King's College London
Department of Informatics
MSci Computer Science

6CCS3PRJ Project Report

Robot Receptionist

SUPERVISOR
Dr. Matteo Leonetti

CANDIDATE
Pawel Makles
K21002534

Academic Year 2023/2024

Declaration of Originality

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection, and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Pawel Makles

April 12, 2024

Word Count: 13763

Acknowledgments

I would like to express my gratitude to Dr. Matteo Leonetti for the continued help and support throughout the project. I would also like to extend my thanks to Jared Swift, Matt Barker, and Benteng Ma for assisting in testing my project in the lab and for their contributions to the LASR code repository which proved to be a vital foundation for the development of this project.

Abstract

Service robotics is a rapidly growing field with many applications from hospitality to medicine creating real-world impact on real people in the service industry. This project intends to explore the application of service robots to a social setting, specifically a gathering of people where the robot acts as the receptionist, greeting new guests, and introducing guests.

This project has successfully reached its intended goal and can successfully perform the social interactions that are asked of it. However, some areas of improvement have been identified which are discussed in the evaluation.

Contents

1 Introduction

Service robotics is a developing field and an invaluable area of study, they have the potential to impact our lives through many different forms. There are many possible applications such as to help with care for the elderly in their homes, provide a safe barrier in hospitals between doctors and patients, or simply serve you food at your local restaurant. Robots can prove to be instrumental to alleviate the burden on service workers to allow them to avoid focusing on mundane tasks and let them excel at their job. Often these robots also leave a positive impression on those they interact with [1], [2] and provide a long-lasting and novel experience.

There is continual growth potential in this industry, with a yearly revenue increase of about 5% per year [3], and in fact, service robots are slowly being rolled out to real-world use by customers to varying degrees of autonomy. For example, a popular fast-food brand "Slim Chickens" has rolled out robot waiters that deliver food and interact with guests [4], [5]. Another example is the Henn na Hotel in Japan [6], which since 2015 [7] has allowed guests to check in using robots. However, these deployments seem to lack natural end-to-end interaction which can only be achieved by exclusively using vision and speech, something we aim to tackle in this project.

Research like this can help develop the field further if at least to provide software components that help grow the ecosystem or contributions that help stimulate further research and work. Components such as improved computer vision, better human-robot interaction systems, and more stable manipulation can be repurposed into completely different applications from what this project intends to implement.

RoboCup@Home is a robotics competition league in which participants compete to solve service robotics problems in the most friendly and efficient manner. Code and many research papers [8] are often published after the competition by participating teams which help further the service robotics field.

1.1 Aims & Objectives

This project will implement the Receptionist task (Appendix Section B) from the 2024 RoboCup@Home league competition [9]. In this task, the robot is tasked with greeting new guests to a party and introducing them to others. To effectively implement this problem, the robot must have natural end-to-end interaction as mentioned previously, which in effect means the people interacting with the robot can feel as if they are interacting with another person.

This task is appropriate for research at this time since we have reached a point where the technology is just about right to exploit we have all of the tools required to successfully implement a well-performing solution. There is a clear lack of deployments of service robots which do not rely on unintuitive or cumbersome HRI.

2 Background & Context

2.1 TIAGo

For this project, I will be using the PAL Robotics TIAGo "mobile-manipulator" robot [10] to implement the task, it is a generic robot platform that serves as an excellent basis for building service robotics applications. The robot features: A **LiDAR** which is a special device that uses lasers to determine the distance of objects around the robot, this is necessary for the robot to be able to determine where it is in the world and subsequently navigate through it. It has a **differential drive** base which means it can turn in place and move both forwards and backward, this means the robot can follow more natural and concise movements akin to humans.

An **RGBD camera** is used to map out the environment around the robot, while also being able to correlate this with information gathered from its 2D view. A **mechanical arm** is present which can be used for object manipulation, however, this is not necessary for this task. The **onboard computer** is used to run and operate the robot, it is effectively immutable to us and we need not modify it. Instead, we connect to it over the network and run our logic on another machine.

2.2 ROS Overview

The **Robot Operating System** is a "meta-operating system" that provides a common platform for us to interact with many supported robots. We will be using ROS Noetic Nimjemys which is the LTS release of ROS1 based on Ubuntu 20.04 LTS, which is compatible with our robot TIAGo.

A **node** is a "process that performs a computation" [11], it represents an independent program that runs somewhere on the robot's network that can provide or consume information, but crucially, nodes do not need to care about how data is sent to/from them as this is all handled by the respective ROS client libraries for each language. In our case, we will be using rospy [12] to create our nodes.

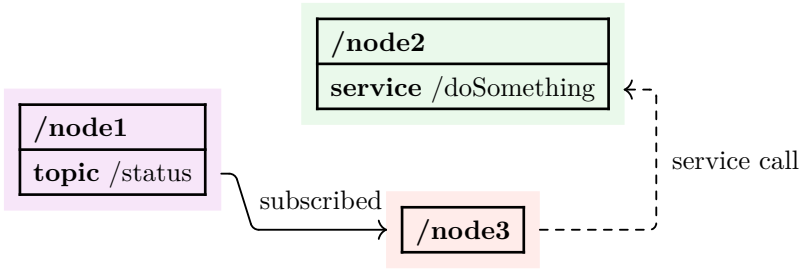


Figure 1: Nodes on the network can provide different functionality

Before we can think about sending data, we need to look at message types. ROS implements a common data format for efficient transportation across the network, we define `.msg` files in our project which define key-value entries. The value can be one of many types such as boolean, differently-sized and signed integers, differently-sized floats, strings, and time/duration.

Data Structure 1: Example Message Definition

MyMessage.msg
- header: **std_msgs/Header**
- pose: **geometry_msgs/Pose**
- count: **int**

Disclaimer

ROS messages described throughout the paper are not entirely spec-compliant as they have been simplified for brevity. The implementation section will later discuss the challenges of applying complex data structures to ROS message types.

When it comes to this project, there are two packages which provide especially important messages, `std_msgs` and `geometry_msgs`. Specifically the message definitions for: **std_msgs/Header** [13] which includes the current time, a frame of reference, and a unique sequence ID. The **geometry_msgs/Point** [14] which is a 3D coordinate. And the **geometry_msgs/Pose** [15] which includes a point defined as before and a quaternion for rotation.

Data Structure 2: Important Message Definitions

Header
- seq: **uint32**
- stamp: **time**
- frame_id: **str**

Point
- x: **float**
- y: **float**
- z: **float**

Quaternion
- x: **float**
- y: **float**
- z: **float**
- w: **float**

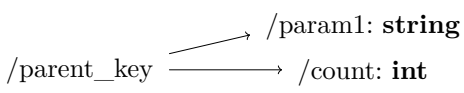
Pose
- point: **Point**
- orientation: **Quaternion**

When using ROS there are 3 primary ways to send data between nodes: A **topic** is a channel through which we can send messages over the ROS network [16], anybody may subscribe or publish to any topic they wish but the message type must be negotiated ahead of time. A **service** is defined by two message types defined in an `.srv` file (see Data Structure 3), the request and the response, a node may send a request to a node that provides a given service and will receive a response with the result [17]. Lastly, we can also use the **parameter server** to persistently store some chunks of data [18], in essence it is a dictionary that lives on the network that we can read and write from at any time, it supports a multitude of data types and we can use this to store configuration which lives outside of our program & persists between invocations.

Data Structure 3: Example Service Definition

MyService.srv
- arg1: **string** - arg2: **int**
- result: **string**

Data Structure 4: Example Parameter Server Data



Sometimes, we want to be able to cancel or monitor the status of a given task, we could build this infrastructure ourselves or use an existing well-integrated framework called **actionlib** [19]. This library provides a way to create action servers that accept given requests, regularly give feedback on the request, and allow requests to be preempted.

Data Structure 5: Example Action Definition

MyAction.action
- arg1: **string** - arg2: **int**
- progress: **float**
- result: **string**

tf2 is a library providing that lets us "keep track of coordinate frames over time" [20] by keeping a buffer of changes in relationships between different coordinate frames in memory. We can use it to transform points between different coordinate frames such as the robot's camera, the robot's base, or the map. This allows us to, for example, find out where an object relative to the robot's camera is positioned absolutely within the map. tf can also perform transformations at given points in time assuming that information is still in the buffer.

SMACH [21] is a powerful and extensible library for building state machines, it has built-in support for data passing between states and preempting running states, and also provides us with many useful concepts out of the box such as concurrency and iteration. It is also well integrated into the ROS ecosystem through the use of the **smach_ros** library [22], which gives us additional helper classes that wrap around ROS concepts such as services and actions.

