Guardian of the Aisles: Enhancing Customer Experience in Shopping Malls with Pepper

Group 08: Andrea Vincenzo Ricciardi (2009), **Andrea Zinno** (2064), **Giovanni Rolando** (2018)

[1] Dept. of Information Eng., Electrical Eng. and Applied Mathematics - University of Salerno, Italy

Abstract This project introduces an AI-assisted robotic system, designed for shopping malls, with **Pepper** at its core acting as a guardian and interactive assistant. Pepper relies on a connected camera and video analytic system for detecting people and recognizing their attributes, such as gender, clothing color, and accessories. However, it does not process this data; instead, it accesses the information from a database. This allows Pepper to interact with visitors through spoken language, enhancing customer experience by serving as an interactive interface to the mall's sophisticated data processing system.

For correspondence:

a.ricciardi38@studenti.unisa.it (AVR); a.zinno6@studenti.unisa.it (AZ); g.rolando1@studenti.unisa.it (GR)

1 | Introduction

In the evolving landscape of retail technology, the fusion of advanced video analytic and robotic assistance has opened new frontiers in enhancing customer experience and operational efficiency. At the heart of this innovative ecosystem is **Pepper**, envisioned as a robotic sentinel for the shopping mall. Pepper is not just a passive observer; it actively interacts with customers using natural language processing. However, it's important to note that Pepper does not directly process the recognition of pedestrian attributes. Instead, Pepper accesses a database that stores information processed by the video analytics system. This system is responsible for detecting people and identifying specific attributes such as gender, clothing color (both upper and lower garments), and the presence of accessories like bags and hats. By leveraging the data acquired from the camera system database, Pepper can answer various queries related to the mall's environment. For instance, it can provide real-time information about the number of people in the mall or identify individuals based on specific attributes like clothing color or accessories.

2 | ROS Based Architecture

In the following section, we delve into the ROS architecture that underpins Pepper's effective operation. Specifically, as illustrated in the Figure 1, the ROS architecture employs its two primary communication mechanisms, *topics* and *services*, to achieve this objective. This strategic use of ROS functionalities ensures that Pepper can efficiently process and respond to the data and requests it encounters in the shopping mall environment [1].

2.1 | Image Publisher Node

The **image publisher node** is developed to manage and publish video feeds from a camera, specifically designed to interface with Pepper. This node is adaptable to both Pepper's internal camera and external camera sources, making it versatile in various operational settings. As designed, the node acts as a *publisher*. Its primary role is to publish the images captured by the camera on the `/image_feed` topic.

[1] The complete code is available at the following link.

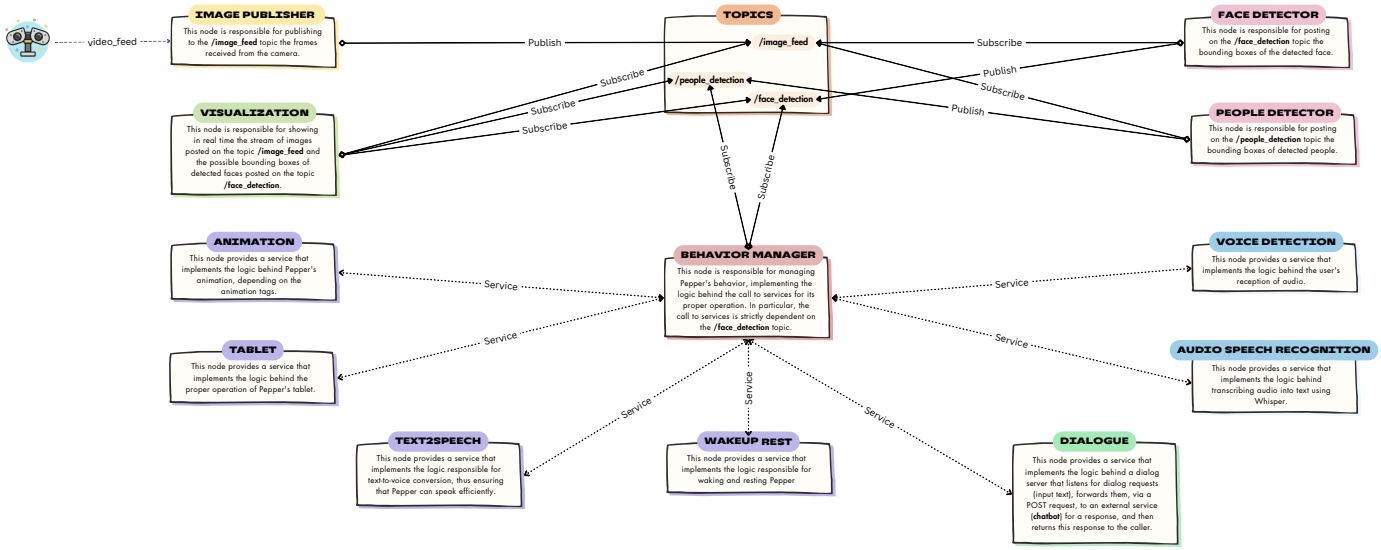


Figure 1. Design of the ROS-based architecture behind Pepper's operation as a the robotic guardian of the shopping mall.

2.2 | Face Detector Node

The **face detector node** serves as both a *publisher* and a *subscriber*. It is designed to detect faces in the image stream that is published on the `/image_feed` topic. To accomplish this, the node utilizes two important files from OpenCV's deep learning module:

- The `opencv_face_detector.pbtxt` file is a configuration file that outlines the structure of the neural network used for face detection. It details the layers and connections within the network.
- The `opencv_face_detector_uint8.pb` file, on the other hand, contains the pre-trained model with the weights of the neural network. This model is trained to identify faces within images.

