoked to ask a question. If the bot could not localize the

object, the path planning sequence is continued if there are viewpoints left to explore. Else the error handler is invoked to let the user know the bot couldn't find the object.

If the bot is not in the middle of a path planning sequence, we pop the next group of actions that must be executed from the action queue and handle it based on the type of the action.

- *Goto*: Goto instructions can have a room name or an object name as its target. For the case of a room, we can directly invoke the request generator to construct the *Goto* action request. For objects, we must first decide the room to navigate to and then execute a path planning sequence to find the object
- *Navigation*: Navigation instructions do not require image input and can directly be passed on to the request generator to construct a navigation request
- *Interaction*: For interaction actions, the bot must again invoke the image segmentation sequence of actions described in the Goto workflow before executing the interaction action

7.3.3 Navigation and Interaction Data Flow Diagram

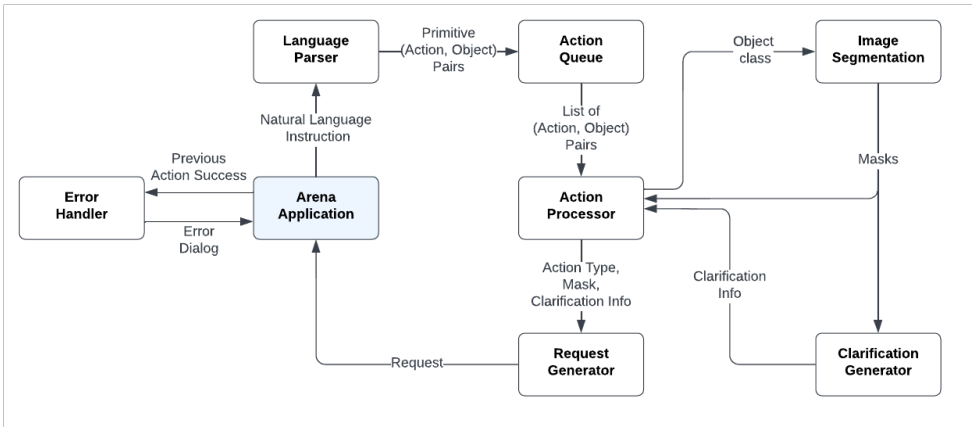


Figure 9: Data Flow Diagram

Figure 9 illustrates the flow of data between the various modules involved in . The Arena Application provides natural language instruction to the language parser, which then processes the instruction and converts it to a set of primitive instructions which contains pairs of (action, object). Action Queue receives these primitive instructions and groups them to allow batched execution and sends them to the Action Processor. Action Processor sends the object class and images as input to the Image Segmentation model which processes the images, and sends the masks that it finds for those images corresponding to the object class to the Clarification Generator and also to the Action Processor. The clarification generator uses the masks and sends back the clarification information to disambiguate the images to the Action Processor. The Action Processor creates a request by calling the request generator with the appropriate information like action type, mask, and clarification information. Finally, the Arena Application sends the previous action success message to the error handler, which does the parsing and sends back the appropriate dialog response.

8 Experimental Design

8.1 Dataset

The dataset provided by Amazon consists of 3 components:

- **Vision Data**
  It contains the images, ground truth segmentation masks, and the associated metadata. These

images are scene images collected from different runs of the bot and different game layouts used for that specific run. The color and segmentation images are the RGB egocentric views of the robot and the corresponding ground truth segmentation images respectively. The metadata contains a list of object annotations for the image such as the bounding boxes for the various objects in the image, paths to the RGB and the segmentation mask images, and the object class corresponding to each bounding box. The metadata also contains information about all the commands executed prior to the capture of that image. We have about 450k annotated segmentation images and a total of 62 object classes in the data, and each run has its data used for either training or validation. This data is used for training our instance segmentation models like MaskRCNN and MaskFormer.