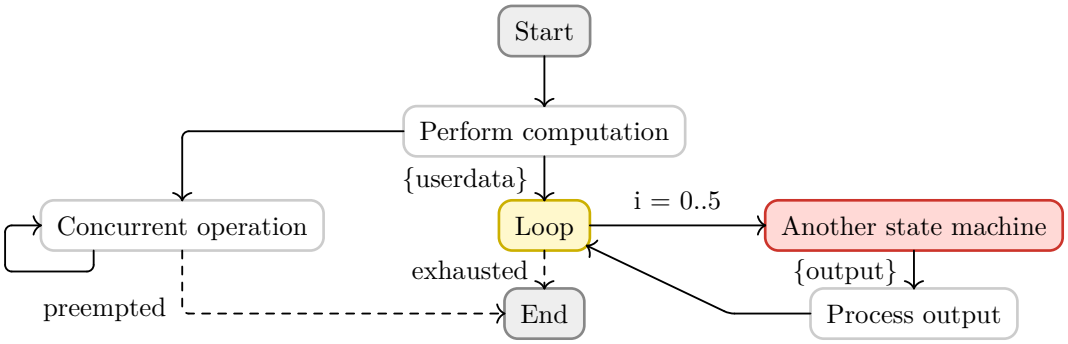


Figure 2: Example state machine

We also want a way to visualise data coming from the robot, for this we use rviz [23] which provides us with the ability to see depth and map information in 3D space as well as view individual topics and simultaneously other 2D information such as the camera feed.

Figure 3: RViz

For testing we also need to be able to simulate a robot in software, for this we use Gazebo [24]. The TIAGo software package already provides us with the necessary resources required to use Gazebo to simulate the TIAGo robot.

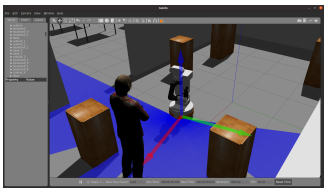


Figure 4: Gazebo

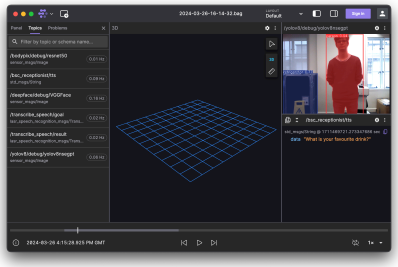


Figure 5: Foxglove

Sometimes we may want to record and look back on previous experiments or invocations of our code, or otherwise, we might find it tricky or time-consuming to debug issues with our code on either the real robot or even in simulation. For this, we can record messages sent to topics into special files called **bags** using the *rosbag* utility [25]. We can then play these back over a ROS network to view and capture the data through things such as rviz. However, handling rosbags in this way can also prove to be tedious and time-consuming, so instead, we can use a tool called Foxglove [26] which allows us to load individual bags and inspect all the data using a timeline view.

One last note on ROS: In order to simplify starting and managing different configurations for lots of nodes, we can use roslaunch files [27] which are XML files that define the nodes and arguments that we want to load. Arguments to nodes can be passed via command line arguments or through the parameter server using private parameters, once defined they may be accessed through a special parameter namespace ~, e.g. the parameter `hello` is mapped to `~hello`.

2.2.1 TIAGo Actions

TIAGo provides several useful action servers that will be used throughout the project. The important ones to remember are the navigation stack `/move_base`, a playback server for predefined motions `/play_motion`, and a text-to-speech server that uses the onboard speakers `/tts`.

Data Structure 6: Key Action Definitions for TIAGo

MoveBaseGoal.msg
- header: **std/Header**
- pose: **geom/Pose**

PlayMotionGoal.msg
- motion_name: **str**
- skip_planning: **bool**

TtsGoal.msg
- rawtext.text: **str**
- rawtext.lang_id = “**en_GB**”

The play motion server will read motion information from the parameter server using the interpolated key `/play_motion/{motion_name}`, so we can load our motions into parameters to later play them back. In the case of all of these actions, they will always run to completion before succeeding so there is no risk of, for example, starting to listen for speech while the robot is still saying something.

2.3 Prior Art

This project dips into many adjacent fields such as computer vision and speech recognition, this section intends to introduce the many tools and prior work done by people that will be discussed later.

YOLO (You-Only-Look-Once) is an ongoing research project spanning dozens of individuals into one-shot computer object recognition. Currently the latest stable software release is YOLOv8 by ultralytics [28], with a new YOLOv9 paper [29] on the horizon and being implemented however it is not stable enough for use yet.

PyTorch is a deep machine learning framework that we can use to train custom computer vision models to better suit our needs. **DeepFace** is a Python framework [30] that provides us access to state-of-the-art facial feature extraction models that allow us to perform facial reidentification. **BodyPix** is machine learning model that allows us to segment body parts from pictures of people [31], [32]. **Weaviate** [33] is an open-source vector database.

Whisper is a speech recognition model built by OpenAI [34] that offers outstanding performance over many different languages. **RASA** is a complementary framework for creating and training models to recognise intent in dialog [35].

LASR (Learning Autonomous Service Robots) is a team of students from King's College London "competing in service robotics challenges at major competitions" [36]. We publish reusable software components that will also be integrated into this project, in particular, **lasr-skills** [37] which offers many commonly-used states for SMACH, intended for use with TIAGo.

3 Literature

This section goes over various resources I've looked into that are relevant to implementing the task. The primary metric for the usefulness of any frameworks, ML models, and software chosen here is that they must be able to run offline and perform in real-time.

3.1 One-shot person and object recognition

YOLO (You Only Look Once) is a "widely used algorithm" for one-shot object detection [38]. It was first unveiled in 2015 and presented a new approach to object detection. The 'object detection' problem is reframed as a 'regression problem' where "a single neural network predicts bounding boxes and class probabilities directly from full images" [39].

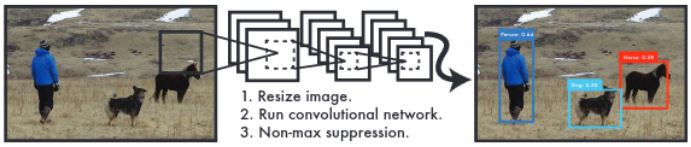


Figure 6: Core concept of the YOLO algorithm (from [39])

Over the years, there have been many successor algorithms including YOLOv2-5 [38], each attempting to optimise performance by simplifying the network or extending it to improve accuracy. More recent work such as YOLOv6 [40] and YOLOv7 [41] achieve real-time performance with reasonable accuracy.

Building on all of this prior work, YOLOv8 [28] is the current state-of-the-art model that provides image classification, object detection, object segmentation, object tracking, and human pose inference. The project claims much lower latency, with greater precision, using much smaller parameter models. To understand their evaluation, we need to look at the metrics used.

The IoU measures overlap between two given boundaries (usually the real object boundary and our predicted boundary). It is given by:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of intersection}} \tag{1}$$

So the IoU threshold is the minimum IoU between two given boundaries for a bounding box or mask to be considered a true positive, otherwise it is discarded (false negative).

Given N classes and AP_i is the average precision of a class i [42], the Mean Average Precision is given by:

$$\text{mAP} = \frac{1}{N} \sum_{\{i=1\}}^{n} \text{AP}_i \tag{2}$$

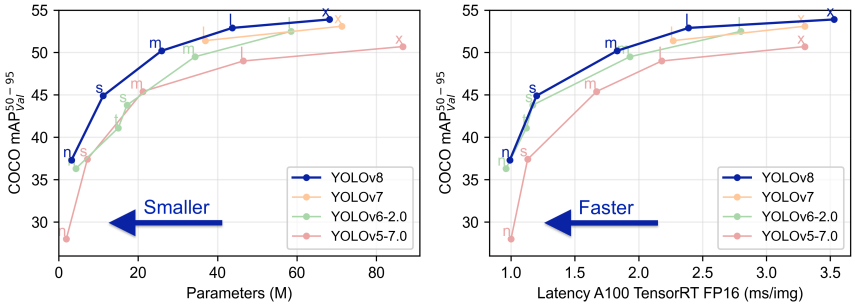


Figure 7: Comparison of YOLO models (published by Ultralytics [43])

These graphs show the Mean Average Precision (averaged over IoU thresholds of .50 and .95) of each model against the parameters used and latency for inference. YOLOv8 dominates all other models at all parameter/latency tiers.