As the node processes each image, it applies this neural network model to scan and detect faces. If a face is detected in an image, the node then calculates the bounding box for each detected face. Once these bounding boxes are determined, the node packages this information into a data structure known as Detection2DArray. This data structure is specifically designed to hold details of detected objects in a two-dimensional space, including their positions and sizes, which in this case are the positions and sizes of the bounding boxes around the faces. Finally, the node publishes this data on the `/face_detection` topic.

2.3 | People Detector Node

The **people detector node** is similar to *face detector node*. It is designed to detect people in the image stream that is published on the `/image_feed` topic. The **MobileNetSSD** model is chosen for its efficiency and effectiveness in real-time object detection tasks, making it suitable for scenarios where quick and accurate people detection is required. It strikes a good balance between speed and accuracy, making it well-suited for robotic applications where real-time processing of camera input is essential. As the *face detector node*, once the bounding boxes are determined, the node packages this information into a data structure known as Detection2DArray and then publishes this data on the `/people_detection` topic.

2.4 | Visualization Node

The **visualization node** serves as a *subscriber* within the environment. Specifically, its primary purpose is displaying a real-time stream of images published on the `/image_feed` topic, and showing bounding boxes obtained from the data published on `/face_detection` and `/people_detection` topics. The node handles both raw image data and object detection data, which need to be synchronized for accurate display. The lock mechanism ensures that the image is not updated or displayed while it is being modified with new bounding boxes or labels.

2.5 | Voice Detection Node

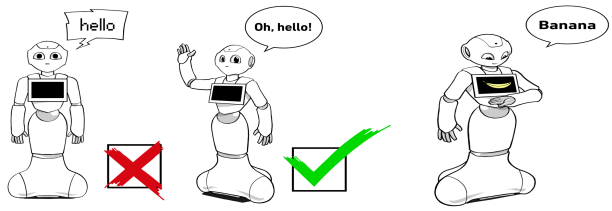
The **voice detection node** defines a ROS service - known as `voice_detection_service` - that allows othere nodes in ROS to request voice detection. When a voice detection request is received through the service, the node activates the microphone to start listening for audio input. It captures audio data for up to 2 seconds or until speech is detected. Configuring the timeout is essential to prevent potential deadlocks or scenarios where an individual who is no longer within Pepper's field of view can still communicate with Pepper. The node sends a response back to the requesting node, including the detected audio data. If no speech is detected within the 2-second timeout or if an error occurs during audio processing, appropriate response data is sent.

2.6 | Audio Speech Recognition Node

The **audio speech recognition node** is designed to perform speech recognition using the Whisper ASR Api. The node sets up a ROS service - known as `asr_service` - that receives requests for speech recognition. The decision to employ Whisper, as opposed to Google's ASR, can be attributed to the superior performance of Whisper in specific aspects. Firstly, Whisper demonstrates remarkable accuracy and precision in transcribing spoken language. Furthermore, it offers an additional feature that enhances the quality of transcriptions: sentence punctuation. In addition to transcribing spoken words, Whisper can intelligently insert appropriate punctuation marks into the transcribed text, accurately reflecting the structure and cadence of spoken sentences. However, we have observed that Whisper faces challenges in recognizing short words like *yes*, *no*, etc.

2.7 | Animation

The **animation node** is responsible for controlling Pepper's animations to achieve an empathetic behavior. Its goal is to enable Pepper to engage emotionally with the person it interacts with by selecting and executing animations based on the responses provided by the chatbot. To achieve this objective, the node sets up a ROS service - known as `animation_service` - that receives a string as input, which specifies the desired animation to be executed. The node makes use of the ALAnimationPlayer, a service provided by Pepper's software, to run the selected animations. It randomizes the choice of animations to introduce variety and naturalness into Pepper's emotional responses.



(a) (b)

Figure 2. Depiction of Pepper's ideal behaviors implemented by us.

2.8 | Tablet

The **tablet node** is responsible for handling tablet interactions and is designed to facilitate the user's search for a person based on specific attributes. To achieve this objective, the node sets up a ROS service - known as `tablet_service` - that receives as input the response processed by the chatbot, on which the tablet screen itself depends. Furthermore, we implemented a script - known as `app.py` - that employs the Flask web framework to create a web-based user interface for the tablet. The node leverages the ALTabletService provided by Pepper's software to display web-views on the tablet interface. By default, the tablet displays a video of Pepper comforting a child Figure 2a. When the user interacts with Pepper to search for a person, the node captures and processes the user's choice. These choices are saved in a JSON file, and when Pepper asks the user if they want to submit the form, the node displays the user's previous selections on the tablet screen Figure 2b. This function ensures that the user can review their choices before submitting the form.

(a) (b)

Figure 3. (a) Default Tablet Screen. (b) Finding People Form Screen.

2.9 | Text2Speech Node

The **text2speech node** is designed to convert text messages received via a service - known as `t2s_service` into audible speech. When a request is made to this service with a text message, the node utilizes the ALTextToSpeech service to have Pepper pronounce the provided text.

2.10 | WakeUp Rest Node

The **wakeup rest node** is primarily designed to control Pepper's posture and motion. This node plays a crucial role in managing Pepper's behavior by utilizing the ALMotion and ALRobotPosture services provided by Pepper's software. The key functionalities of this node include:

- **WakeUp**: When the `wakeup_service` is called, the node uses the ALMotion service to wake up Pepper, turning on its motors.
- **Rest**: The node also provides a `rest_service` that allows Pepper to enter a resting state. When called, it uses the ALMotion service to put the robot in a resting position, conserving energy.

