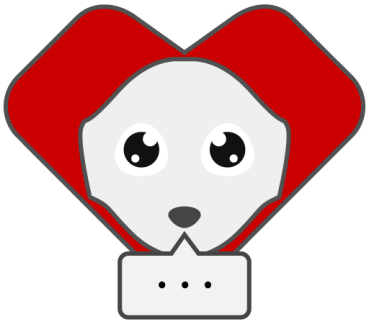
PepperTale: Interactive storytelling with Pepper



Report – Elective in AI (HRI & RA)

Department of Computer, Control and Management Engineering Antonio Ruberti
Sapienza, Rome, Italy

Students:

Pasquale Silvestri *(1749274)*

Marco Lo Pinto *(1746911)*

Academic Year 2022/2023

Summary

Introduction

In the rapidly evolving landscape of artificial intelligence and robotics, the integration of robots into social environments has emerged as a new and compelling experience. These reasoning agents are not just relegated to factory floors, laboratories or sometimes football fields, because they are now making their presence felt in our daily lives, in our homes, schools, healthcare facilities, and public spaces. The significance of this shift cannot be overstated, as it marks a fundamental departure from the traditional notion of robots as mere mechanical tools. Instead, they are increasingly assuming roles as social agents, capable of rich interactions and deep engagement with humans, even on a more emotional level. This evolution has profound implications for society, affecting not only how we work and live but also how we perceive and relate to the machines that share our spaces. Entertainment for the mere purpose of entertaining is the furthest possible application of these machines with respect to their usual settings, but a one-to-one interaction allows for a more personalized way of entertaining, much harder with normal human interactions.

We've named our project PepperTale, a clever play on words that reflects the robot's primary purpose: storytelling. Unlike typical scripted storytellers, PepperTale is an intelligent agent that collaboratively builds narratives with users. While the robot has its own narrative objectives, users engage in a creative tug-of-war to shape the story's outcome.

Pepper is able to craft a compelling story together with the user, telling it with words while mimicking with its body the mood and spirit of the events happening in the story. Every action chosen by the robot is a direct consequence of the user selected action, and is chosen intelligently to get to one of its wanted goals while counteracting the user actions.

The robot uses speech recognition to receive the input from the user and uses the camera to do face detection and recognition, so to be able to address each

user in a more personalized way. Finally, a screen is used as secondary aid to the interaction, showing in textual form what the robot says and giving feedback for the speech and face recognition.

We affirm that each student has made an equal contribution to this project.

1 Related Work

The work that we will present aligns with the broader trend of integrating emotional expression into HRI and exploring the potential of robots as social agents, capable of not just interacting but also engaging in meaningful and emotionally Reasoning Agents. Through the integration of personalized interactions and story co-creation, our system enables robots to go beyond simple information processing and engage users in immersive narrative experiences.

1.1 Human Robot Interactive stories

In the domain of interactive storytelling, which has been proposed by [1] as a standard activity for robots in the near future, it generally follows a well-structured format, typically involving an introduction, climax, and evaluation. During the evolution of the story, Selting [2] demonstrates that speakers tend to employ different ways of emotional indicators (such as speech rate, gestures and/or body movements) as the story progresses towards the climax. Nevertheless, as we can see from the papers collected by [3], examination reveals a notable gap in the recognition of the interactional context as a crucial factor in determining suitable emotional displays in robots. Existing literature in human-robot interaction largely overlooks the notion that emotional expressions are frequently exhibited due to societal expectations rather than genuine emotions experienced by the robot. To cite the authors of the paper: "Emotional expressions are expected rather than felt".

Our architecture aims to consolidate these observations in order to enhance human-robot interactions by understanding and effectively incorporating socially determined emotional displays in the field of interactive storytelling. By using Pepper as the base robot for our system, in combination with Computer Vision, Text-To-Speech, Speech Recognition and many more techniques, we are able to effectively create an interactive environment for the end-user.

1.2 Reasoning on the evolution of a story

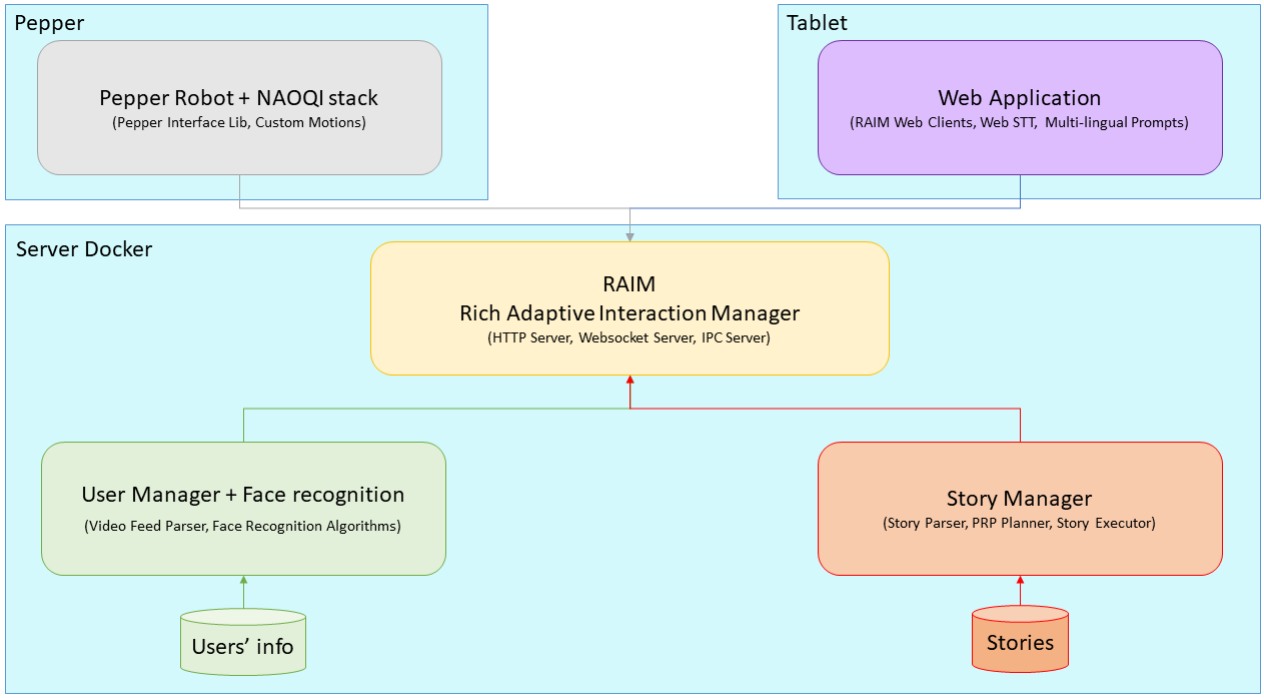
Traditional storytelling, by its very nature, represents a unidirectional flow of narrative where the speaker, be it a human or an entity like a robot, serves as the sole active agent while the user assumes a passive role as the audience. In this conventional approach, the user's role is primarily that of an observer, often limited to listening or reading, with minimal or no agency to influence the narrative's course. In stark contrast, interactive storytelling with reasoning agents fundamentally alters this paradigm. It transforms the user from a mere audience member into an integral participant within the narrative fabric. In this new dynamic, the user is no longer a passive recipient of the story but an active contributor, shaping the story's trajectory through their choices, decisions, and interactions. This shift from a one-way directional narrative to an immersive, interactive experience redefines the user's role, making them an essential element of the storytelling process. In order to do so, we need to compute policies for fully-observable non-deterministic (FOND) planning problems, which are the stories generated with the interaction of the user with the robot. The authors of the paper [4] proposed a planner called PRP that incorporates FOND and online probabilistic planning techniques to build a strong cyclic solution. They introduce improvements based on state relevance that make the planner more efficient and capable of generating more succinct policies. The significance of this paper for interactive storytelling lies in its ability to handle non-deterministic aspects of the environment, which is crucial for creating engaging and dynamic narratives. In interactive storytelling, the actions and decisions of the user often lead to different outcomes and branching paths. By incorporating non-deterministic planning techniques, the PRP planner can model and simulate multiple potential outcomes, allowing for a more realistic and diverse narrative experience.