This project requires us to detect and segment chairs and people, so it is appropriate to use the pre-trained YOLO models that are based on the COCO dataset [44]. The dataset comprises 80 object categories which strikes a good balance between having a dataset that can accurately determine one particular class but also has knowledge of many other classes that it can then ignore and not falsely classify as a chair or person.

3.2 Human-oriented design

In the field of service robotics, one must ensure that the solutions they build are well tailored to any person that may encounter them.

A meta-review of the landscape of social robotics in public spaces [45] proposed multiple guidelines by which robots should engage with humans. Robots must have a "proactive approach" to initiate any potential interaction first or risk losing or never gaining a person's interest, combined with a preference to speak shorter phrases that can be quickly understood and parsed by humans.

Furthermore, a robot can be designed to provide rich and complex interactions that appear more natural and hence more approachable to humans. One such suggestion comes from a paper that observed interactions between humans and a robot in different spaces such as a reception and other social events, their design recommendations state that in group interactions, the robot must be able to "switch attention from one person to the other and follow the rhythm of the interaction" [46]. This includes understanding behaviour of humans around the robot, such as where they are facing or what sort of movements they are performing.

We may also consider understanding a person's behaviour in terms of their speech patterns. A paper analysing human-robot conversations in the real world found humans would often not follow many social norms or otherwise show very low verbosity in what they said to a robot [47]. Understanding this behaviour, we could adapt the robot to converse similarly to the person, such as increasing or decreasing the verbosity of the robot's speech.

Another more specific improvement relating to the project is that a 'roboreceptionist' should opt to "identify individuals open to interaction rather than bothering every passerby" [46]. For example, looking for social cues such as a person looking at the robot for an extended period, moving towards the robot, and so forth.

3.3 Accurate segmentation of body parts

Nakajima et al. describe a full-body person recognition system using 'SVM-based classifiers', the "experimental results show high recognition rates" and is said to work in real-time [48]. However, it does not directly provide segmentation results for different body parts so it is not suitable. Juang et al. propose a method for estimating posture and therefore body segmentation by using "body silhouettes and skin-colour information" [49]. It is claimed that the proposed approach can "efficiently and effectively locate significant body points for most postures". However, this approach requires the generation of silhouette data which is gathered by segmenting moving objects in the scene, this makes it unsuitable as we are expecting the guest to be relatively still and facing the robot.

Zhu et al. propose a method for "multi-person detection and 2-D pose estimation that achieves state-of-art results on the challenging COCO keypoints task" [50]. It works by first trying to find smaller parts of the image likely to contain people using a "fast RCNN detector", then using these smaller images it tries to estimate keypoints that correspond to each potential person in each. This results in an average precision of 65% [50] on the COCO test dataset.

Later work by Zhu et al. proposes "a box-free bottom-up approach for [..] pose estimation and [..] segmentation of people in multi-person images using an efficient single-shot model" [51], this system results in an average precision of 66.5% for 'single-scale inference' and 68.5% for 'multi-scale inference' [51] outperforming other solutions.

The work by Zhu et al. culminates in the BodyPix [52], model that can provide us with real-time body segmentation. Further work after that release added support for multiple people and improved accuracy by using ResNet50 bodypix.

3.4 Reliable and robust conversations

To be able to hold a conversation with a user, we must implement three key features: text-to-speech, speech-to-text, and dialog intent recognition. Text-to-speech is already implemented for us on the TIAGo platform, so we just need to find adequate solutions for speech and intent recognition.

3.4.1 Recognising human speech

Mozilla DeepSpeech is an "open source embedded speech-to-text engine" that can run in real-time on a variety of high and low-power devices [53]. It is based on a research paper by Baidu [54] that presents an (at the time) "state-of-the-art speech recognition model" that achieves a 16% error rate [54] on the Switchboard Hub5′00 dataset.

Vosk [55] is another offline speech recognition model that is especially notable because it has first-class support for ROS through their ros_vosk library [56]. A meta-review by AlphaCephei [57] (the organisation behind vosk) describes the relative performance of vosk to other models including two trained on the GigaSpeech dataset [58], NVidia Transducer [59], and the various OpenAI Whisper [34] models (tiny.en, small.en, large.en).

The meta-review concludes that Vosk's accuracy is "outdated, need to improve as-soon-as-possible" and that relatively to Whisper, "other models are not that bad too like GigaSpeech models [..], if you look for best accuracy they are about the same" [57]. Whisper itself can perform speech recognition in real-time even with constrained resources and is quite accurate in noisy environments so it is an appropriate choice for this component of the project.

3.4.2 Recognising speech intent

Dialogflow is "a natural language understanding platform [.. for] conversational interfaces" [60]. An evaluation by Muhammad et al. found almost all the agents they designed using DialogFlow had an accuracy rate of 100% [61]. However, the use of DialogFlow requires us to interact with a cloud-hosted API which comes with additional charges [62]. While the paid API is not necessarily a deal-breaker as Google provides enough free credit to develop the solution, the always-online/latency as a result of this architecture makes it unsuitable for this project.

RASA [35] is a "conversational AI framework" that allows you to train a neural network to understand and interpret conversation transcripts. It would prove very easy to integrate into the project as they provide a Python library we can hook into [63] and a straightforward CLI interface for building and training new models [64].

3.5 Approaches to face reidentification

Face reidentification consists of two problems, how do we find a face in an image and how do we match faces to each other?

First, we focus on face recognition which is essentially a solved problem, work on face recognition as far back as 2014 was already achieving results with accuracies of "97.35% on the Labeled Faces in the Wild" dataset [65]. Labeled faces in the wild [66] being a popular benchmark used to test these models. The current consensus is that the human accuracy on this dataset is about 97.53% [67]. VGG-Face [68], [69], a model developed by a group at University of Oxford, achieved 98.78% [67] accuracy which surpasses human ability.

Two other notable models include: SFace [70], which is a model built to address privacy concerns with existing face training datasets; breaking from the status-quo, SFace is trained on synthetic face images and performs exceedingly well with 99.60% accuracy [67]. Facenet512 [71], which is the best performing model seen so far with 99.63% accuracy [67].

Next, we need to tackle matching previously learned faces to newly gathered face images. All of these models produce mathematical representations, or vectors, of given faces, so we just need to figure out how we can store and look them up. The DeepFace framework [30] provides an end-to-end solution as a Python library that provides access to most of the facial recognition models mentioned above but also allows us to create a local database of face images, from which the library automatically builds and manages a cache of the vector output from each model. The library then looks up the closest vector to match faces.

However, the scalability of using a local directory of face images with a cache is not great, so we can instead opt to use a purpose-built vector database. Weaviate [33] is an "open-source vector database" that can let us store and lookup vectors as well as any arbitrary associated data that we care about, in the context of this project, this would be a unique identifier for a guest such as an index or a name.

We also see that there is an alternative approach to solving this problem by implementing a generic 'image search engine' [72] using the previously mentioned vector database. Weaviate features built-in vectorisers such as text-to-vec, image-to-vec, and CLIP. One could use the resnet50 image-to-vec vectoriser in order to convert face images into vectors and then look them up after the fact. (though we would first have to align to the face, which we can do using the body segmentation mentioned earlier or using a YOLO model trained on faces [73])

4 Specification & Design

We look at the procedure outlined in the RoboCup rulebook [9] for the receptionist task. A copy of the full ruleset and expectations are included in the appendix, Section B.

The task is composed of three main phases:

1. Wait for, greet, and learn about new guests
2. Seat new arriving guests in the lobby
3. Describe the first guest who arrives to the second guest once they are both in the lobby

This section will describe in more detail how the three phases above should be implemented in practice and how we expect the robot to perform.

4.1 Requirements

4.1.1 Functional Requirements

At its core, the main requirements are:

T1 Able to navigate to preconfigured beacons in the arena

T2 Wait for a person to appear and approach them

The robot should only react to people who are directly in front of it during the setup.

T3 Generate a description of a guest including at least 4 characteristics

T4 Engage in conversation with a guest asking for name & favourite drink

T5 Learn a guest's face features and then later reidentify them

This should be done during guest setup, and reidentified during later interaction.

T6 Direct guests to sit in empty seats in the waiting area

4.1.2 Non-functional Requirements

It is also important to consider the robustness and human aspects of the project, for this the following requirements will be evaluated:

N1 Robot should maintain gaze with any person it talks to

Penalties are applied if this is not met.