A key component of this node is its interaction by using the ALBasicAwareness service. This service plays a crucial role in the robot's ability to track and respond to humans. When enabled, it allows the robot to track a person, enhancing its interactive capabilities.

2.11 | Dialogue Node

The **dialogue node** provides a service - known as `dialogue_server` - that implements the logic behind a dialog server. In particular, it acts as a bridge between a ROS-based system and an external chatbot service. It takes input text from the ROS service request, forwards it to the chatbot, making a POST request at the URL http://localhost:5002/webhooks/rest/webhook, receives the chatbot's response, and sends the response back to the ROS service caller. This enables the integration of natural language dialogue capabilities into a ROS-based system.

2.12 | Behavior Manager Node

The **behavior manager node** serves as the core controller for managing Pepper's behavior. Its primary function is to facilitate interactions between Pepper and users by coordinating the various services and actions, described in the previous sub-sections.

It initializes various services, including `ASRService`, `Dialogue` (for chatbot interactions), `VoiceDetectionService`, `AnimationService`, `TabletService`, and `Text2Speech` services. The decision to make services persistent makes sense in scenarios where all these services might be needed multiple times in rapid succession, such as the one described. For example, in a conversation with a user, there could be frequent ASR, chatbot interaction, and TTS requests. Making these services persistent reduces the overhead of initializing them repeatedly, improving overall system responsiveness.

The node continuously monitors the presence of faces and people using face and person detection sensors. When a user's face is detected, it initiates a friendly greeting animation and vocalizes a welcoming message. It continuously listens for the user's voice input, employing voice detection and Automatic Speech Recognition (ASR) to convert spoken words into text. This text is then sent to a chatbot service to generate a response.

Depending on the nature of the response, the node triggers different animations to match the context. For example, if the response requires an explanation, specific animation are activated. Additionally, the response text is converted into speech using Text to Speech (TTS) services and played back to the user.

The node also includes a feature that monitors the continuous presence of a user's face. If no face is detected for an extended period (more than 5 seconds), the node intervenes, interrupts the ongoing operation, and informs the user.

3 | Chatbot

At the heart of the project lies the **chatbot**, which plays a pivotal role in facilitating human-Pepper interactions, and its functionality is of paramount importance. This means it should understand natural language, respond in a friendly and relatable manner. This approach enhances the overall user experience and makes interactions with Pepper more intuitive and enjoyable. As mentioned in the previous section, Pepper incorporates a range of bodily animations, allowing for a more natural and less static conversation with the user. These animations add a layer of expressiveness to Pepper's interactions, making the robot's responses not only linguistically coherent but also visually engaging.

Based on the information gathered from the database, Pepper possesses the capability to inquire about specific aspects related to a shopping mall environment. These inquiries may encompass:

- **Count of Individuals in the Shopping Mall with Specific Attributes**: Pepper can determine the number of individuals in the shopping mall who meet certain predefined criteria. These criteria can include attributes such as gender, clothing color (both upper and lower garments), the presence of a bag, or of a hat. In addition to these attributes, Pepper also has the capacity to assess the number of passages and the duration of presence in two specific regions of interest (ROIs) monitored by the camera system. In our case study, we consider the *supermarket* and the *bar* as these ROIs.
- **Search for an Individual Meeting Specific Attributes**: In addition to counting, Pepper can also perform a search to locate a person within the shopping mall who matches specific attributes, as described above.

The implementation of the operations for counting and searching for people in the customer tracking system is facilitated by the `CustomerTrackingSystem` class.

One of the key features of this class is its ability to update the internal data state based on input entities. The `update` method plays a pivotal role in this regard. It processes a list of entities, evaluating each one to determine its relevance to gender, clothing, or ROI-related criteria. It effectively updates its fields by leveraging language-specific characteristics in English, where negations and color descriptions precede nouns. This ensures accurate updates for clothing and gender data. The logic governing the update of ROI fields, including entities such as negation, place, passages, and duration, is more intricate. This complexity arises from the need to consider the sequence of these entities and their interplay [2].

Box 1. Count People Task

This scenario illustrates the practical use of the chatbot in a `count_people` task. To begin, the chatbot initiates the conversation with a friendly greeting. Subsequently, the user engages with the chatbot by requesting information about the current number of people in the shopping mall. Upon completing the `count_people` task, the chatbot consistently follows a structured approach. It first inquires with the user whether they would like to either initiate a new search based on the data previously collected or proceed with a different task.

```
Your input → Hello there!
Hey! I'm Pepper, the robotic guardian of this shopping mall.
What can I do for you?
Your input → How many people are currently in the mall?
There are currently 15 people in the mall.
Do you want to stop or do you want make a new research based on the previous data?
Your input → Can you identify the number of customers that don't wear a white shirt?
There are currently 12 people in the mall that meet the required specifications: no white top.
Do you want to stop or do you want make a new research based on the previous data?
Your input → and how many of them are wearing a hat but not carrying a rucksack. Moreover, how many of them have w
alked past the supermarket for more than 7 seconds?
There are currently 1 people in the mall that meet the required specifications: no bag, hat, no white top, have stay
ed at least once and at least 7 seconds in front of the supermarket.
Do you want to stop or do you want make a new research based on the previous data?
Your input → You can stop.
What can I do for you?
Your input → How many of them are boys?
There are currently 8 people of male gender in the mall.
Do you want to stop or do you want make a new research based on the previous data?
```

Figure 4. Example of Count People Task

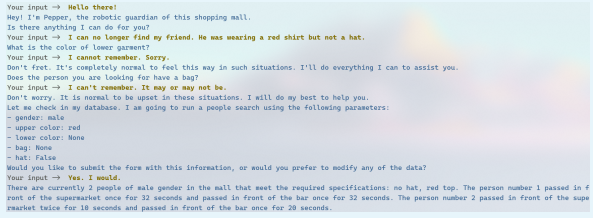
We can see that the chatbot functions correctly in this context. After recognizing the user's intent to conclude the interaction (the `stop` intent), it takes the appropriate action by resetting any slots that may have been filled during the previous requests. This behavior ensures a smooth transition between tasks and a user-friendly experience, as the chatbot is responsive to the user's cues and maintains a coherent conversation flow.