2 Solution

The architecture of our solution is a modular client-server architecture. Individual features are built as standalone components that can interact with one another using a central server, RAIM.

This approach is intended to enhance development speed, robustness, elasticity and extendibility.

RAIM (Rich Adaptive Interaction Manager) is the core of the solution. It allows the different modules to talk to each other seamlessly, abstracting the underlying platform, development stack, and even the host machine, demonstrating its versatility. For instance, it's possible to have the UI built with web technologies and deploy it on an external tablet, the face recognition network running on a powerful remote machine powered by python 3 and the scripts talking to pepper in a python 2 environment.



2.1 RAIM

RAIM stands for Rich Adaptive Interaction Manager. The initial idea was to use the provided MODIM, but unfortunately it wasn't the right fit for how we wanted to handle all the features. We needed a more elastic solution that could run on a different machine, on python 3 and using modern web technologies like websockets. Moreover, we needed more freedom in the interaction development. Nonetheless, MODIM is a very important inspiration for the development of RAIM.

RAIM's operational model is predicated upon a modular framework that comprises two distinct categories: Server modules and Client modules.

A server module is an abstraction of the underlying channel and/or platform. Multiple client modules are connected to each server module, and each client module is entrusted with the management of a specific functional aspect of the project.

The primary responsibility of a server module is to facilitate inter-client communication, thereby enabling the relay of requests initiated by one client to be transmitted to another client, or broadcasted to all clients if explicitly specified. It is important to specify that the communication is possible among all client modules, irrespective of the server module they are connected to.

The modules communicate with one another using a standardized custom communication protocol. This protocol is based on what we call "RAIM commands". In this context that the act of relaying requests by server modules is denominated as "command dispatching."

A RAIM Command is a packet containing those fields:

- id: The unique packet identifier
- to_client_id: The unique id of the receiving client module
- from_client_id: The unique id of the sending client module
- request: A boolean flag referring whether the sender expects a response to this request or this packet is a standalone command

- data: The data field of this command. Is a JSON object so can contain any type of data with no limitations on byte size or character length.
- is_succesful: A boolean flag used to characterize responses with no returning data. Is essentially an acknowledgement flag

The packet has no size limit, the receiving and recomposition of the full packet is one of the server's jobs.
Another one of the server's jobs is to allow the targeted delivery of commands to the designated client, as indicated in the "to_client_id" field. This is regardless of which server module the receiving client module is connected to. Consequently, if the recipient client is directly linked to the same server module as the sender, it will immediately receive the command. In cases where the recipient is not connected to the same server module as the sender, the server will then transmit the command packet to all other servers, which will subsequently ensure the correct distribution of the command. This gives rise to two types of dispatching. The first one is the primary dispatch, that is the relaying of a command to a client done by the server directly receiving the command from the sending client. The second one is a non-primary dispatch, that is the relaying of a command received by another server. The primary distinction between the two types of dispatch is that a primary dispatch broadcasts the received command to all clients directly connected to the originating server and to all other servers, while a non-primary dispatch confines the distribution of the received command solely to its linked clients, thereby preventing an infinite loop of servers passing commands among themselves.
For the project we developed three server modules: HTTP Server, IPC Server and Websocket Server.
The HTTP Server abstracts the HTTP communication. This allows one or more browsers to connect to PepperTale using a tablet or any other web-capable device.
The IPC Server abstracts the communication between multiple python-based client modules. The python clients can be built on either python 2 or python 3,

with no loss in capabilities. The IPC server uses TCP sockets as underlying channels for the RAIM commands.
The Websocket Server abstracts the communication between multiple javascript-based client modules. It uses websockets as underlying channels.
For what regards client modules, they are the interface between a specific feature and the rest of the architecture. Each client exposes a set of APIs, taking care of packaging a request for a needed feature or a response for a specific feature request. For example, the story manager client module exposes a certain number of story related APIs, one of them being listing all the available stories and whether they are suitable for a user of a certain age.
For the project we developed three client modules exposing specific API: Story Manager Client, Face Recognition Client and Pepper Bot Client, each one with a corresponding consumer client, that makes use of the related APIs.

2.2 PepperBot: Pepper robot interface

To communicate with the robot, we developed PepperBot: an easy-to-use interface that can well-integrate in our infrastructure. By specifying the ip and port of the robot, it will use the NAOqi APIs to connect to all available services of the robot, such as ALTextToSpeech, ALMotion and so on. Once connected, the user can modify different behaviours of the robot, such as adjusting voice volume and speed for a softer or more assertive tone or altering the robot's behavior to make it appear more natural, with actions like looking around rather than remaining stationary. Different functionalities are implemented in order to express the robot to its fullest: the speech, the movement of the joints of the robot to execute, the colors of the eyes and a variety of sensors (sonar, touch, laser). Moreover, this system comes with ready-to-use complex motions that can express a wide range of emotional expressions. Each method can be called in a synchronous or in an asynchronous way thanks to the usage of threading, thus allowing different combinations of actions.