- **Trajectory Data**
  For the action trajectories, the dataset contains ground-truth action trajectories for completing 3.5k+ game missions. The action trajectories are also paired with robot-view images. Each trajectory is annotated with 3 sets of language instructions. And on top of each set of human language instruction, there are 2 sets of questions and answers collected. Other than human language annotations, for each action trajectory, there is a set of template-based synthetic language instructions.
- **CDF Data**
  The dataset also provides CDF (Challenge Definition Format) files to store game mission information. It contains all the necessary information to run a game mission in Arena, including: 1) *Game initial state*: game scene, robot initial location, and state object initial location, and initial states (such as cabinet door open or close). 2) *Game goal state*: robot and object locations and states that are necessary to complete the game mission (Once all the goal states are met, the game mission is considered completed). 3) *Game-related text data*: game mission description text, etc.
  CDF data is used to obtain information about the possible locations of the large and small receptacles across missions. This allowed us to determine that the locations of the large receptacles are fixed across missions, and narrow down the list of possible rooms where each of the small receptacle objects could be located.

8.2 Machine Learning Models/Algorithms

8.2.1 Data Cleaning and Preparation

The vision dataset contains nearly 450k images and associated metadata, containing information about pixel-wise mapping to classes (semantic segmentation), bounding boxes for separate object instances, and pixel-color-to-class mapping, along with annotations and other image specifics. As mentioned earlier, many of these images contain instances of classes that are not *interactable*, or not relevant to our model. This necessitates a data-cleaning process.

We iterate through the trajectory dataset to collect all interactions of the bot with objects in the simulator, to obtain a list of classes that are *interactable*. Using this set of classes, we prepare and clean the dataset to ignore images that have no instances of objects belonging to these classes. We additionally consider a class-to-area threshold to only consider instances of classes that have a visible area greater than the selected threshold. This is done to prevent selecting and training our model on images that have very small, misrepresented renditions of objects.

We also generate additional metadata including train-validation-test splits and compiled lists of image and segmentation image files, class-to-index mappings, class-compression mappings (multiple sub-classes are combined to a single class, for eg: green button, red button, e7 button, b7 button to button and hammer print cartridge, action figure cartridge to cartridge).

8.2.2 Model Architecture and Training

In our experiments, we have experimented with two models, namely MaskRCNN [7] and MaskFormer [8].

- **MaskRCNN [7]** This model is the most widely used Convolution-based instance segmentation model, and this is also the model that is used by Amazon in their placeholder models. The MaskRCNN model contains a Resnet feature extractor backbone and uses Feature Pyramid Networks (FPN) [9] and Region Proposal Networks (RPN) [10] to generate segmentation masks. The MaskRCNN model is trained on the color images as the input, along with the bounding boxes of each instance recorded in the image, a semantic segmentation map, with each pixel colored based on the representative class. The output obtained is the predicted class, predicted mask, as well as a confidence score. The latest MaskRCNN model was finetuned and retrained on smaller objects of interest, such as the bowl in the microwave, or in shadows behind other objects. This model is capable of identifying all objects required to complete the missions as part of the Alexa SimBot challenge.
- **MaskFormer [8]** While MaskRCNN performed really well for the large and medium-sized objects, we observed a performance degradation for small objects such as apples, donuts etc. MaskFormer is one of the state-of-the-art models for instance segmentation, achieving superior mIoU on several datasets like Cityscapes [11], ADE20K [12], COCO-Stuff [13]. It uses a transformer decoder to compute a set of pairs, each consisting of a class prediction and a mask embedding vector which is used to get the binary mask prediction. It predicts a set of binary masks, each associated with a single global class label prediction. We use the MaskFormer model with Swin base backbone pretrained on ADE20k dataset, and finetune it on our dataset comprising 62 object classes.

The MaskRCNN model was trained on a distributed GPU setting, for around 60 epochs. Each training loop iterated through nearly 450K color and segmentation images.

Parameter	Value
Model Name	MaskRCNN
Hidden Layer Size (mask predictor)	512
Batch Size (per GPU)	8
Learning Rate	0.00125
Learning Rate Scheduler	Custom Linear LambdaLR
Optimizer	Adam & SGD
Epoch Decay	15
Decay Factor	0.1
Momentum	0.9
Weight Decay	0.005
Number of Epochs	60

Table 2: Training Details and Hyperparameters

The Maskformer model was also trained using parameters mentioned in [8] as well as on the official HuggingFace page. We ran multiple separate experiments with the MaskFormer model using our own custom pipeline, but this resulted in highly divergent behavior, without loss convergence. We are continuing to improve this model, through a more rigorous hyperparameter search. This is proving to be a challenge due to the size of the model, and the size of the training data. We would be taking up debugging the training process of the model once top-priority tasks are completed, for goal completion, as mentioned in 17.