[2] Additionally, there is a constraint when performing consecutive searches on both ROI regions. To conduct such searches, **it is imperative to specify the number of passages for both ROIs**. This requirement ensures that the class can differentiate between the two ROI and perform accurate searches.

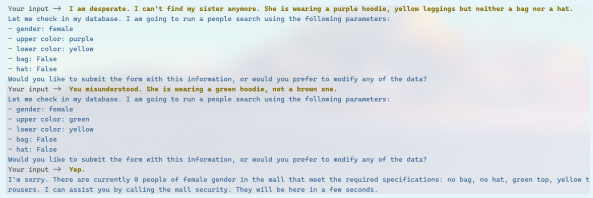
Box 2. Finding Person Task

This scenario illustrates the practical use of the chatbot in a `finding_person` task. To achieve this goal, the chatbot requests the user to specify the values of several slots corresponding to attributes that Pepper acquires from the database, namely `gender`, `upper_color`, `lower_color`, `bag`, and `hat`. To fill these slots, we employ a dynamic form that prompts the user to complete any remaining empty slots.

In the case of the Figure 5a, the user asks Pepper for assistance in finding their friend. The chatbot deduces from the pronoun 'He' that the user is searching for a male individual wearing a red shirt but not a hat. Since values for the `lower_color` and `bag` slots were not provided, the user is prompted to provide these details.



(a)



(b)

Figure 5. Examples of searching for a person by filling in the form slots in (a) multi-steps and (b) in one step.

Once all the slots are filled, the chatbot performs a filtering operation to determine the subset of individuals who match the user's specified attributes by using the methods of the `CustomerTrackingSystem` class. In cases where the search yields no results, Pepper takes the initiative to call the human security personnel at the shopping mall to provide reassurance to the user [a].

[a] Obviously, this is only a statement.

To ensure that the chatbot effectively interprets and responds to user inputs, a comprehensive and well-structured setup of *intents*, *entities*, *slots*, *lookup tables*, *synonyms*, *stories*, *rules*, as well as a carefully chosen pipeline and policies is essential.

3.1 | Domain

Each **intent** is integral to fostering an intuitive, user-focused conversational experience, adeptly catering to the diverse needs and expressions of users. Specifically:

- `affirm`: Detects affirmative responses from the user.
- `capabilities`: Helps the chatbot understand when a user is inquiring about its functionalities or the range of tasks it can perform.
- `count_people`: Activated when the user wants the chatbot to count or identify people, typically within a given context or set of parameters.
- `deny`: Opposite of `affirm`, this intent captures negative responses from the user.
- `doubt`: This intent is for situations where the user may be anxious and unable to recall certain attributes. In such cases, the chatbot always responds with reassuring phrases, assuring the user that it will do its utmost to locate the missing person (see Figure 5b).
- `finding_someone`: Triggered when the user's objective is to locate or identify a specific person.
- `goodbye`: Recognizes when the user intends to end the conversation.
- `greet`: Used to detect greetings from the user.
- `inform`: This intent is crucial for scenarios where the user provides new information or updates previously provided information.
- `nlu_fallback`: Handles situations where the chatbot is unable to understand the user's input.
- `stop`: This intent is used when the user wants to halt a current process or interaction.
- `thanks`: Used to detect user gratitude.

Entities are predefined categories of information that the chatbot can recognize and extract from user input. The entities in this file include:

- `clothing`: Used to identify mentions of clothing items in user input.
- `color`: Recognizes color references, crucial for identifying objects or clothing.
- `duration`: Captures time durations in a ROI.
- `gender`: Identifies gender references.
- `negation`: Detects negation words or phrases, important for understanding the context of responses.
- `passages`: Used for identifying the number of passages or movements in a ROI.
- `place`: Recognizes place names or locations in the shopping mall.

In this project, the **slots** are strategically utilized to serve two primary functions: facilitating the form for searching a person and enabling multiple counting operations while retaining previously set parameters.

3.2 | Training Data

For each intent defined in the `nlu.yml` file, there are corresponding examples of user utterances. To enhance the model's accuracy in identifying intents and entities, we have incorporated lookup tables into our training data. Specifically, we created a **lookup table** for `gender`, which includes nouns that directly indicate a person's gender, such as *aunt*, *boys*, etc. Another table is for `clothing`, listing various garments to help deduce whether an item is an upper garment, a lower garment, a head accessory, or a bag. The `negation` table contains negation terms, crucial for detecting negation in user utterances, such as *don't*, *not*, *no*. Additionally, we have the `color` that contains the list of color that Pepper can recognize. Lastly, the tables for `duration` and `passagges`, which respectively consist of time expressions, such as *one minute*, *14 seconds*, *two hours*, etc., and frequency expressions, such as *once*, *twice*, *twenty-five times*, etc.

Synonyms play a significant role in enhancing the entity recognition process. By using synonyms, we group a set of related terms from the lookup table `clothing`, allowing them to be associated with a specific value or category. This approach simplifies the classification and understanding of various clothing items. For example, different types of upper body garments like *t-shirt*, *shirt*, *blouse*, *hoodie*, and others can be grouped under the synonym `top`. The same principle applies to the `gender` lookup table. Here, synonyms are used to associate each title or term referring to a person with the appropriate gender. For instance, terms like *women*, *girls*, *mum*, etc., are grouped under the synonym `female`.