In order to later use it as an external service by simple API call through the internet, the interface was wrapped in RAIM client module, PepperServer: it uses both PepperBot and the RAIM infrastructure in order to exchange information with the other services in our system with a standardized interaction policy. It will listen to commands that have as data a list of actions for the robot and computes each of them. Each action has two fields: action_type which is the name of the action that the robot must perform, and action_properties which are additional information needed to execute the action.

2.3 Face Recognition

When people interact with each other, different behaviors can arise depending on who are they talking to: if it's someone they know they will have a different approach with respect to someone they already met. The same procedure must be applied also to our robot: that is why we needed to use a face recognition system to do reasoning and differentiate between new users and experienced users, as well as to understand if the user is currently interacting with the robot or not. Our approach is based on the face-recognition python library, built using dlib's state-of-the-art face recognition. The module developed is initialized with two parameters: "resize value" and "unknown face threshold". The former is used to scale down the image in order to improve the speed of the response, the latter is used to set the number of frames to wait when detecting a new face in order to classify it as an unknown user (this is done to prevent false negatives when detecting a known user as unknown).
When the recognition system is called to process a frame captured from a video, it first resizes it by the resize value passed on the initialization step, then finds all faces in the current frame, computes the embeddings and, for each of them, checks if it's a face that the system has already seen: if that's the case, then the detected face is marked as known and the name of the user (and any other additional information of that specific person) is identified. Instead, if the

face is not recognized, it is marked as a possible unknown face. If the same unknown face is still in the other frames (so he/she didn't leave the view of the robot) after the number setted with the variable "unknown face threshold", then the face is marked as unknown face. At the end of the computation, the info of the known faces and the faces of the unknown people are returned in order to be used as information for the reasoning part of the robot.
To save information regarding the unknown people detected, the module also exposes a method to do it, so that in a future encounter they will be detected as known people.
Moreover, this system is wrapped in the same fashion as PepperBot, in order to expose and be used in the RAIM infrastructure.

2.4 Story Manager

The Story Manager module is a client module that exposes all the APIs related to the storytelling. The module is made up of various parts, the Story Parser, the Story Executor, the endpoint functions and the PRP Planner.
This component is the one implementing the actual reasoning for storytelling. The PRP Planner (Planner for Relevant Policies) is a Fully Observable Non-Deterministic (FOND) planning model. By writing the story in PPDDL (Probabilistic PDDL), this planner is able to compute the best sequence of actions to achieve the goal. We needed a nondeterministic planner to have an interactive experience, because the user, that is considered part of the environment, can take one of many possible actions at each step, and each time the agent should be able to adapt and replan for the new state of the environment. The planner is thus used as an external tool to compute the possible solution to the problem. PRP is very powerful, one of the few planning models supporting the *"oneof"* nondeterministic PDDL directive. But it still has some major limitations. The most prominent one for the way we wanted stories to be, is the lack of support for the *"or"* statement in the goal definition, and the ability to set a priority for the different goals.

While defining a story that can evolve in unpredictable ways, you want to be able to establish a set of goals and have them be ordered by priority. This is because, during the execution, the main goal might become suddenly unfeasible. In this kind of situations, the agent has to quickly move onto trying to achieve a secondary goal that might still be on the table, instead of just giving up (as the planner would do if the goal was only one and wasn't reachable from the current state). This makes for a more interesting story and a more natural interaction.
To work around this issue, we cleverly designed the Story Executor in a way that allows to have multiple goals with priorities. But before talking about the executor, it is better to describe the parser.
The system supports any number of stories. Each story is kept in its own folder as a JSON file designed to make easy writing stories with support to multiple languages, goals and nondeterminism.
The first field we can see in the JSON is the multilingual values set:

```
"values_set": {
    "text0": {
        "en-US": "Action 0 gets done",
        "it-IT": "Viene fatta l'azione 0"
    }
}
```

It simply is a dictionary with text values for multiple language.
Next there is the definitions of the atoms:

```
"atoms": {
    "main_goal": false,
    "secondary_goal": false,
    "attribute_0": false
}
```

Atoms are Boolean attributes that can be set and unset by the actions. They can be used as goals or to define a story knowledge base. For example, if a character has the ability to get a tool and use it later, an atom can represent the owning of such tool and thus allowing the execution of the action that requires that tool.

The goals using those atoms are defined in a list:

```
"goal": [
    ["and", [ "atom", "main_goal", true ], [ "atom", "another_goal", true ]],
    [ "atom","secondary_goal",true ]
]
```

The list represents the descending priority of each goal. The atoms can also be combined with an *"and"* to compose a more articulated goal. Then we have the actions:

```
"actions": {
    "action0": {
        "nondeterministic": false,
        "pretext": "pt0",
        "text": "t0",
        "preconditions": {
            "attribute_0": true
        },
        "effects": {
            "secondary_goal": true
        },
        "mood": "happy"
    }
}
```

Every action is represented as an object that comprises both the visible text displayed on the screen and a condensed version called the "pretext" which appears on selection buttons. The nondeterministic attribute flags this action to be treated as a user action. Preconditions is a list of atoms required to make this action executable, while effects is a list of atoms set by the execution of this action. Finally there is the mood of the action. This will be mimicked by Pepper during the storytelling.

The actions are connected in a tree-like structure represented by the actions_children field:

```
"actions_children": {
    "action0": ["action1","action2"],
    "action1": [],
    "action2": []
}
```

Per each action there is a list of possible subsequent action children.
The last attribute within the JSON is the initial action:

```
"root_action": "action0"
```

The parser ingest the JSON file, constructs the tree structure, taking the selected language into account, and writes the PDDL domain file along with multiple problem files. Each problem file corresponds to one of the goals and will be used by the story executor in the requested order by priority.
To constraint the planner to choose only the allowed actions for that state of the story, "control predicates" are generated and associated to each action.
An example of this PDDL strategy is this one *(consider action1 and action2 nondeterministic)*:

```
(:action action0
    :parameters ()
    :precondition( and (do_action0) (attribute_0))
    :effect(
        and (not (do_action0)) (oneof (do_action1) (do_action2)) (secondary_goal)
    )
)
```

Ultimately there is the Story Executor which also functions as a RAIM client module, thereby making all its functionalities accessible to the broader infrastructure. It allows to list all the available stories, with the ability to request the identification of stories unsuitable for a user based on their age, enabling subsequent notification. Once a story is selected, it can be initiated and, once per turn the executed actions, in conjunction with the forthcoming potential user actions are returned. A more comprehensive description of the whole story telling interface and interaction is better described in the results. In summary, pepper will narrate all actions performed by itself and those chosen by the user, until a conclusion is reached.
The exposed APIs employ data packets featuring attributes such as "action_type" and "action_properties," adhering to the established pattern shared with PepperBot and Face Recognition.