N2 Robot should look in the direction of travel when moving around

Penalties are applied if this is not met.

N3 Robot can reliably detect and reidentify faces

The robot needs to be able to reidentify guests correctly in order to face the correct individual that we want to speak to during phase 3.

N4 Robot can reliably recognise speech and intent

The robot needs to be able to consistently understand what different guests are trying to communicate in order to complete phase 1.

N5 Robot can reliably generate the selected characteristics for a guest

It is necessary to provide a valid description of the guest during phase 3.

4.2 Nodes & Services

This project needs multiple auxiliary nodes that processing is offloaded to or otherwise that provide information necessary for the completion of the task.

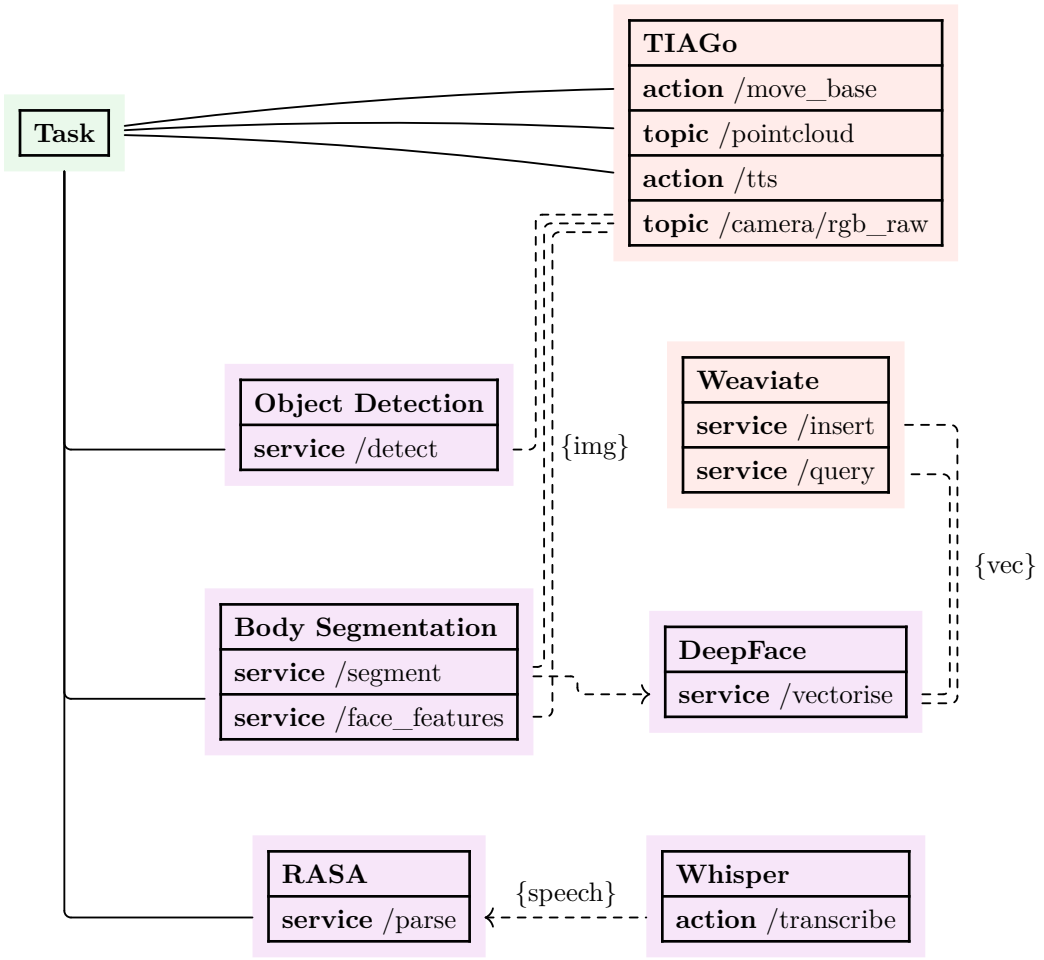


Figure 8: Interactions between nodes

This diagram shows which services we must implement and their rough dependence on each other within the task. Dotted lines indicate that this node relies on information that comes from another node, but not necessarily that the node itself accesses it directly.

Navigation, a pointcloud (depth information mapped to 3D space), text-to-speech, and camera images are provided directly by a node running on the robot indicated by **TIAGo**.

The **Object Detection** node provides a service that accepts an image (we use a frame from the robot's camera), runs this through a pre-trained COCO [44] YOLOv8 [28] model and spits out detections in the form of Data Structure 8. The actual service will be defined as follows.

Data Structure 7: YOLO Detection Service

YoloDetection.srv
- image: **sensor_msgs/Image** - dataset: **string** - confidence: **float** - nms: **float**
- detected_objects: **YoloDetection**[]

The **dataset** value will be used to select the specific model that we want to use, the **confidence** value is the minimum required confidence for a detection to not be pruned from the results, the **nms** (non-maximum supression) is a guiding value to remove overlapping detections.

Data Structure 8: YOLO Detection Result

YoloDetection.msg
- name: **string** - confidence: **float** - xywh: **int**[] - xyseg: **int**[]

Each detection provides the object found, the confidence that it is that object, the pixel-space coordinates of the bounding box, and the pixel-space line segments that form the segmentation mask.

The **Body Segmentation** node will provide two services, one for segmenting an entire body and one for segmenting features from a picture of a face. The service segmenting the entire body will accept an image and a list of requested masks, it will then run the image through a chosen BodyPix [31] model (the two main options are using: resnet50 or mobilenet50) with the given image and process the results by selecting the masks that were requested.

Data Structure 9: Segment Body Service

BodyPixDetection.srv
- image: **sensor_msgs/Image** - dataset: **string** - confidence: **float** - masks: **string**[][]
- masks: **bool**[][][]

The **dataset** and **confidence** values function in the same way as the object detection node, the **masks** value in the request indicates a list of requested masks where we specify a list of body parts for each mask, and as such the **masks** value in the response indicates a list of 2D arrays representing the segmented mask.

The node will also provide a service that given an image of someone's head, will use a custom-trained PyTorch model to segment their hair, face, glasses (if present), hat (if present). Using these masks, it will get the mean colour of the hair, glasses, and hat. The service is defined as follows.

Data Structure 10: Segment Face Service

FaceFeatures.srv
- image: **sensor_msgs/Image**
- features: **FeatureWithColour**[]

Data Structure 11: Feature Information

FeatureWithColour.msg
- name: **string**
- colours: **ColourPrediction**[]

ColourPrediction.msg
- colour: **string**
- distance: **float**

The **RASA** node will be a simple wrapper around the RASA API, it is defined as follows.

Data Structure 12: RASA Service

Rasa.srv
- text: **string**
- response: **string** (JSON-encoded string)

The **Whisper** node will provide an action server for transcribing speech from a connected microphone, there will be no arguments to the action. When invoked, it will start listening and capturing audio, transcribe it incrementally using the Whisper speech-to-text model, and then return the final transcription when it believes the phrase is complete.

Data Structure 13: Transcription Action

TranscribeSpeech.action
- transcribed_speech: **string**
- transcribed_speech: **string**

The **DeepFace** node will provide a service that accepts an image with a face in it, vectorises it using a model provided by the DeepFace framework [30], and returns the vector accordingly. This service is defined as follows.

Data Structure 14: DeepFace Service

DeepFace.srv
- image: **sensor_msgs/Image** - model: **string**
- detected_faces: **Embedding**[]

Embedding.msg
- vector: **float**[]
- confidence: **float**
- xywh: **int**[]

The **model**, **confidence**, and **xywh** values are like we've seen before, a **vector** is a series of floating point numbers representing the face features.

The **Weaviate** node will be a thin wrapper around the Weaviate vector database, providing simple services that allow us to insert a vector with one or more properties, and then query vectors.

Before we can use the database, we have to first initialise a collection for our vectors. We provide a service that can do this and also handle the conditions where the collection exists, in which case we can either ignore it, error out, or clear the collection.

Data Structure 15: Weaviate Create Collection Service

CreateCollection.srv
- name: **string** - skip_if_exists: **bool** - clear_if_exists: **bool**

Now, we provide a service to insert vectors with any arbitrary properties.

Data Structure 16: Weaviate Insert Vector Service

InsertVector.srv
- name: **string** - properties: **Property**[] - vector: **float**[]

Property.msg
- key: **string** - value: **string**

Finally, we provide a service that lets us query these vectors.