To train the dialogue management model in our Rasa project, we employ a strategic blend of rules and stories. **Rules** are implemented to govern specific segments of conversations that are expected to follow a consistent and predefined path. An illustrative example of such a rule is depicted in Listing 1.

Deactivate the form if intent is `count_people`

```
rules:
    - rule: Deactivate the form if intent is 'count_people'
    condition:
        - active_loop: find_person_form
    steps:
        - action: find_person_form
        - intent: count_people
        - action: action_reset
        - action: action_deactivate_loop
        - active_loop: null
        - slot_was_set:
            - requested_slot: null
        - action: action_count_people
        - action: utter_ask_stopping
```

Listing 1. Example of rule that defines a specific sequence of actions when the intent `count_people` is detected during an active `find_person_form` loop.

On the other hand, **stories** are essentially examples of real conversations that guide the model in learning how to handle different conversation flows, including branching paths and varying user intents. To avoid the common challenge of omitting important events in stories, we leverage interactive learning. This approach, initiated via the `rasa␣interactive` command, allows us to directly engage with the bot, simulating real conversations and providing immediate feedback to refine the story writing process. An example of such a story is visually represented in Figure 6.

3.3 | Pipeline and Policies

The **pipeline** defines a series of processing components for interpreting user input. This includes the `WhitespaceTokenizer` for tokenizing sentences, `RegexFeaturizer` and `LexicalSyntacticFeaturizer` for extracting features, and `CountVectorsFeaturizer` for vectorizing tokens. The `DIETClassifier` is used for intent classification and entity extraction, and it's configured with a specific number of training epochs and a similarity constraint. The `EntitySynonymMapper` maps different expressions to the same entity, while the `ResponseSelector` is used for selecting responses for specific intents.

The **core policies** define how the chatbot decides the next step in a conversation. The policies include a `MemoizationPolicy` for recalling specific conversation sequences, a `RulePolicy` for handling predefined conversation rules, and a `TEDPolicy` for predicting the next best action based on the conversation history.

config.yml

```yaml
pipeline:
    - name: WhitespaceTokenizer
    - name: RegexFeaturizer
    - name: LexicalSyntacticFeaturizer
    - name: CountVectorsFeaturizer
    - name: CountVectorsFeaturizer
      analyzer: char_wb
      min_ngram: 1
      max_ngram: 4
    - name: DIETClassifier
      epochs: 100
      constrain_similarities: true
    - name: EntitySynonymMapper
    - name: ResponseSelector
      epochs: 100
      constrain_similarities: true
    - name: FallbackClassifier
      threshold: 0.3
      ambiguity_threshold: 0.1

policies:
    - name: RulePolicy
    - name: MemoizationPolicy
    - name: TEDPolicy
      max_history: 6
      epochs: 100
      constrain_similarities: true
```

Listing 2. Pipeline and Policies used.

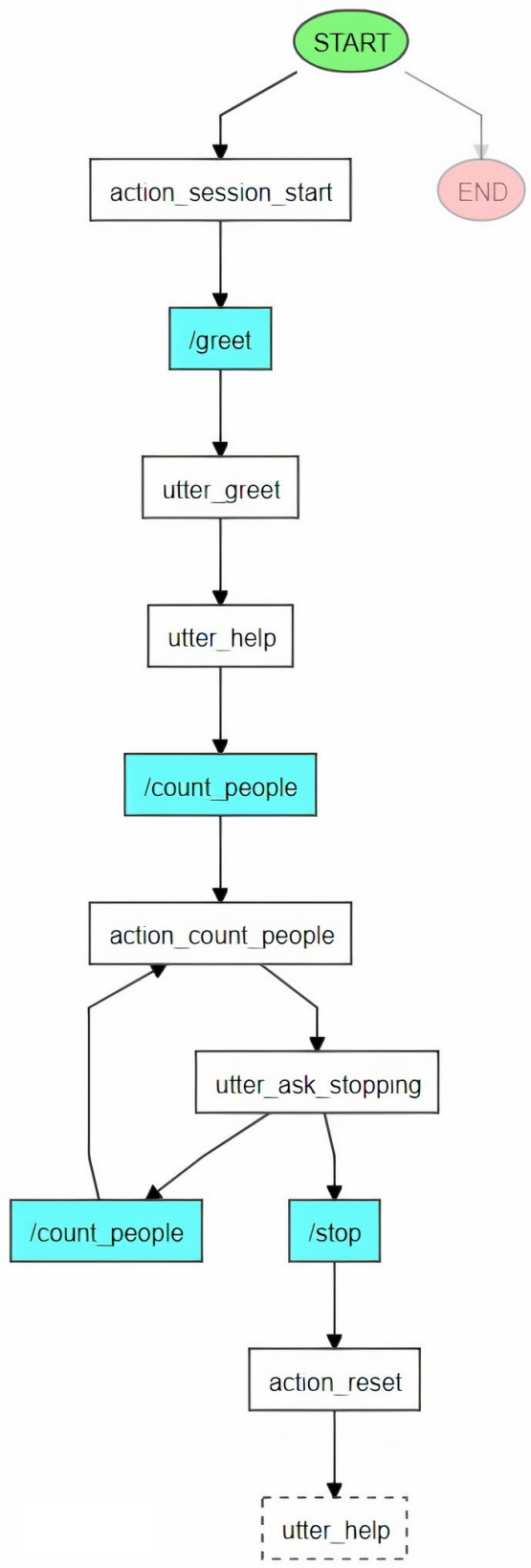


Figure 6. Example of Rasa conversation for intent `count_people`.