2.5 Web Application

All the components, tools and services that have been explained can be used to create any human-robot interaction with reasoning agents. Thus, this project can be considered as a fully finished framework to use as a starting point for any application. In order to do so, the developer will create a web application using RAIM's HTTP web server and vanilla JavaScript, HTML and CSS. To ease the process of creation, many utility web components were developed:

- Routing: Within the RAIM framework, one of the central elements that greatly enhances the user experience and interaction is the routing mechanism. Routing refers to the process of determining which view or component should be displayed to the user based on the URL or user interaction. It plays a pivotal role in creating seamless and intuitive user interfaces. Routing in traditional multi-page applications often involves full page reloads when navigating between different sections or views. This approach can introduce delays and disrupt the user experience. However, with Single Page Application (SPA), routing becomes a highly efficient and dynamic process. SPAs load the initial page and subsequently update content dynamically without requiring the entire page to be reloaded. This characteristic aligns perfectly with the goals of modern applications.
- Speech Recognition: Sometimes the user cannot interact directly with the robot, maybe because it is in a distant environment, or the quality of the robot's microphone is not good enough or even not present at all. For all these cases, the module to provide speech recognition using an external device, such as a tablet, can be used. This service is based on the Web Speech API and enhances it with a promise-based interaction with timeout, so that the robot can listen to the user and interact accordingly.
- Multilingual prompts: When developing an application, one of the key considerations is catering to a diverse user base. In an increasingly globalized world, supporting multiple languages is essential for reaching

a broader audience. The module addresses this need by allowing developers to easily incorporate text and audio prompts in various languages within their applications. Developers can create language-specific prompts for user interactions, notifications, error messages, and more. These prompts can be stored in a structured format, making it easy to update and expand the supported languages as the application grows.

Additionally, to interact with each module in the infrastructure, there are many client modules exposing specific APIs:

- RAIM Client: It's the standardized library that is used to connect to RAIM and send/receive RAIMCommand objects. It is used to facilitate communication between clients through WebSockets.
- Pepper Client: This module can be used to control and interact with the Pepper robot. It exposes lots of different methods to send requests and receive responses from various actions, including speech, movements, LED lights, video capture and so on, via a WebSocket-based communication protocol.
- Face Recognition Client: As the name suggests, it connects with the face recognition system hosted on the remote machine, sending frames in order to be analyzed and receiving responses for the known and unknown people present in the view. It also provides methods to save new faces, giving them a name and other information such as age and sex.
- Storytelling Client: This module helps to interact with the core part of this project, which is interactive storytelling. It enables the user to list the stories, start a new one and perform actions in the story.

The services we've outlined are seamlessly integrated with the async/await paradigm. This empowers developers to craft asynchronous code in a highly sequential and easily comprehensible fashion. At its core, async/await liberates us from the limitations of blocking code, enabling the smooth handling of tasks like I/O operations and network requests without creating performance

bottlenecks. In the context of our application, this capability proved pivotal. It allowed us to advance the robot's interaction with the user while concurrently monitoring the user's presence within the robot's field of view, ensuring a responsive and engaging user experience.

3 Implementation

In this chapter we are going to see the tools, libraries and technologies we used in order to develop our solution. The following technologies are all free to use and many of them are also open source.

3.1 Dlib's Face Recognition

Dlib's face recognition library is a powerful and versatile open-source software toolkit designed for facial recognition and facial feature analysis. Developed by Davis E. King, this C++ library offers a wide range of capabilities for detecting and identifying faces within images and videos. The library is highly accurate and efficient, making it suitable for real-time applications and provides a Python wrapper for ease of use.
Dlib uses a deep learning model to extract facial features from images. These features are then used to create a unique identifier for each face. This identifier can then be used to compare faces to a database of known faces to identify or verify individuals.
The library can be used to perform a variety of tasks, including:

- Face detection: Identifying the location of faces in an image
- Face recognition: Identifying individuals based on their facial features
- Face clustering: Grouping faces together based on their similarity
- Face verification: Verifying that a person is who they claim to be

In our project, this library is the heart of the Face Recognition module and proves to be highly valuable in the context of reasoning.

3.2 IPC Communication

Inter-Process Communication (IPC) is a fundamental concept in computer science, enabling processes or applications to exchange information and collaborate. One of the most versatile and widely used methods for IPC is through sockets. Sockets serve as communication endpoints, offering a standardized and efficient means for processes to connect, communicate, and

share data. This method is crucial for the modular infrastructure we built, allowing multiple python-based script to communicate seamlessly. It has the added benefit of not constraining us to a single machine and thus being able to run a module on pepper, while the rest is on a more powerful computer.
It has the disadvantage of being a barebone TCP connection, a low level type of channel, requiring a more complicated handling of the data passing through it.

3.3 Flask HTTP Server

Flask is a lightweight web framework for Python, known for its simplicity and minimalism. It offers flexibility without imposing strict structures or dependencies. Developed by Armin Ronacher, Flask empowers developers to create web applications quickly and easily.
This framework is extensible, allowing developers to add functionality using third-party libraries and extensions. It features a built-in development server for local testing and uses the Jinja2 templating engine for dynamic HTML pages. Flask simplifies URL routing and is particularly suitable for building RESTful APIs.
Is this simplicity that allowed us to build the HTTP Server module of the RAIM infrastructure and consequently the Web UI interface to complement the Human-Robot Interaction.

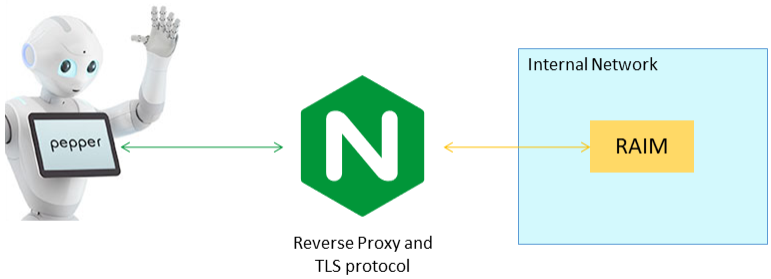
3.4 WebsocketServer Lib

WebSockets are a communication protocol that enables real-time, bidirectional data exchange between a client and a server over a single, long-lived connection. Unlike traditional HTTP, which follows a request-response model, WebSockets allow both the client and server to send messages to each other asynchronously. This makes this technology ideal for applications that require low-latency, interactive communication, such as sending commands to a talking robot.
The "WebSocketServer" Python library is a specific implementation of a WebSocket server that simplifies the process of creating WebSocket-based

applications in Python. This library provides the necessary tools to set up and manage WebSocket connections.
Within the scope of our project, this library played a pivotal role in effortlessly constructing the WebSocket server module.