Data Structure 17: Weaviate Query Vector Service

QueryVector.srv
- name: **string** - limit: **int** - vector: **float**[]
- results: **SearchResult**[]

SearchResult.msg
- certainty: **float** - properties: **Property**[]

4.3 Task Design

A state machine will be used to implement the task requirements, it will be implemented using the following overarching design:

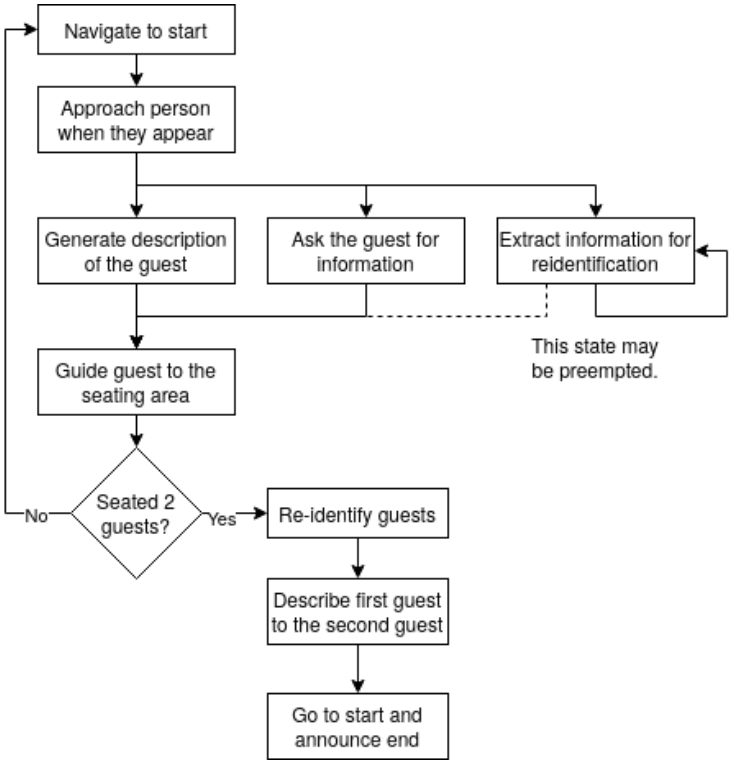


Figure 9: State machine

But before we run the task, we need to prepare information about the arena and ensure the program can correctly identify where it is. We provide a setup instructions that detail how the user to configures the start/end position in the arena, the waiting position, the position of the seating area, the area of the waiting area, the name & favourite drink of the host, and information required to reidentify the host (Figure 13 describes the steps necessary to learn a new face, this will be described in more detail below). This can be provided through a configuration utility or we can instruct the user to write information into specified files.

Keeping track of information throughout task

However, we still need to consider how we keep track of all of this information. To begin, we consider the persistent information such as the arena information and the host's information. We can use the following data structure within the parameter server:

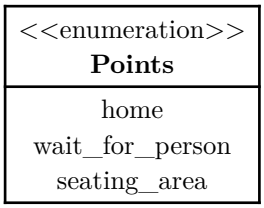
Data Structure 18: Data on Parameter Server

/receptionist
- → /points → /<name>: **float**[]
- → /polygons → /<name>: **float**[]
- → /host → /name: **string**
- → /host → /face: **float**[]

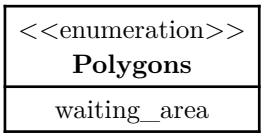
This information will persist through invocations.

The point and polygon names are described by the following enums:

Data Structure 19: Points enumeration

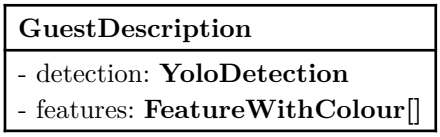


Data Structure 20: Polygons enumeration

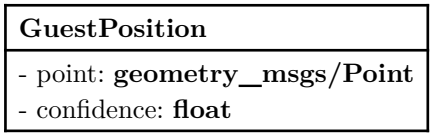


Next, we want to consider how to store ephemeral information such as the learned guest's name, description, position (in the seating area), and favourite drink. First, we define two new data structures that represent the more complex information such as their description and position, for which we

Data Structure 21: Guest description

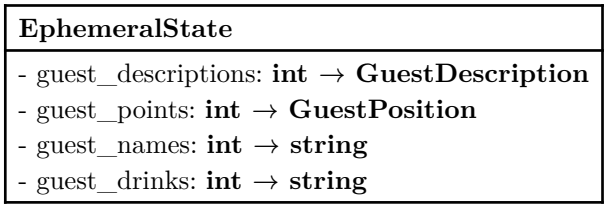


Data Structure 22: Guest position



Then we define a class that contains all of this information. We use dictionaries that map the guest's index to the respective features. This class will be instantiated once and then passed through the state machine to all states so that we can read/write this data at any time.

Data Structure 23: Ephemeral state (persists through task)



Navigation to different beacons

TIAGo's navigation stack can already handle planning around obstacles and looking in the correct direction while navigating, such that we just need to send a target goal to the move_base server running on the robot.

Wait for a person to appear and approach them

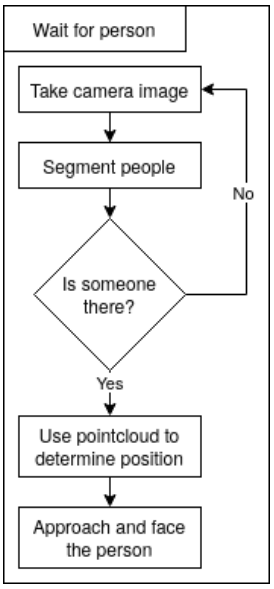


Figure 10: "Wait for a person"

To implement this state, we need to position ourselves in front of the waiting area then continuously fetch camera and depth information from the robot. There will be a object detection service that uses YOLOv8 (trained on COCO [44]) to segment out the objects in the scene, the service will accept the camera image and return a response containing the detections (Data Structure 8). We can filter the list of detections where `name: person`.

These segmentation mask(s) can be used to select points in the pointcloud (a 3D representation of the depth information) corresponding to the people in the scene. We can then filter the people positions for those who are in the waiting area we defined earlier. If there is one or more people (though we shouldn't ever see more than one person in the waiting area) then we can select the closest or any random person and approach them by planning to an unoccupied point within an area around them, then turn to face them.

Generate a description of a guest

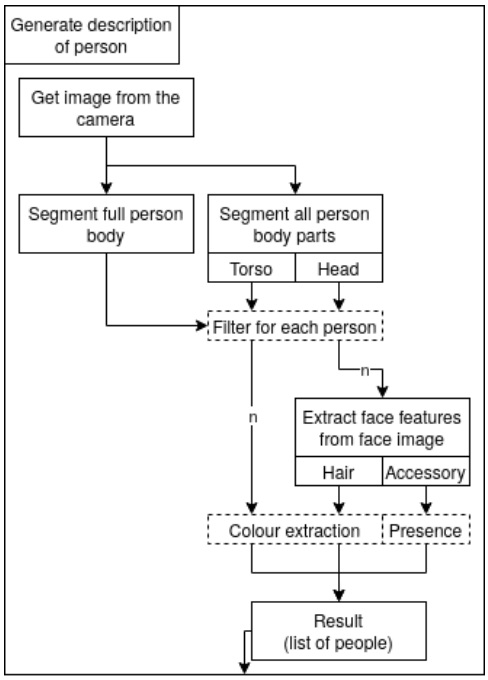


Figure 11: "Describe guest"

This state will take an image from the robot's camera and feed it into both the object segmentation service described earlier and a body part segmentation service, this new service will accept the camera image and feed it into BodyPix along with a requested set of body parts (we want to find the torso and head). The service will then return a set of segmentation masks corresponding to the list we requested.

Now, we can combine the information we've gathered. First, we filter the object segmentation results for only people as described before, and now we can use the segmentation masks for each person to mask out different body parts, and those new masks are now in turn used to mask the original image to find colour information. We do this as the body part segmentation model won't give us enough information to distinguish between different people so we have to do it ourselves. At this point, we can now also determine the colour of the torso by taking the mean colour in the segmented area and finding the closest known labeled colour.

Then for each person, we can process the segmented head to extract additional information such as if they are wearing glasses or a hat, or find the colour of their hair. To do this, we can take an existing dataset that labels and segments these characteristics and train it using a tool such as PyTorch. We package this into a new service that accepts a picture of someone's

face, segments it, tries to determine the colour of the segmented image, and spits out a list of features (Data Structure 11).