(a) (b)

Figure 7. Example of People and Face Detection

4 | Detectors

To initiate a conversation with Pepper, a facial recognition-enabled robot, it is imperative that Pepper first detects a person and their face. This is a fundamental requirement for starting any interaction. Additionally, the system has been programmed with a specific behavior: if Pepper does not detect a face for more than five seconds, the ongoing conversation will automatically reset. This means that the robot's interaction protocol is designed to discontinue the current conversation if it loses track of the human face it was interacting with, and this reset happens after a five-second period of not detecting any face.

4.1 | People Detection

We utilized `MobileNetSSD` for people detection because we found it to be superior to `EfficientDetB0` in terms of inference performance. It is a deep learning model designed for object detection tasks, particularly optimized for performance on mobile and resource-constrained environments:

- **MobileNet**, the backbone of the model, is structured using depth-wise separable convolutions. This design significantly reduces the number of parameters and computational cost compared to traditional convolutional neural networks, making it efficient and fast, especially for real-time applications.
- The **SSD** part of the model allows for detecting multiple objects within the image in a single pass, providing both speed and accuracy.

4.2 | Face Detection

For face detection in your application, we are using the **OpenCV Face Detector model**, which is an efficient and effective choice for such tasks. This model is built on deep learning techniques, utilizing a convolutional neural network (CNN) specifically tailored for detecting faces. In practice, implementing the OpenCV Face Detector involves loading the model using specific files, typically a combination of a `.prototxt` file for the model architecture and a `.caffemodel` file containing the trained weights. After loading the model, it processes input images or video frames, detecting faces by identifying and outlining them with bounding boxes. This process is made efficient and adaptable to different image resolutions and conditions.

5 | Test

This section provides a comprehensive evaluation of the project from both quantitative and qualitative perspectives.

5.1 | Qualitative Test

Qualitative tests, on the other hand, focus on the quality and usability of the project. They aim to assess how well the system performs in terms of user experience, natural language understanding, and overall satisfaction.

5.1.1 | Detectors Qualitative Test

This test allows displaying real-time images captured by Pepper's camera (when `pepper_camera_on="True"`) or by the computer's camera (when `pepper_camera_on="False"`). It also shows any detected bounding boxes around faces and people. To perform this test, you need to launch the `detectors.xml` file as demonstrated in Listing 3.

Detectors Qualitative Test

```
# Pepper ON
roslaunch features detectors.xml
# Pepper OFF
roslaunch features detectors.xml pepper_on:=False pepper_camera_on:=False
```

Listing 3. This script starts nodes for people and face detection, and by default, it can control a Pepper robot by waking it up or putting it to rest and publishing images from its camera.

5.1.2 | Animation Qualitative Test

This test allows Pepper to perform a specific animation. To do this, you need to provide Pepper with one of the animations listed at the link. To execute this test, you should launch the `animate.xml` file as demonstrated in Listing 4.

Animation Qualitative Test

```
# 1° Terminal
roslaunch features animation.xml
# 2° Terminal
input="animations/Stand/Gestures/Hey_1"
rosservice call /animation_service "input: {data: '$input'}"
```

Listing 4. This script requests Pepper to perform the *Hey_1* gesture.

5.1.3 | Tablet Qualitative Test

This test allows changing the screen displayed on Pepper's tablet. To perform this test, you need to launch the `tablet.xml` file as demonstrated in Listing 5. To display the form screen, you need to pass a string to the `tablet_service` in the format shown in Listing 5. Modify these parameters to achieve a change in the screen content. The default screen is obtained by passing any other string.

Tablet Qualitative Test

```
# 1° Terminal
roslaunch features tablet.xml
# 2° Terminal
input="Let me check in my database. I am going to run a people search
↪  using the following parameters: - gender: female - upper color: yellow
↪  - lower color: red - bag: None - hat: False"
rosservice call /tablet_service "input: {data: '$input'}"
```

Listing 5. This script sets the tablet screen to the form screen. Specifically, the tablet will display the female gender symbol, the jersey symbol with a yellow background, the trouser symbol with a red background, the bag in gray scale and opaque, and the hat with a red x in the overlay, indicating that we are looking for a female person, who is wearing a yellow top, a red trouser, who does not have a hat, and who may or may not carry a bag. The Figure 3b describes this situation.

5.1.4 | Dialogue Qualitative Test

This test allows testing the Rasa chatbot through ROS. To execute this test, you should launch the `dialogue.xml` file as demonstrated in Listing 6. This launch file will create two terminals: one for the *dialogue_server*, *rasa_actions*, and *rasa_server*, and another for the *dialogue_interface*, facilitating the conversation between the user and the chatbot.

Dialogue Qualitative Test

```
roslaunch chatbot dialogue.xml
```

Listing 6. This script starts the chabot interface in ROS.

5.1.5 | S2T Qualitative Test

This test allows testing the Speech-to-Text (S2T) functionality using either the PC's microphone (`mic_index="None"`) or Pepper's microphone. To find the microphone index for Pepper, you can run the `sounddevice.query_devices()` code. We have implemented voice detection and ASR modules as services so that they can be called only when a person is present. To verify the correct operation of these two S2T modules, we have defined the `audio_test.py` class. To execute this test, you should launch the **`s2t.xml`** file as shown in Listing 7.

S2T Qualitative Test

```
roslaunch features s2t.xml
```

Listing 7. This script transcripts the user's audio into text.

5.1.6 | T2S Qualitative Test

This test allows testing the Text-to-Speech (T2S) functionality in Pepper. To do this, you need to provide Pepper with a text to be played. To execute this test, you should launch the **`t2s.xml`** file as shown in Listing 8.