3.5 NGINX Reverse Proxy

Nginx is a high-performance, open-source web server and proxy server software widely used for serving web content and managing network traffic. Big names like Netflix, ING, Adobe, Cloudflare, WordPress and others. It's known for its speed, scalability, and efficiency in handling a variety of tasks in the realm of web hosting and networking.
One of Nginx's notable features is its reverse proxy functionality. A reverse proxy is a server that sits between client devices and a web server, acting as an intermediary to handle incoming client requests and directing them to the appropriate backend server.
Leveraging Nginx is a convenient way to implement proxying and more importantly adding TLS support. This is due to the requirements of many contemporary web-based technologies, such as the Web Speech API, which mandate a secured connection. To implement TLS, port routing, efficient static file serving and the hosting of the web application on a readily accessible domain, we opted for Nginx as proxy server. The simplified schematic of the network architecture is in the figure.



3.6 Web Speech and MediaDevices API

To provide users different types of feedback and different modalities of human-robot interaction, we used the SpeechRecognition API, which allows

JavaScript to have access to a browser's audio stream and to convert it to text, in combination with the MediaDevices API, which grants the ability to connect to media input devices like cameras and microphones as well as screen sharing. In this way, the user (and the developer of the web app) can choose to either directly use the services provided from the web application's components or the PepperBot library to interact with the robot in an intelligent way. For security reasons, these web features can only be used with a TLS protocol. Nonetheless, this is not a problem for our environment, which already supports this type of communication.

3.7 NAOqi APIs

At the core of our PepperBot module lies the NAOqi API. It is developed by SoftBank Robotics for programming and controlling humanoid robots, with a primary focus on their NAO series of robots, including NAO and Pepper. This library provides a comprehensive set of tools and APIs to interact with, manage, and customize the behavior of these robots. NAOqi is designed to simplify the development of applications that enable robots to perform a wide range of tasks, from basic movements and interactions to complex cognitive tasks.

However, it comes with certain limitations when compared to the qi library, which is a broader software framework that encompasses NAOqi but extends its capabilities to a wider range of robots and robotic systems. Consequently, we have developed an alternative version of PepperBot that leverages this library. This implementation maintains consistency by offering the same set of methods as the previous version, ensuring seamless backward compatibility for both modules.

4 Results

Upon launching the RAIM server and initializing all the essential services required for our application, users are just a single web page connection away from embarking on their interactive journey with the system.

Our developed application is primarily structured around an internal state mechanism. This intricate design empowers the robot to not only comprehend but also retain information arising from interactions with various users. Moreover, it is divided into multiple phases of interactions, thereby simplifying the problem-solving process by breaking it down into smaller, manageable segments.

When the web app is accessed for the first time, it will first check and load all the services available. For instance, it will attempt to utilize the device's camera for visual data acquisition. In cases where this option is unavailable, the application gracefully resorts to connecting with the robot's camera system, ensuring uninterrupted functionality.

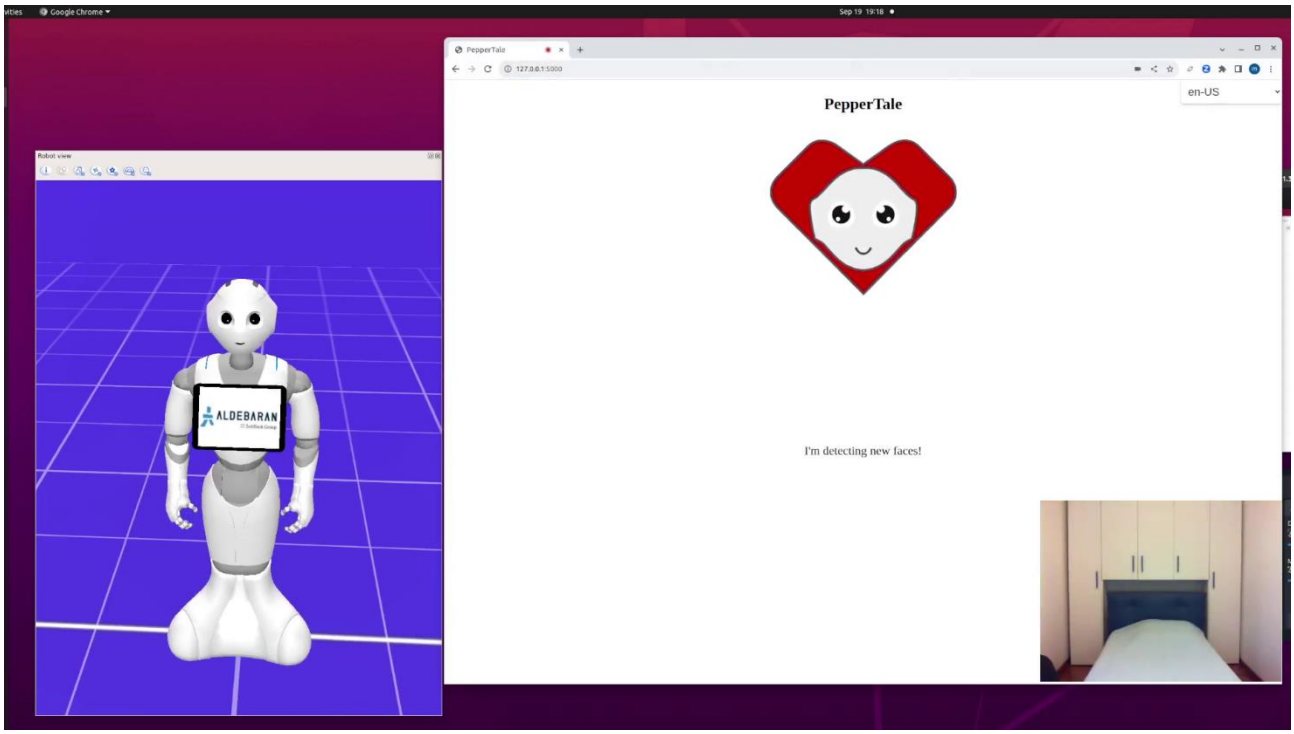
The human-robot interaction will have various forms of feedback to facilitate effective cooperation:

- Intuitive Icons: Users will receive immediate visual cues through intuitive icons, enabling them to discern whether the robot is speaking, listening, or facing difficulty in understanding their commands.
- Language Selection: To enhance user comfort, the robot will speak in the language of the user: this is done by selecting in the top-right corner of the screen the preferred language.
- Textual Representation: Everything that the robot says will be reported in a text field in the web app UI.