Combining all of this information, we can now extract the colour of someone's torso, the colour of their hair, whether they are wearing glasses, and whether they are wearing a hat. We take all of the data gathered and format it as Data Structure 21 which can be stored in the ephemeral information passed between states (Data Structure 23).

Ask guest for their favourite drink and their name

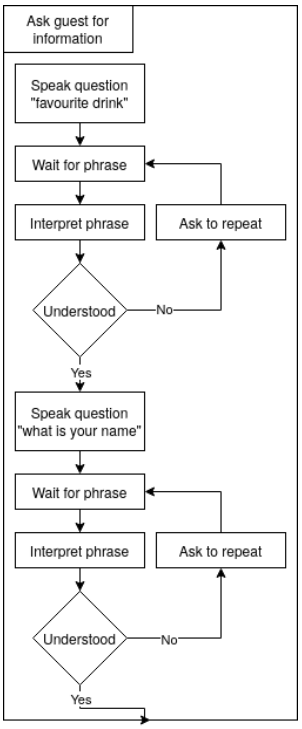


Figure 12: "Ask guest for information"

This state is quite simple, it repeatedly asks the guest their favourite drink until they're understood, and then likewise prompts the guest for their name until they're understood. To talk to the user, we use the TIAGo's onboard text-to-speech service which we can simply call with a string to be read out.

To implement it, we will deploy two new services: one that provides speech recognition and another for dialog intent recognition. The speech recognition service will implement an action server that listens on a given preconfigured microphone, feeds the audio data in chunks into Whisper speech-to-text model and returns a string containing the transcribed speech after a phrase is detected. If it fails to recognise any speech, it will prompt the user to repeat and continue to generate a new transcription.

This transcription will be sent to the other new service that passes the given string to a custom RASA model that is trained to understand responses such as "My name is Adel." (intent: name) and "orange juice" (intent: drink). The given response will be processed by the state and if the intent is unclear, the system will ask the user to try again.

Extract information for reidentification

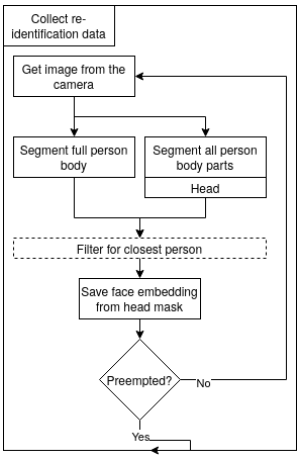


Figure 13: "Collect reidentification data"

This state will take an image from the robot's camera and, like the description state, segment both the full body and body parts using the appropriate models. We specifically segment the closest person's head and then convert this image into an embedding that we can later query. To convert the image to an embedding, we need to vectorise the face features, we can use many different models described earlier in the face reidentification literature section.

This state will run concurrently when we speak with the guest and it will keep gathering data until it is preempted. This is done to gather as many embeddings of the person as possible and as such produce a much more reliable dataset.

Guide guests to an empty seat and ask them to sit down

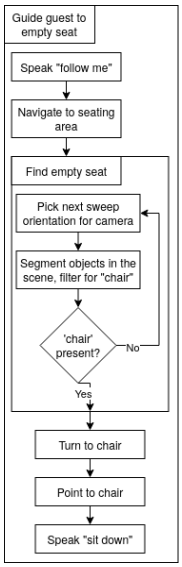


Figure 14: "Guide guest"

This state will make the robot tell the guest to follow it to the seating area. Once the robot arrives at the seating area, it will start sweeping the camera and will take a camera image with depth information in each direction to try to detect an empty chair. To sweep the camera, we can send a 'motion' to the TIAGo's head controller, we define target destinations where we are looking slightly left, slightly right, and centre. To detect the chairs and their positions, we use the same strategy as when we detect and describe a guest.

However, we specifically want to find empty chairs, so we opt to filter for both person and chair detections from the object detection service. We can now omit all of the chair masks that intersect person masks, hence finding all of the empty chairs.

To direct the guest to the chair, we will turn to it and point at it with a predefined gesture using the TIAGo's arm. Finally, the robot will tell the guest to sit down in the given seat. If we've only met one guest so far, we need to the waiting area and meet the next guest.

Reidentify guests

At this point, the robot will be in the seating area and must reidentify all of the guests. We cannot simply remember where we sat the guests down because they may move at any point, and for RoboCup competition runs, guests have been instructed to move after the robot leaves.

We use the same strategy as described before to generate a face image (Figure 13), we can then vectorise this, and query the embeddings database we populated earlier through the task to determine which face corresponds to each guest, and hence find the centroids of each person.

Describing the first guest to the second guest

For this state, we use the guest information from the reidentification step prior. The robot must turn to the second guest and speak details found earlier about the first guest. This state could be made more natural by doing a formal introduction between the two guests, turning to face each other, and for example, saying the other's name.

Ending the task

To end the task, the robot will navigate back to the starting position and announce that the task has ended.

5 Implementation

5.1 Development tooling and environments used

For this project, the editor of choice was Visual Studio Code [74] with the catkin_tools [75] build system. While Visual Studio Code is not necessarily considered an IDE, it can still work like a traditional IDE and provides some additional useful features that were taken advantage of. VS Code still provides language server features through extensions and offers the ability to remotely work on another machine through SSH. catkin_tools is the successor to catkin_make [76] which itself is just a thin wrapper around cmake, but it additionally provides functionality such as isolated and parallel builds out of the box, which significantly reduces the time to build the project.

Throughout the project, different environments were used depending on changing requirements throughout the development process, this section describes what was used and justifies why it was most appropriate.

5.1.1 Environment 1: Virtual machine with ROS Noetic

At the start of this project, a virtual machine was configured on a private server that could then be accessed remotely through SSH and consequently allowed development through VS Code. To achieve this, Ubuntu 20.04 was used as the operating system, and then ROS Noetic and the TIAGo public workspace were installed on top. Any additional dependencies were also directly installed into the virtual machine. If a graphical environment was required, one could connect to the machine over VNC running on the host machine.

This solution allowed development to take place anywhere regardless of the operating system in use or the hardware configuration of any one computer it was accessed from.

5.1.2 Environment 2: Bare metal with Apptainer

Apptainer [77] is a program that allows us to build and run containers. Containers are essentially self-contained images usually containing entire operating systems in themselves, or sometimes just an application with all the dependencies required to execute it. PAL Robotics provides us with built containers that include the correct version of Ubuntu, ROS Noetic, and the dependencies to run the TIAGo simulator or otherwise interact with a TIAGo robot. We can use Apptainer to bootstrap from this image and add our dependencies on top.

The resulting container [78] has a few notable changes including: Python 3.9 and Python 3.10 which are not available by default but we need for certain dependencies due to their strict version requirements. Various Python packages such as numpy, scipy, tensorflow, etc are dependencies to other Python packages we will later install or use within our workspace. An "overlay workspace" that includes additional ROS packages that we rely on.

A notable ROS package that we include in the overlay is catkin_virtualenv [79] which allows us to directly bundle any Python requirements we need within individual ROS packages via the use of Python virtual nvironments. This is essential to allow us to use a specific version of Python where libraries necessitate it, for example, the ultralytics [28] package requires at minimum Python 3.9, where the container uses Python 3.8 by default.

5.2 ROS Packages

This section details ROS packages that I've built or contributed to that were necessary to build this project.

5.2.1 lasr_vision_yolov8

This package provides us with the ability to do object detection and segmentation from any ROS node. It is equivalent to the object detection node described in the specification, however, to simplify the final implementation of 3D detections of objects and people, another service was also created that accepts pointcloud information to map the detections into map space.

As such, the node provides a service `/detect` which is implementing according to Data Structure 8, and it provides another service `/detect3d` which is implemented according to the data structure below.