T2S Qualitative Test

```
# 1° Terminal
roslaunch features t2s.xml
# 2° Terminal
input="Hello there!"
rosservice call /t2s_service "input: {data: '$input'}"
```

Listing 8. This script requests Pepper to play the text: *Hello there!*.

5.2 | Quantitative Test

Quantitative tests involve objective measurements and data-driven assessments of the project's performance.

5.2.1 | Chatbot Quantitative Test

Rasa lets validate and test dialogues end-to-end by running through *test stories*. The use of test stories is the best way to have confidence in how our assistant will act in certain situations. For our project, we have developed a suite of 15 test cases that encompass a range of functionalities, from basic greeting forms to more complex tasks such as counting operations, form filling, and recognition of fallback intents. Some of these test cases are integrated with each other. An example of test is shown in Listing 9. All of these tests have been successfully passed.

Example of Test Story

```
- story: 7 - A test story with unexpected input during a form
  steps:
  - user: |
    'I can no longer find my cousin. [She]{"entity": "gender", "value":
    ↪  "female"} is wearing a [blue](color) [sweater]{"entity":
    ↪  "clothing", "value": "top"} but [neither](negation) a
    ↪  [hat](clothing) [nor](negation) a [purse]{"entity": "clothing",
    ↪  "value": "bag"}.'
    intent: finding_someone
  - action: action_slot_mapping
  - action: find_person_form
  - active_loop: find_person_form
  - user: |
    'How many people have walked past the [supermarket](place) for [eight
    ↪  seconds](duration) but [not](negation) for [sixty-one
    ↪  times](passages)?'
    intent: count_people
  - action: action_reset
  - action: action_deactivate_loop
  - active_loop: null
  - action: action_count_people
  - action: utter_ask_stopping
  - user: |
    'Stop!'
    intent: stop
  - action: action_reset
  - action: utter_help
```

Listing 9. Example of test story where a intent `count_people` is detected when filling out the form.

Once the assistant is deployed in the real world, it will be processing messages that it hasn't seen in the training data. To simulate this, we should always set aside some part of your data for testing. A solution is use **cross-validation**, which automatically creates multiple train/test splits and averages the results of evaluations on each train/test split, that is all data is evaluated during cross-validation.

NLU testing in cross-validation mode

```
rasa test nlu --nlu data/nlu --cross-validation
```

Listing 10. Command to run NLU testing in cross-validation mode.

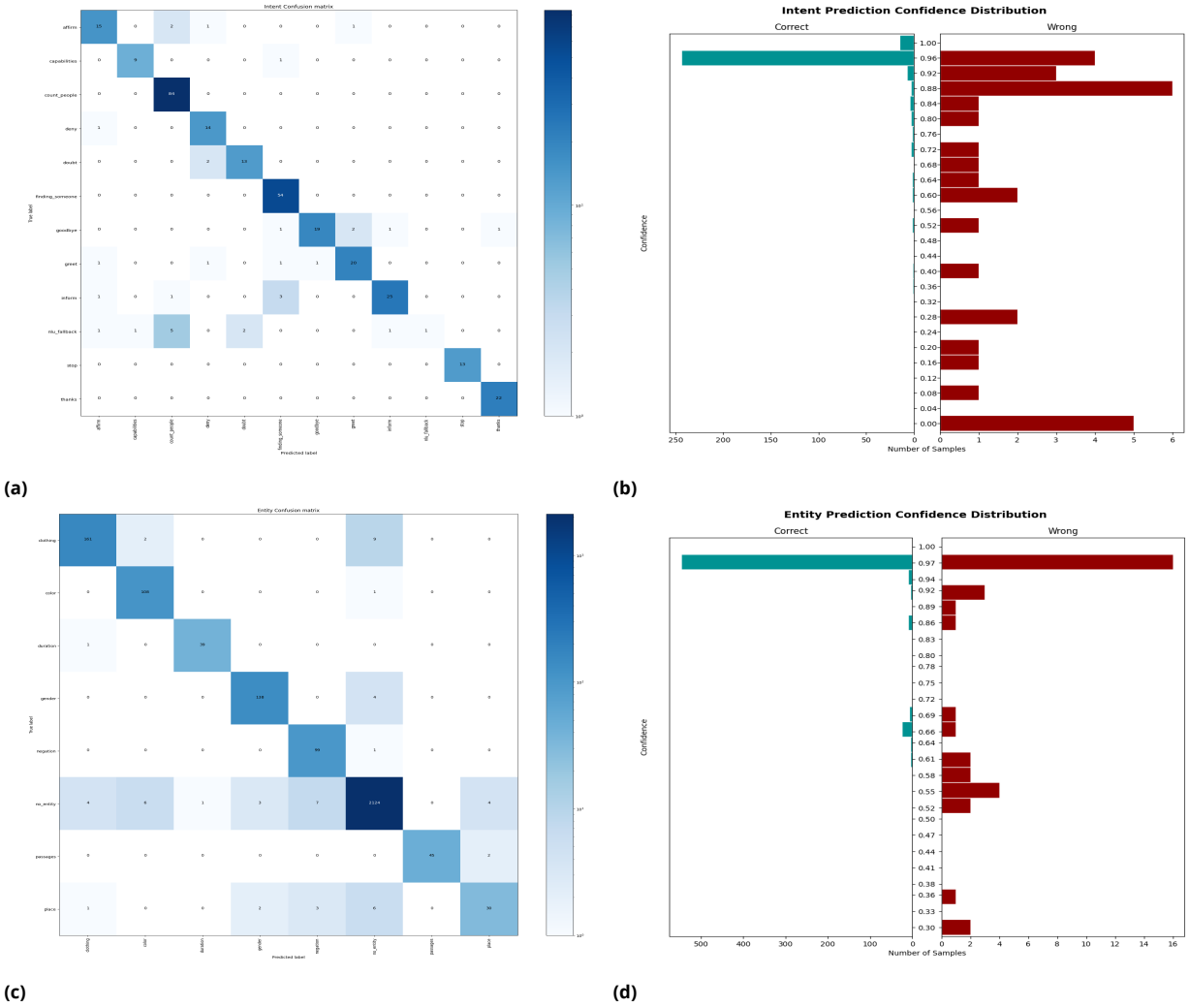


Figure 8. Confidence histograms and confusion matrices for (a) intent classification model and (b) entity classification model.