8.3 Evaluation Metrics

We will adopt the following 5 metrics to evaluate the performance of our system -

- Customer Satisfaction Rating (CSAT) or Customer Experience (CX) score: Indicates the rating provided by customers based on the gameplay experience and the interaction with the bot. The primary metric used to evaluate the efficiency in the Alexa SimBot Challenge
- Success Rate (SR): A binary indicator of whether all sub-tasks were completed

- Goal Condition (GC) Success: the ratio of goal-conditions (or the state of objects in the environment) completed at the end of an episode
- Path length weighted SR (PLWSR): SR weighted by (path length of the expert trajectory)/(path length taken by the agent). Shorter path lengths, indicative of more optimal solutions, yield better path length weighted SR
- Path length weight GC (PLWGC): GC weighted by (path length of the expert trajectory)/(path length taken by the agent). Shorter path lengths, indicative of more optimal solutions, yield better path length weighted GC

9 Test Design

Rank	Avg Feedback Rating				Number of Conversations	Avg Duration of Conversations	
	L7d**	CI	Week-Ago	L1d	L7d	median	90th Percentile
1***	4.24	± 0.38	2.16	4.0	63	6:04	10:43
2	3.95	± 0.40	2.95	3.82	62	5:44	9:58
3	3.79	± 0.39	3.38	3.81	65	5:02	10:55
4	3.62	± 0.43	2.83	4.0	58	5:24	11:45
5	3.51	± 0.41	2.93	3.45	62	6:01	11:03
6	3.46	± 0.45	3.52	3.25	59	6:39	14:57
7	3.23	± 0.39	2.52	3.78	58	8:48	11:52
8	2.97	± 0.41	2.03	3.33	72	6:24	11:55
9	2.75	± 0.43	1.13	2.27	68	6:53	14:41
10	2.64	± 0.36	2.14	2.21	68	8:25	15:21
Average	3.42	-	2.56	3.39	63.5	6:32	12:19

** *Sorted by Feedback Rating over last 7 days*

*** *Your Team*

Figure 10: Simbot Alexa Challenge Rankings

Before deploying our instance segmentation models on the Arena environment, we determine its performance on the validation set of the vision data. Table 3 shows the performance of MaskRCNN for the validation set. The evaluation is carried our on game runs that are not seen in the training data to ensure there is no data overlap.

Due to the presence of multiple modules in our project, we need to ensure that each of these modules is functioning as expected before integrating them together. For this purpose, we perform unit testing on each of the modules and ensure that the underlying models are all performing as expected. Much of the unit testing is performed on the ML Toolbox using the data provided by Amazon. Once, we are confident of our individual modules, the next step is to test the combined system on the ML Toolbox, and finally, the system is tested on the Echo Show device by moving the code over to the Inference Server, which runs the Arena application.

In addition to our testing, our bot is extensively tested by the Amazon Alexa AI team, as well as by Beta Testers on a daily basis. This includes an average of 5 game runs per day by the Beta Testers and we get feedback from them based on each game run on a daily basis regarding improvements. Figure 10 shows the leaderboard we get at the end of each day showing how our bot performs at the internal beta testing. Thus, at the end of each working day, we look at the feedback file and make a note of the changes we need to make to improve our system. In addition to the feedback for the individual runs, and the mission completion rate, the key metric on which our bot is evaluated is the Customer Experience (CX) score provided by the Amazon internal beta testers. The CX score is assigned on a scale of 1.0 - 5.0 and depends on the overall user experience while interacting with the bot and completing the missions. This kind of iterative testing allows the exhaustive testing of our bot features and goes a long way in making SimBot robust. One important point to note is that a lot of the testing is focused on not just creating a functional and efficient virtual agent, but also on

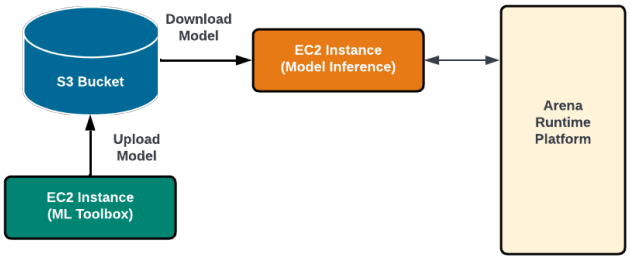


Figure 11: SimBot Deployment Model

improving the overall customer experience which is measured by the CX score. The goal is to create a system where the users learn incrementally while interacting with the bot and enjoy the overall game experience. At the end of the beta testing, as shown in Figure 10, because of our constant changes, we are placed on top of the leaderboard.