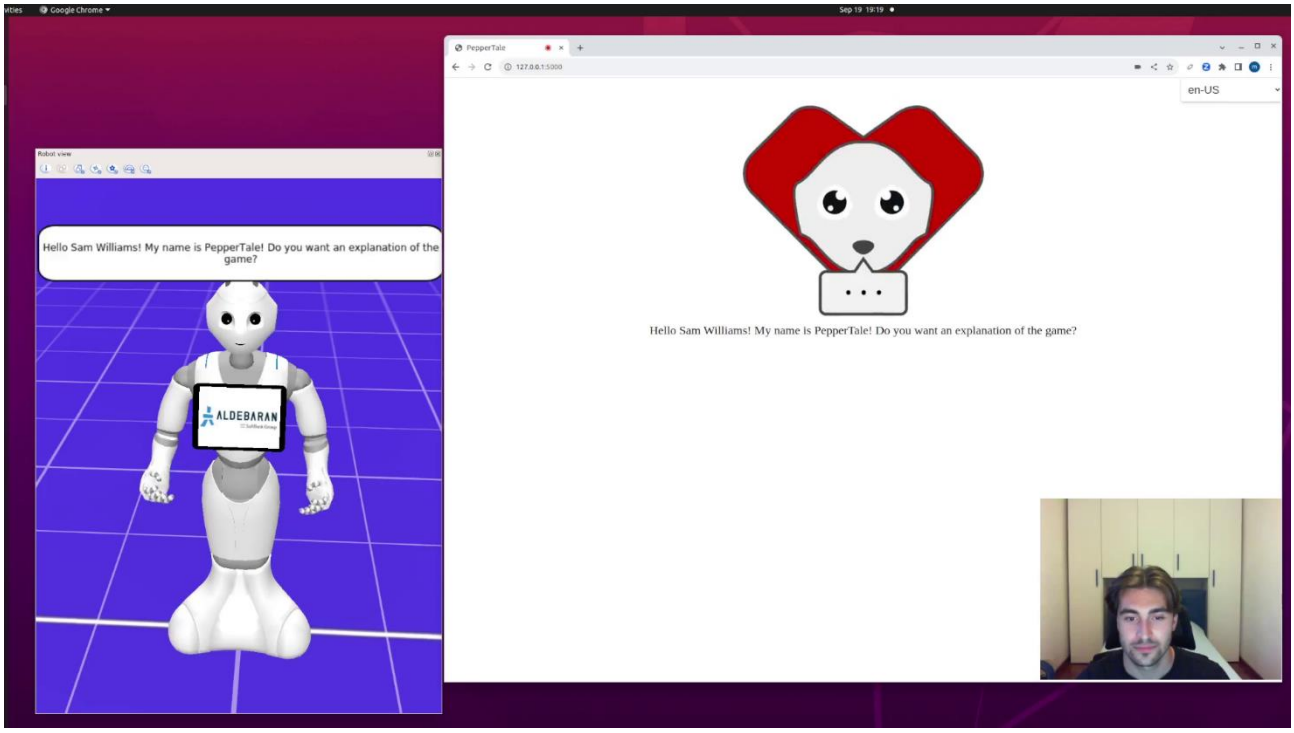
In most cases, if the robot fails to capture a user's response due to insufficient audibility or clarity, it will politely request the user to repeat their statement. Furthermore, if the robot misinterprets the user's input, the user retains the

ability to inform the robot of the correction, ensuring accurate and effective communication.

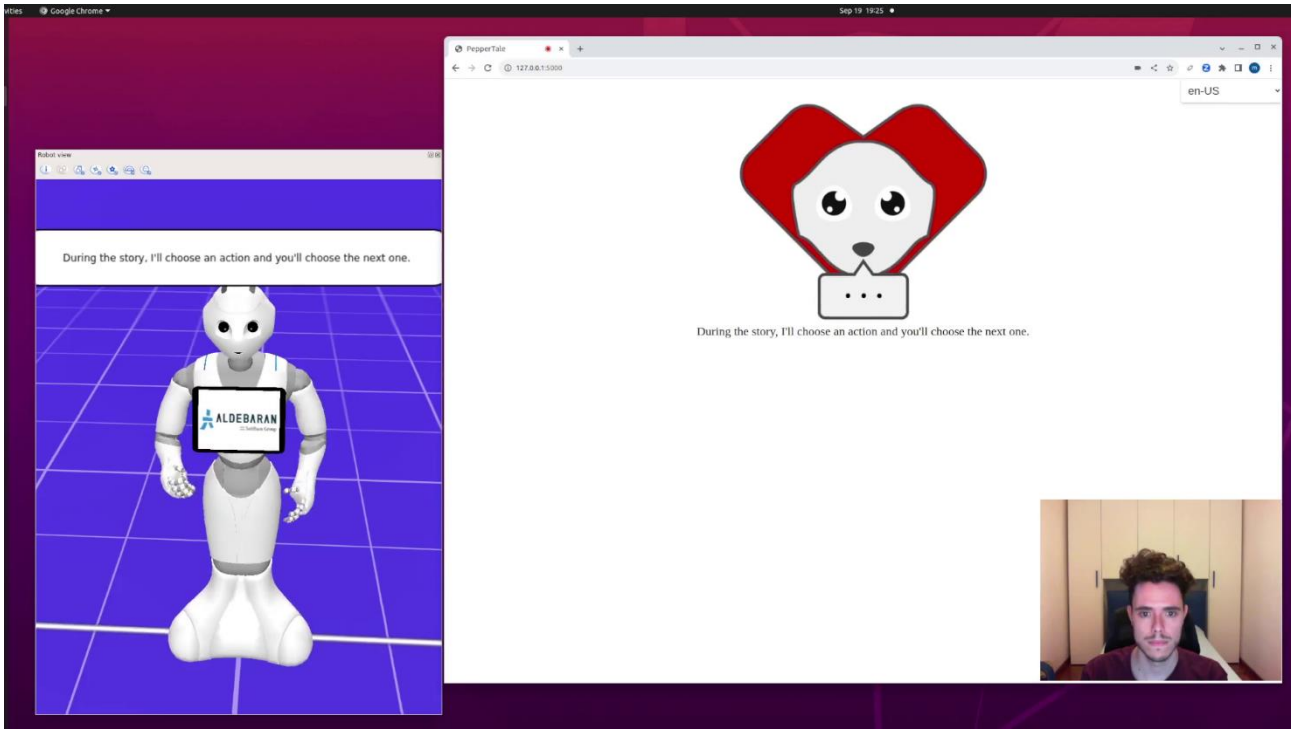
When one or more users enter the robot's field of view, various behaviors come into play. If a user is simply passing by without any intention to interact, the robot will refrain from initiating contact and will persist in searching for a potential candidate. Instead, if an user wants to communicate with the robot and makes eye contact with it for a small amount of time, the robot will check if the user is a face that it has never seen or it's someone that have interacted before: this reasoning is done by using the face recognition system. From here, different branches of interactions can arise.