The `rasa test` script produces a report, a confusion matrix and confidence histogram for the intent classification model and also for the entity (DIET) classification model.

	Accuracy	F1-score	Precision
Train (Intent)	0.96 (0.002)	0.95 (0.001)	0.94 (0.003)
Test (Intent)	0.90 (0.021)	0.88 (0.022)	0.89 (0.025)
Train (DIET)	1.00 (0.000)	1.00 (0.000)	1.00 (0.000)
Test (DIET)	0.98 (0.008)	0.94 (0.018)	0.95 (0.025)

Table 1. Intent and Entity evaluation results with cross-validation.

5.2.2 | Detectors Quantitative Test

This test was designed to evaluate the effectiveness of detectors in identifying people and their faces. To this end, two classes, `people_detector_test.py` and `face_detector_test.py`, have been implemented. These classes process a series of frames from two videos, executing detection on each individual frame. The videos, recorded by the computer's camera, depict two distinct scenarios:

1. The first scenario, labeled as `TestVideo0`, illustrates a situation where a person enters the scene after a certain period of time and then exits the scene.
2. The second scenario, labeled as `TestVideo1`, shows a case where a person is already present in the scene and then leaves shortly thereafter.

From these videos, we can derive the following metrics, as detailed in Table 2:

	Recall	F1-score	Precision
People Detector	0.81347	0.89458	0.99367
Face Detector	0.86315	0.76102	0.6804

Table 2. Quantitative measures of the detectors' performance.

5.3 | Final Test

This test allows testing the functionality of all implemented modules and, consequently, the overall behavior of Pepper, which is implemented by the `behavioral_manager.py` class. To execute this test, you should launch the `all.xml` file as demonstrated in Listing 11. It is also possible to test the complete behavior without Pepper. In this case, Pepper modules - namely `wakeup_rest_node.py`, `tablet_node.py`, `text2speech_node.py`, and `animation_node.py` - will not be executed as `pepper_on=False`. Specifically, the person and face detectors are executed. If a person is detected by the PC camera, the voice detection service is enabled, and its output is passed to the ASR module. The transcribed text is then passed to the chatbot, which generates a response.

Behavioral Test

```
# Pepper On
roslaunch features all.xml
# Pepper Off
roslaunch features all.xml pepper_on:=False pepper_camera_on:=False
↪  mic_index:=None
```

Listing 11. This script runs all the defined nodes, allowing you to test the complete behavior of Pepper. It is also possible to test the complete behavior without Pepper.

In particular, the test, of which you can see an example application in the video at the following link, consists of the following phases:

1. **New People enters the scene**: Initially, there is no one in front of Pepper. When a person enters the scene, and Pepper successfully detects the person and their face. Then, Pepper greets the person with a welcoming gesture and says, "Hello there!".
2. **Greetings**: The person greets Pepper, and Pepper responds with another greeting gesture while uttering a welcoming phrase. In the greeting, Pepper introduces itself as the robotic guardian of the shopping mall and asks how it can assist the person. Subsequently, the person inquires about Pepper's capabilities, and Pepper responds by explaining that it can conduct people searches based on specific attributes.
3. **Count People Task**: In the case of the video in question, the following types of questions are asked for the intent `count_people`, all with positive results [3]:
   (a) **Count People without Stop**: The first case concerns the sequential execution of several count questions, adding or modifying previously set attributes.
   (b) **Count People with Stop**: The second case concerns the execution of two count questions interspersed with a `stop` intent, which causes the slots set in the previous question to be reset.
   (c) **Fully Count People Sentence**: The third case concerns the execution of one count request covering all slot types.
4. **Find Person Task**: In the case of the video in question, the following types of questions are asked for the intent `find_someone`, all with positive results:
   (a) **Find Person by filling the form dynamically**: The first case concerns filling the form dynamically, where Pepper asks the user to fill in the empty slots in order to facilitate the search for the person.
   (b) **Find Person by filling in the form in one go**: The second case concerns the case where the user provides Pepper with all the attributes it needs, filling in all the necessary slots.

[3] During a video analysis, it was noticed that the chatbot incorrectly identified the word `'by'` as a negation entity. This error was promptly corrected. However, due to the unavailability of Pepper, it was not possible to re-shoot the video sequence.

(c) **Find Person with changing attributes**: The third case concerns the case where the user changes a previously imposed attribute before submitting the form.

(d) **Find Person with stop intent**: The fourth case concerns the case where the user finds the person they are looking for (e.g. the person contacts the user by phone). In this case, all slots are reset. The same procedure occurs in the case where the intent is `goodbye`.

(e) **Find Person with `count_people` intent**: The fifth case concerns the case where the user requests a count operation while the form is active. In this case, all slots are reset and the count operation is performed [4].

5. **Goodbye**: The person says goodbye to Pepper, who promptly bids him farewell with a farewell gesture and message.

[4] This test is not in the video due to timing problems with Pepper.