10 Deployment Model

Throughout the course of the capstone, as we tackled different multiples, we needed different deployment strategies to tackle these problems. For testing our bot's performance and execution, two different types of environments were provided to us by Amazon.

- **ML Toolbox:** The toolbox environment served as a sandbox environment where we did the testing and the initial deployment of the code. The Toolbox was deployed on an AWS EC2 instance which used an AWS Lambda to communicate with the Arena Application. The ML toolbox was also used to store the dataset and train both vision models. The model is then uploaded to S3 from where the inference service can use as shown in Figure 11. We also used the toolbox to parse through the data to create the static mapping of the data. The ML toolbox served as the offline testing tool for our deployment.
- **Inference Service:** After the testing of the code on the ML toolbox, the code was then deployed to the Inference Service, which was also hosted on the AWS EC2 instance. This machine served as an Online testing platform, as it enabled the bot to be tested by using the Amazon Echo device. The Beta testers and the folks from Amazon used this online testing method to test and give feedback to our bot.

11 Risks/Challenges

This section discusses the major risks and challenges involved in developing and deploying embodied task completion agents.

11.0.1 Product Risks

Incorrect predictions and actions can be severely harmful and even life-threatening in some scenarios. For instance, if an agent assisting a person with vision impairments provides incorrect actions, it can lead to accidents both indoors and outdoors. Similarly, an agent navigating incorrectly can collide with humans, or other objects leading to a loss of public order, safety risks, and potential economic losses.

11.0.2 Project Risks

The current frameworks like Arena have very limited documentation. Working with these frameworks requires understanding the limited documentation as well as thoroughly going over the code base.

Modifying the code for any version upgrades is a challenge. With the development of multiple frameworks for embodied AI, porting the code so that it works with most of them is challenging. For example, the Arena framework does not provide the status codes for all the batched tasks but only the last failing task, which might be different from other frameworks, and this in turn affects the queueing and batching code logic. Thus, the same code does not work across frameworks. There is also the additional risk of limited data availability since data collection for embodied AI tasks is a very complex process requiring sophisticated frameworks.

11.0.3 Regulatory Risks

Regulatory restrictions could be a serious concern for embodied AI tasks in the future. If there are increased cases of inconvenience to the public due to agents operating especially in outdoor environments, or instances where human safety is compromised and economic losses are incurred, it might not be surprising to see new laws restricting the overall use of these agents. There could also be privacy concerns caused by the hacking of these systems and their deliberate misuse to cause harm. Another potential issue could be data security wherein the data gathered by the agents such as semantic maps could be misused if made publicly available.

11.0.4 Business Risks

The risks mentioned above falling under Product, Project and Regulatory risks eventually culminate into business risks for embodied agents. Any issues regarding the correctness and efficiency of agents could adversely impact widespread adoption. Project risks that pose a challenge to researchers building these agents might discourage them from working on the problem and instead divert attention to other less risky propositions. In addition, issues with the law and regulations would discourage people from buying these agents, hampering business.

12 Tools and Dependencies

This section enumerates the tools and dependencies of our project. Where relevant, we also provide descriptions of these tools and the rationale for their use.

12.0.1 Datasets

- Vision Data - This corresponds to the metadata that contains the images and ground truth segmentation masks.
- Trajectory Data - The dataset contains the ground truth action trajectories along with the robot-view images.
- CDF Images - This provides the game initial state, game goal state, and game-related text.