Data Structure 24: YOLO 3D Detection Service

YoloDetection3D.srv
- pcl: **sensor_msgs/PointCloud2** - dataset: **string** - confidence: **float** - nms: **float**
- detected_objects: **YoloDetection3D**[]

YoloDetection3D.msg
- name: **string** - confidence: **float** - xywh: **int**[] - xyseg: **int**[] - target_frame: **string** - point: **geometry_msgs/Point**

Under the hood, it uses the ultralytics [28] Python library to load and use YOLOv8 models. When either of the services are invoked, the first thing that occurs is the **dataset** value is read and the corresponding model is instantiated. We use a basic caching mechanism here to avoid having to initialise models more than once:

Algorithm 1: Basic Cache Algorithm

input: dataset: *string*
output: instatiated_model: *Model*
globals: loaded_models: *int* → *Model*

```
1 if dataset in loaded_models then
2     instatiated_model ← loaded_models[dataset]      ▷ use previously loaded model
3 else
4     instatiated_model ← Model(dataset)              ▷ load the model into memory
5     loaded_models[dataset] ← instatiated_model      ▷ save it for later use
6 end
```

The actual detection service works as follows:

Algorithm 2: YOLO Detection Service

input: request: *YoloDetectionRequest*
output: response: *YoloDetectionResponse*
globals: load_model: *string* → *Model*

requires: cv2_img ▷ cv2_img is another package described later in this section

```
1 image ← cv2_img.message_to_image(request.image)          ▷ prepare image
2 model ← load_model(request.dataset)                      ▷ load the requested model
3 result ← model(image, request.confidence, request.nms)   ▷ infer †he results
4 response.detected_objects ← result                       ▷ return the result from the model
                                                           ▷ in the actual implementation, the result out of the model needs to be converted to the response type
```

However, in the case of 3D detections, we need to flatten the given pointcloud into a 2D image from the camera's perspective, we replace the first step in the service as such:

```
1 − image ← cv2_img.message_to_image(request.image)        ▷ prepare image
2 + image ← cv2_pcl.pcl_to_image(request.pcl)              ▷ convert pointcloud to 2D image
```

The **cv2_pcl** library provides various utilities for working with pointcloud messages, in this case we only care about **pcl_to_image** which flattens the pointcloud, and **seg_to_centroid** which converts a pointcloud and given 2D segmentation mask into a point in 3D space (the centre of the points selected by the segmentation mask).

So to then also determine the actual position of the detected objects we use this additional method provided above. However, this will put us in the camera coordinate frame, so we must transform it to the map frame using tf (for the purposes of demonstration, the code below assumes all transforms were gathered at the same time and that the camera frame is relative to the robot base which is relative to the map frame).

Algorithm 3: Determining object detection centroid

input: pcl: *PointCloud*, detection: *YoloDetection3D*
output: point: *Point*
globals: tf_buffer: *TfBuffer*

```
1 centroid ← cv2_img.seg_to_centroid(pcl, detection.xyseg)   ▷ find centroid
2 point ← tf_buffer.transform(centroid, "map")               ▷ transform point to map frame
```

transform() will perform the following: ▷ (simplified to explain how the transforms work)

input: point: *Point*
output: transformed_point: *Point*
buffer: T_c^r, T_r^m: *matrix transforms from camera to robot, robot to map*

```
3 T_c^m ← T_c^r * T_r^m [80]                                 ▷ find the transformation from camera to map
4 transformed_point ← T_c^m * point                          ▷ transform the point accordingly
```

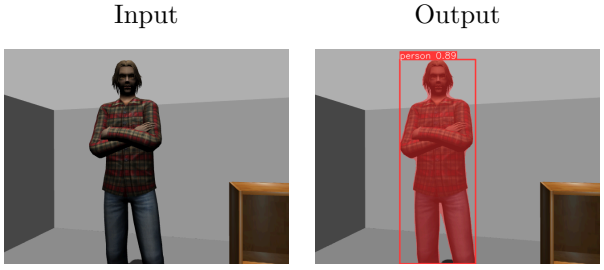


Figure 15: YOLO Inference

5.2.2 lasr_vision_torch

This package uses PyTorch to run inference with a custom-trained face segmentation model as a service `/torch/detect/face_features` defined by Data Structure 10.

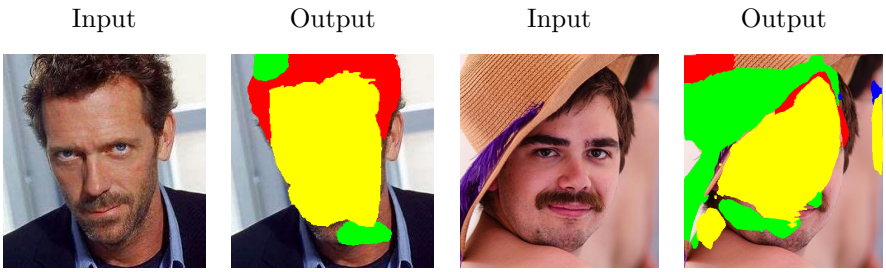


Figure 16: Inferences on Hugh Laurie (House M.D. [81]) and Morten Rieger Hannemose (recreation of Lena [82])
Segmented areas in figures: face, hair, hat

Disclosure

Package was developed in collaboration with a colleague on the KCL LASR robotics team. Work as part of this project was limited to integrating a pre-trained model into a ROS package that could then be used to provide these inferences.

5.2.3 lasr_vision_deepface

This package provides a service `/deepface/detect` implemented according to Data Structure 14. The service uses the DeepFace Python library [30] to find and represent all faces in a given image as vectors with their corresponding bounding boxes.

However, unlike the proposed implementation, we first perform alignment which is the process of roughly estimating where faces exist in the image and generating a smaller bounding box that we can then process with the purpose-built facial detection models. By default, the library uses the OpenCV Haar Cascade face detection algorithm [83], which is fast and efficient but is very error-prone. We opt to use mtcnn which is slower but promises "superior accuracy" in theory and experimentation [84].

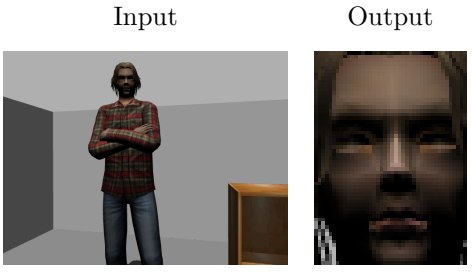


Figure 17: mtcnn Inference Result (using DeepFace framework)

After alignment, we use the requested model to generate a vector representing each face. In this project, we opt to use VGG-Face for all inferences.

5.2.4 lasr_vision_bodypix

This package provides the `/bodypix/detect` service implemented according to Data Structure 9, and uses the tf_bodypix [32] library to load a pre-trained BodyPix model for use in inference on the given images.

The library outputs masks in the form of a 2D boolean array which is not possible to directly represent as a ROS message type. We have to flatten the array and then reshape it on the other end. The new service hence is defined as follows:

Data Structure 25: Amendments to service

BodyPixDetection.srv
[..]
- masks: **BodyPixMask**[]

BodyPixMask.msg
- mask: **bool**[]
- shape: **uint32**[]

To perform the flattening and reshaping, we use the numpy library:

Algorithm 4: Flattening and reshaping data

Given a 2D numpy array **mask**

```
1 message.mask ← mask.flatten()
2 message.shape ← mask.shape
```

Given the mask message **message**

```
3 mask ← numpy.array(message.mask).reshape(*message.shape)
```

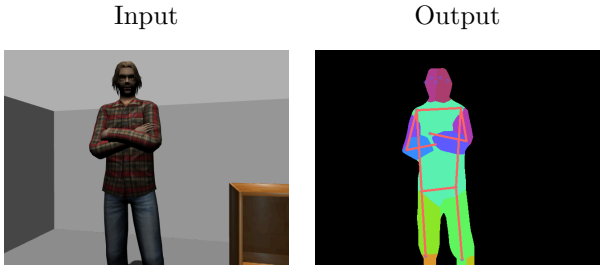


Figure 18: BodyPix Inference

5.2.5 lasr_vector_database_weaviate

This package provides a way to quickly deploy the Weaviate [33] vector database within our containerised environment and then provide a ROS native way to communicate with it. To deploy Weaviate, the project recommends installing Docker and using their configuration generator to build and run the database, this includes configuring modules such as vectorisers from text or images. Unfortunately, this is not feasible for our environment as quite often we run in unprivileged Linux environments with no option to install Docker or alternatives such as Podman (which would let us run the actual containers without root!).

Digging around on their source code repositories, you can find each tagged release gets corresponding binary releases built and uploaded to GitHub [85]. However, this would not be enough to run the database. Inspecting the Dockerfile used to build their images you can find found what sort of arguments are required to start the binary [86]. The sample compose file also provides what environment variables are needed (one can generate one in their documentation [87]).

Using all of this information, the provided node in the package performs the following: Download the chosen Weaviate binary if it does not exist yet (we provide the rosparam `~version` to select the appropriate release). Unpack, configure and launch the database as a subprocess. We must allocate to some unique port on the local device to communicate between our node so we choose two ports which can also be changed using the rosparams `~http_port` and `~grpc_port`. We also allow the end user of the package to run multiple databases by changing the name of the database using the param `~db_name`.