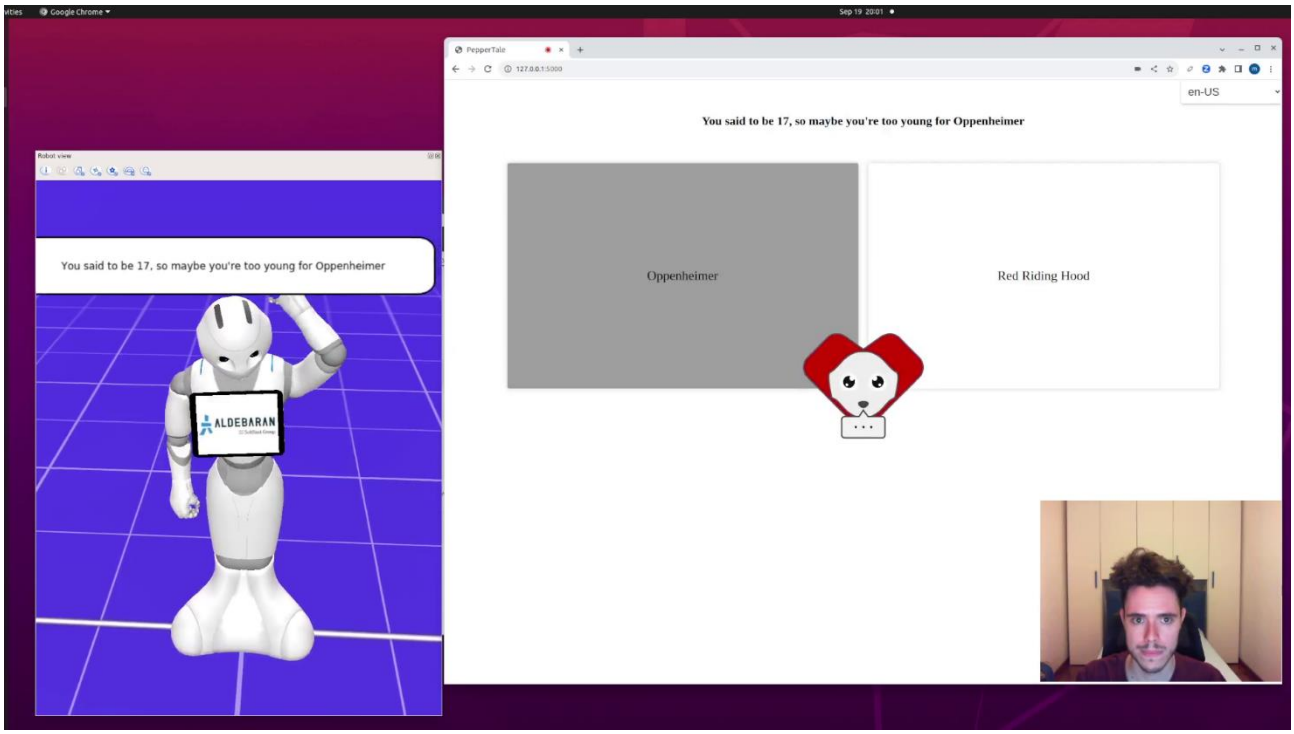
In case it finds new faces, the system will communicate with each of them in order to get acquainted with each individual. To do so, it will show the face of the person on the web page and asks the name, age and sex. After this process, these users will be treated by the robot as new players that don't know how to play the game.



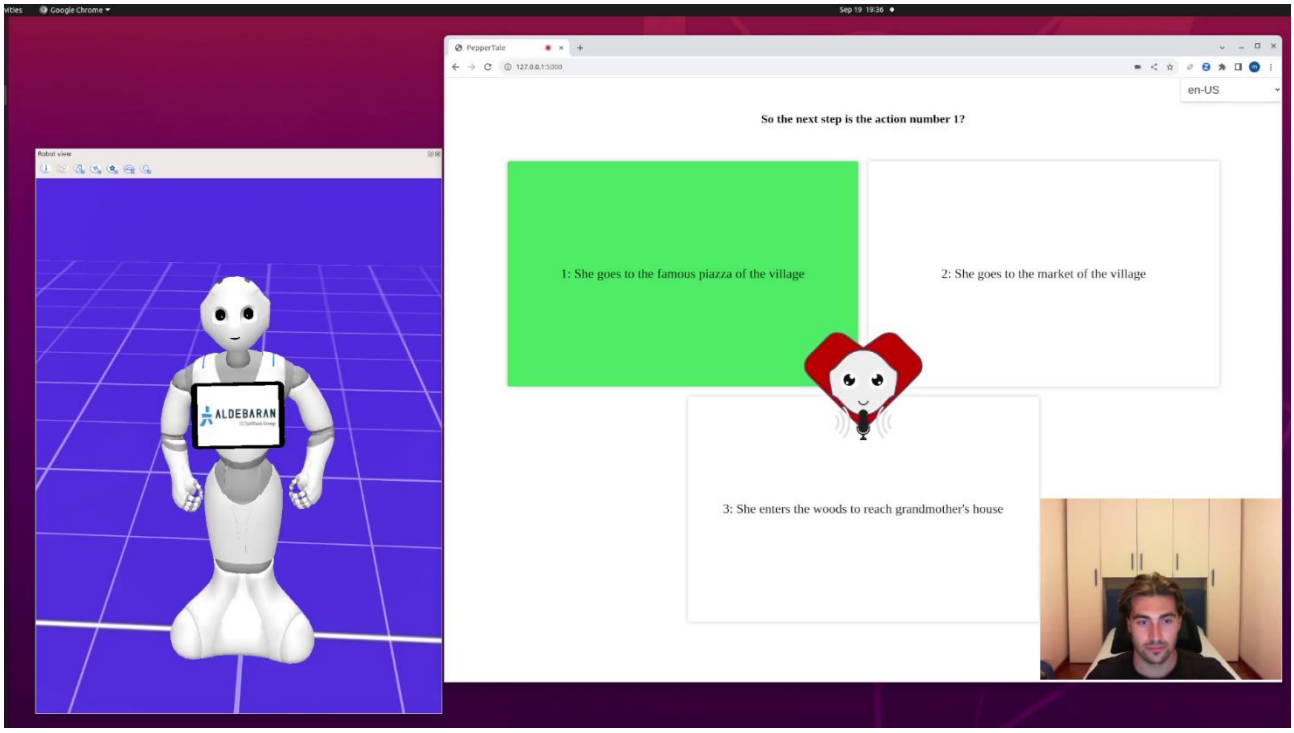
Once a user that has been memorized by the robot wants to play with it, two different types of dialogues will be executed by the robot, based on whether the player is new or returning. In both cases, the user can request to explain the game.