12.0.2 Programming languages

- Python (3.7.10)

12.0.3 Libraries

- numpy - Python library to use and manipulate multidimensional array
- pandas - Open source data analysis and manipulation tool, built on top of the Python programming language
- opencvpython - OpenCV provides a real-time optimized Computer Vision library, tools, and hardware
- base64 - Python Module to work with base64 data
- h5py - It lets you store huge amounts of numerical data
- json - JSON encoder and decoder

- tqdm - Is used for creating Progress Meters or Progress Bars
- jinja2 - Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax.
- revtok - Reversible tokenization in Python.
- Pillow - The Python Imaging Library adds image processing capabilities to your Python interpreter.
- torch (1.8.0) - Pytorch is a python package that enables tensor computation
- torchvision (0.9.0) - Python module that provides access to model architectures, and common image transformations
- wandb - WandB provides the visualization and tooling needed for machine learning experimentation
- boto3 - Boto3 is the Amazon Web Services (AWS) Software Development Kit (SDK) for Python, which allows Python developers to write software that makes use of services like Amazon S3 and Amazon EC2.
- seaborn (0.9.0) - Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.
- imageio (2.6.0) - Imageio is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats.
- asyncio - asyncio is a library to write concurrent code using the async/await syntax.
- scikitimage - scikit-image is a collection of algorithms for image processing.
- types - This module defines utility functions to assist in the dynamic creation of new types.

13 Results

We have been provided with a set of five missions on which our bot will be evaluated until the end of the internal beta testing. As it stands, our bot is able to complete all five of these missions with a wide variety of different trajectories. We are able to successfully generate intelligent dialogs for prompts and clarifications grounded in the scene to help users complete tasks.

This section details the results of the instance segmentation model. The MaskRCNN model was evaluated on a test dataset of 60,000 images obtained from the latest build of the Arena simulator. The model was evaluated using two different metrics, `uberMAP` and `cocoMAP`, on all classes, as well as individual subsets of small, medium, and large classes. Classification of small, medium, and large is done as follows:

1. `small`: $0 <$ area < 1296 pixel squares
2. `medium`: $1296 <$ area < 9216 pixel squares
3. `large`: $9216 <$ area < 90000 pixel squares

MaskRCNN Model Evaluation Results						
Number of Classes	uberMAP			cocoMAP		
	Small	Medium	Large	Small	Medium	Large
86	0.914	0.921	0.849	0.545	0.672	0.690

Table 3: Evaluation Results

14 Error Analysis

One of the significant errors that occurred in our system was that small objects were not detected accurately by our instance segmentation models. These could be objects like Apple, Floppy Disk which are small in size are frequently missed by the MaskRCNN model. To resolve this, we finetuned MaskRCNN on these small classes explicitly, and performed training for a larger number of epochs. A limitation of our system is that it is not able to find small objects across rooms. These objects can only be detected in the same room. Say for example, the user is in the breakroom, and requests to go to the computer. Since, there is no computer in the breakroom the command would fail and the bot would not search for the computer in rooms other than the one in which the user is currently present.

15 Discussion

The project consists of a research component and a software engineering component. The research components comprise working to create the best-performing instance segmentation model with the most optimal thresholds for each of the object classes and developing a model for object color identification. It involves identifying the best strategy for path planning keeping into consideration the Arena simulator constraints and providing the best user experience. The software engineering component involves developing the multi-queue coding paradigm for implementation of the path planning and action execution logic, handling sessions and game state using DynamoDB for multi-user concurrent game plays, and adding clarifications such that the bot can interact with the users to help it complete missions, performing error handling among others. The next steps would include making our clarification system more robust so that it can handle a wider set of scenarios, incorporating depth and coordinate information for multi-object detection, and better attribute-based object understanding such as handling attributes like ‘bigger’, ‘farther’, ‘object to the left’ etc.

16 Lessons Learned and Reflections

One of the major challenges encountered during the challenge is the changing requirements and code structure from Amazon periodically. There were bugs and issues in several segments of the pipeline, which were rectified by Amazon as and when they were reported. In addition, they added new functionalities based on the mission requirements based on feedback from competitors. Due to this, we focused extensively on developing a modular and robust code that could easily be adapted to the changing requirements. Another major learning as part of the capstone project was on strong and effective inter-team collaboration with the SimBot Language and Dialog team. This involved understanding the highly convoluted requirements of the team, determining the best communication channels, arriving at the most efficient code communication formats, working with the owners of individual components for adding functionality and handling bugs, integrating our code bases together, and discussing the improvements to our pipeline to provide the best user experience.

As a team, we have established open, frequent and effective channels of communication with each other as well as with our mentors. This has led to effective collaboration on challenges and blockers and improved our productivity.

So far, we have been able to stay on track with our planned milestones and the Amazon deadlines and are on track with our submissions to the SimBot Challenge and also finished Beta Testing at the top of the leaderboard.

17 Future Work

As it stands, our bot is capable of responding to a wide range of instructions and generating a diverse set of useful dialog prompts to enable users to successfully complete missions on the Arena Runtime Platform. However, there are some challenges that prevent our bot from further improving the user experience. One such challenge is the bot’s inability to handle the disambiguation of multiple

instances of the same object type if the distinguishing factor is not color. For example, if there are 3 monitors of the same color in the bot's field of vision, our bot is not equipped with the ability to ask a clarifying question based on spatial location. Future work includes incorporating models like ReCLIP [14] that are capable of grounding referring expressions to images. The Arena Simulation Engine is also supposed to provide depth images for the current field of view of the bot, however, due to the buggy implementation of this feature on Amazon's side, we have been unable to incorporate depth information into our pipeline. This feature will be available to us in the near future, and if time permits we will incorporate it into our flow.

18 Conclusion

This report summarized the Navigation and Interaction pipeline of the SimBot family of projects. While existing solutions for embodied task completion agents are non-conversational in nature, our solution proposes the use of dialog to develop an agent that can ask questions to seek clarification or more information to successfully complete tasks. We have successfully built a SimBot and integrated it into the Arena Runtime Environment. Soon, the bot will be accessible to the general public to play with. The bot we have built along with the dialog team is capable of handling a wide range of tasks specified by a diverse set of language instructions. Our contributions lie in using ego-centric images to guide action planning and dialog generation to cooperatively complete tasks with a user of the bot. Having hit several milestones of the SimBot challenge, the next stage of our work will be geared towards further improving our Customer Experience scores through the planned improvements described in our future work.

19 Acknowledgments

We would like to thank Prof Yonatan Bisk for guiding us throughout the planning phase of our capstone project and also for being the bridge between the team and Amazon. Prof also took up the task of setting up the environment and the inference service during the summer, when the team was not available. Also, we would like to mention the mentoring of Jonathan Francis, So Yeon Min (Tiffany), and Jimin Sun. Finally, would also want to acknowledge the sponsorship given by Amazon and also allowing us to be a part of the Alexa Simbot Challenge. They are also crucial in giving us constant feedback and support to improve our bot.

20 Terminology, Definitions, Acronyms, and Abbreviations

- **Simbot** (A simulation tool for autonomous robots) - An autonomous robot simulation tool
- **FILM** (Following Instructions in Language with Modular Methods) - A modular framework for embodied task completion [5]
- **TEACh** (Task-driven Embodied Agents that Chat) - A dialog-based dataset for embodied task completion [3]
- **Receptacle Objects** - The set of objects that are encountered in the environment that is always located in the same room across multiple layouts and game sessions. Eg Refrigerator, Lasor, and Microwave
- **Non-Receptacle Objects** - The smaller objects that can be spawned in different places and rooms across multiple layouts and game sessions. Eg bowl, cup, and can.
- **CDF** (Challenge Definition Format) - Files to store game mission information.
- **IOU** (Intersection over Union) - The term used to describe the extent of overlap of two boxes. The greater the region of overlap, the greater the IOU.
- **FPN** (Feature Pyramid Network) [9] - The feature extractor that takes a single-scale image of arbitrary size as input, and outputs proportionally sized feature maps at multiple levels, in a fully convolutional fashion.

- **RPN** (Region Proposal Networks) [10] - The fully convolutional network that simultaneously predicts object bounds and objectness scores at each position.

21 References

[1] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks, 2020.

[2] D.J. Turnell, Masuma Fatima, and Q.V. Turnell. Simbot—a simulation tool for autonomous robots. volume 5, pages 2986 – 2990 vol.5, 02 2001.

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

[4] Piotr Mirowski, Matt Grimes, Mateusz Malinowski, Karl Moritz Hermann, Keith Anderson, Denis Teplyashin, Karen Simonyan, koray kavukcuoglu, Andrew Zisserman, and Raia Hadsell. Learning to navigate in cities without a map. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[5] So Yeon Min, Devendra Singh Chaplot, Pradeep Ravikumar, Yonatan Bisk, and Ruslan Salakhutdinov. Film: Following instructions in language with modular methods, 2021.