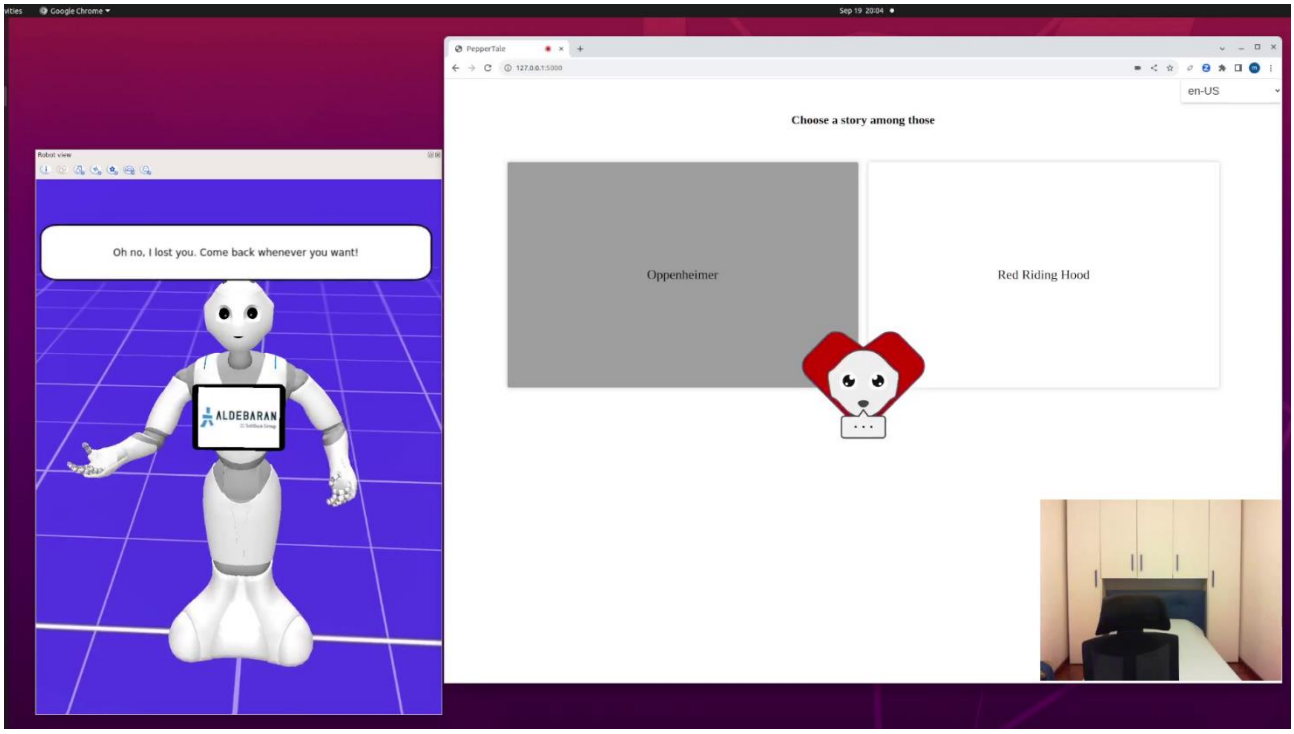
After this last phase, the user can choose which story to play from the available ones created by the developer. Additionally, the robot will provide recommendations regarding stories that may not be suitable for the user's age group. This feature will be especially valuable for new users, helping them avoid stories that might not be appropriate.



When the user chooses the story that wants to play, the game will start. In this interaction, the web app will use the story manager to compute the PPDDL plan with priorities. As the story unfolds, the robot assumes the role of guiding the player into the narrative. After a small introduction, the player will choose which of the possible actions to take and each action will have a different effect on the story, that could be a small one or a big one. In this process, the robot will also take decisions during the evolution of the story, trying to steer the tale towards its own goals.



This way of interaction will generate different stories for different routes taken by both the user and the robot and it will stimulate the user to play again the same story to discover new secrets and possible endings.
Lastly, whenever the user leaves the robot's field of view, Pepper will acknowledge it and start looking for a new user to interact with.



5 Conclusions

We learned the importance of placing the user at the center of the design process. Prioritizing user experience, we integrated intuitive feedback mechanisms, language customization, and age-appropriate content recommendations to enhance engagement and satisfaction.

The development of this project has been both enlightening and challenging. We encountered various technical complexities, from integrating multiple services to fine-tuning the robot's behaviors. However, these challenges served as opportunities for growth, allowing us to develop a robust and versatile system capable of delivering compelling interactions.

To further extend the project's functionalities and enhance its effectiveness in both HRI and RA, several approaches can be explored:

- Usage of a LLM to generate and interact with the story: Continuously improving the robot's language understanding and response generation through advanced natural language processing techniques can elevate the quality of interactions and reasoning.
- Multi-Robot Collaboration: Exploring interactions involving multiple robots can add complexity and richness to the user experience. Coordinated actions and dialogues between robots could create novel storytelling opportunities.
- Accessibility: Ensuring that the system is accessible to a wide range of users, including those with disabilities, is essential. Incorporating accessibility features and options for customization can improve inclusivity and human-robot interaction.
- Different evaluation methods: To further improve the system, possible requirements analysis, task analysis, expert and user-based evaluations can be added to the development process.

In conclusion, our project has been a remarkable journey into the realms of HRI and RA. It has provided us with valuable insights, and we believe that

continuous innovation and adaptation will drive the project to new heights. By focusing on user-centric design, personalization, and technological advancements, we can create a more engaging, effective, and inclusive interactive experience for users in the future.

Bibliography

[1] Emotional Storytelling in the Classroom: Individual versus Group Interaction between Children and Robots
https://scazlab.yale.edu/sites/default/files/files/LeiteEtAl_HRI2015_EmotionalStorytelling.pdf

[2] Affectivity in conversational storytelling: An analysis of displays of anger or indignation in complaint stories
https://benjamins.com/catalog/getpdf?webfile=a477203171

[3] Emotion Expression in HRI – When and Why
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8673078

[4] Improved Non-Deterministic Planning by Exploiting State Relevance
https://haz.ca/papers/muise-icaps2012-fond.pdf