[6] Alexander Pashevich, Cordelia Schmid, and Chen Sun. Episodic transformer for vision-and-language navigation, 2021.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[8] Bowen Cheng, Alexander G. Schwing, and Alexander Kirillov. Per-pixel classification is not all you need for semantic segmentation. 2021.

[9] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[10] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[11] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[12] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 633–641, 2017.

[13] Holger Caesar, Jasper Uijlings, and Vittorio Ferrari. Coco-stuff: Thing and stuff classes in context. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1209–1218, 2018.

[14] Sanjay Subramanian, Will Merrill, Trevor Darrell, Matt Gardner, Sameer Singh, and Anna Rohrbach. Reclip: A strong zero-shot baseline for referring expression comprehension. *arXiv preprint arXiv:2204.05991*, 2022.

A Appendix

A.1 Error Handling

The Arena Application provides different error types to the bot, the Error Handler module processes these errors and sends back intelligent dialog responses to the user to help them understand the issue

and what they can do to fix the issue and complete the mission. The different sets of errors and the corresponding responses provided by the Error Handler are presented in Table 4.

Error type	**Response**
UnsupportedAction	The action that was requested is not supported
UnsupportedNavigation	A location in my path is not accessible. Could you give me a different navigation command?
AlreadyHoldingObject	Sorry, I cannot pickup another object. Try putting down the <object_in_hand> first.
ReceptacleIsFull	Sorry, I cannot complete the action as the <target> doesn't have enough space for it.
ReceptacleIsClosed	Sorry, I cannot complete the action as the <target> is closed. Try opening it first.
TargetInaccessible	Sorry, I can't reach the <target>. Can you guide me closer to it?
KilledByHazard	Oh no! I was killed by a hazard. Watch out for sparks on devices and electrified puddles!
TargetOutOfRange	The <target> is too far away for me to reach. Can you try saying go to the <target> first?
AlternativeNavigationUsed	I couldn't get to the exact spot you wanted me to navigate to, but I got close
ObjectUnpowered	The <target> is not powered. Can you try going to the fuse box and turning on the power by opening it and saying toggle the lever?
ObjectOverloaded	The <target> is overloaded. Can you try going to the fuse box and turning the power off and back on by opening it and saying toggle the lever ?
InvalidCommand	Sorry, something went wrong!
ObjectNotPickedUp	I am not holding the <target>. Can you try telling me to pick it up first?
ArenaUnavailable	I'm still booting up. Can you try again in a few seconds?
InvalidActionFormat	Sorry, something went wrong!
ActionExecutionError	Sorry, I couldn't complete the action

Table 4: Error Types and corresponding Error Responses provided to the user

A.2 Primitive Instructions

Arena Application can take a defined set of primitive institutions as input from the Action Processor as defined in Table 5

B Changes To Previous Deliverables

In the Fall, the specifications of the SimBot challenge changed drastically from the initial phases of the competition. The simulation environment, the dataset, and the SimBot Runtime Platform itself underwent drastic changes. As a result, the set of tasks we had to work on to develop the Navigation and Interaction pipeline changed as well. Our previous focus was in adapting FILM, the state of the art of embodied task completion, from ALFRED to the TEACh dataset. Since our work now uses an internal Amazon SimBot dataset, our previous work was not usefult to the new challenge specification. The new Arena Runtime Platform has also exposed a set of APIs for executing low level actions on the simulator as a result of which some of our tasks such as executing point to point navigation were greatly simplified. These changes to the competition specification meant that our previous deliverables differ drastically from the work reflected in this report.

Instruction Type	Action
Navigation	MoveForward
	MoveBackward
	Rotate Right
	Rotate Left
	LookDown
	LookUp
	LookAround
GoTo	GoTo Room
	GoTo Viewpoint
	GoTo Mask
Interaction	PickUp
	Open
	Close
	Break
	Scan
	Examine
	Place
	Pour
	Toggle
	Fill
	Clean
	Highlight

Table 5: Primitive Instructions supported by the Arena Application