After the subprocess is created, we poll the HTTP API until it is available, after which we create the three services described earlier in the specification: `/database/vectors/{name}/create_collection`, `../insert`, and `../query`. These are defined by Data Structure 15, Data Structure 16, and Data Structure 17 respectively. These simply map the requests from ROS message types to/from the Weaviate API JSON types and send the requests to the Weviate REST API.

5.2.6 cv2_img

Quite often when video or images are sent over ROS, they are serialised as single raw frames that are sent to a topic. However, when dealing with images in Python, one would quite often either have them as an (RGB) Pillow Image or simply as a (BGR) numpy arr (also referred to as a cv2 image).

This package provides a Python library that simplifies converting between images sent over ROS and the format we want to use in Python. It provides the following functions:

```python
def pillow_img_to_msg(img: PIL.Image, stamp=None) -> SensorImage:
def cv2_img_to_msg(img: numpy.ndarray, stamp=None) -> SensorImage:
def msg_to_pillow_img(msg: SensorImage) -> PIL.Image:
def msg_to_cv2_img(msg: SensorImage) -> np.ndarray:
```

5.2.7 colour_estimation

When finding descriptions of people's appearance, we take the mean of the colour of certain parts of their body, such as the torso. To assist with then translating this into natural language, this package provides the following dictionaries and functions:

```python
# mappings of colour names to RGB values
RGB_COLOURS = { 'red': [255, 0, 0], .. }
RGB_HAIR_COLOURS = { 'midnight black': (9, 8, 6), .. }

# function that takes the requested colour
# and a dictionary defined above
# and outputs a list of the closest colour names
# with their distances to the requested colour
def closest_colours(requested_colour, colours):
```

Algorithm 5: Closest Colour Algorithm

input: r requested_colour: *int[3]*, colours: *str* → *int[3]*
output: closest_colours: *(name: string, distance: float)[]*

distances ← {}
for colour_name, v rgb_value ← colours **do**
  $\partial \leftarrow v - r$
  distances[colour_name] ← $|\partial|$
end
sorted_colours ← sorted(distances, **key**: item: item.value)
closest_colours ← sorted_colours[:3]

5.2.8 lasr_speech_recognition_whisper

This package provides the action `/transcribe_speech` implemented according to Data Structure 13. To enable access to the Whisper speech-to-text model, we can use their Python package [88] which handles the process of downloading, loading, and performing inference on our chosen model. We also use the SpeechRecognition [89] Python package which provides many useful utilities that simplify recording audio, capturing likely phrases, and adjusting to the environment.

To select the input device, the end user can provide an argument to the transcription node. This can be either the name or the index of the microphone, found by running the provided `list_microphones.py` script.

Algorithm 6: Selecting Microphone

input: device: *string | int | None*
output: microphone: *Microphone*
requires: sr: *SpeechRecognition [89]*

if device **is** None **then**
  microphone ← sr.Microphone() ▷ try to auto-detect the system microphone

```
    else if device.isdigit() then
        microphone ← sr.Microphone(device_index = device)   ▷ use the given device index
    else
        for index, name in enumerate(sr.Microphone.list()) do
            if device in name then
                microphone ← sr.Microphone(device_index = index)   ▷ use by name
            end
        end
    end
```

Afterwards, the service implements the action as follows:

Algorithm 7: Whisper Service

requires: w: *Whisper [88]*, sr: *SpeechRecognition [89]*, actionlib: *actionlib [19]*

```
model ← w.load_model("medium.en")
microphone ← select_microphone(args.device)   ▷ configure the microphone input
recogniser ← sr.Recogniser()   ▷ instantiate the speech recongnition library
recogniser.adjust_for_ambient_noise(microphone)   ▷ learn background noise and set levels
actionlib.ActionServer(execute)
    execute()
    input: goal: TranscribeSpeechGoal
    output: result: TranscribeSpeechResult
    phrase ← recogniser.listen(microphone, ...)   ▷ start listening for phrase, args control timeouts
    wav_data ← phrase.as_wav_data()   ▷ WAV is a raw data format that we can feed into models
    normalised_data ← float(wav_data)/32768.0   ▷ normalise int16 to float values
    result ← model.transcribe(normalised_data)   ▷ run inference using Whisper
```

5.2.9 lasr_rasa

This package is a relatively straightforward wrapper around the RASA Python package [63], it provides the `/lasr_rasa/parse` service which implements Data Structure 12. In order to train our model, we can use the included CLI [64].

To create the receptionist dialog model, we first have to initialise the RASA project using `rosrun lasr_rasa rasa init` in a new directory where we intend for the project to live in. Now, we can define training examples by editing `data/nlu.yml` relative to the newly created project. We can specify our dialog intents as seen in Listing 1.

```yaml
version: "3.1"

nlu:
  - intent: person_name
    examples: |
      - My name is [Adel](name)
      - [Robin](name)
      - [Poppy](name)
      - ...

  - intent: favourite_drink
    examples: |
      - [Orange juice](drink)
      - [Cola](drink)
      - My favourite drink is [iced tea](drink)
      - ...
```

Listing 1: NLU training data

Now we can run `rosrun lasr_rasa rasa train nlu` to train on this given dataset. The trained model will be output as a `.tar.gz` archive in `models` relative to the created project. We can now use the package to start the RASA service using this new model.

5.2.10 lasr_skills

The lasr_skills [37] package is an effort by the KCL LASR team to produce common SMACH [21] states that can be re-used across many projects, this includes small building blocks such as navigating to a point, detecting objects, talking to the guest, and so on.

This project uses and contributes some states from/to this package. These are discussed at greater length in the following section.

5.3 Task

This section details the specific decisions made during the development of the project and explains why certain strategies were chosen to implement different things.

5.3.1 Storing Data & Setup

The project deals with a variety of different data that is collected before and during the task, different approaches are necessary depending on the type of data we are handling.

5.3.1.1 Persistent arena information

We use Data Structure 18 described earlier to represent persistent arena information, this information lives in the parameter server. To generate the data in the first place, we provide the user with a configuration CLI built using the typer Python library [90]. This library was chosen as it provides a very simple, straightforward, and scalable way to build CLI utilities:

```python
app = typer.Typer() # after defining the Typer application,
@app.command() # we can define a new command
def hello(): # this is equivalent to "python path/to/file.py hello"
  print("hello!")

app() # before we exit, we just invoke the app
```

So the CLI defines the commands: `load [name]`, `save [name]`, `points [set|get] [name]`,`polygons [set|get] [name] [count] (topic)`, and `learn-host [name]`. To begin, load and save simply copy the configuration in the parameter server from/to a JSON file relative to the project. We use `json.loads/dumps` from the Python standard library to convert and we can use `rospy.set|get_param('/receptionist'[, obj])` to set/load the entire state at once.

The points command works by waiting for the next published robot pose from TIAGo, published on `/amcl_pose`, and saves it as-is to the parameter server under `/receptionist/points/[name]`. This records the robot's current position and rotation so it can be used to return to this given position in the future. The user is asked to save points for the home point, the point where we wait for a new person, and the waiting area.

The polygons command works by waiting for n published points on a given topic, assumed to be `/clicked_point` by default, and collects them before saving them to the parameter server under `/receptionist/polygons/[name]`. The user is instructed to use a utility such as RViz [23] in which the user may select the "Publish Point" option in the toolbox and then select points in the map.

Algorithm 8: Polygon Selection

input: topic: *string*, n count: *int*
output: polygon: *float[][]*
requires: rospy: *ROS Python Library [12]*

```
polygon ← []
for i = 0..n do
    p ← rospy.wait_for_message(topic, geometry_msgs/Point)   ▷ wait for user input...
    polygon ↞ p_xy   ▷ we want to ignore the z axis as we are looking for areas in the 2D map only
end
```

Finally, the learn host command will save the guest's name and face vectors into `/receptionist/name|face`. To learn the host's face, we use the detect faces state machine discussed later in Section 5.3.4.2 to generate vectors that we store in the parameter server.

5.3.1.2 Epheremal guest information

For storing guest information during the execution of the task, such as guest names, favourite drink, positions, and descriptions, we use Data Structure 23. An empty copy of this data structure is initialised alongside the state machine on invocation and is then passed through to every state:

```python
sm = smach.StateMachine() # when we define our state machine,
state = EpheremalState() # we also define our state,
# ...
sm.add(MyStateMachine(state)) # and we pass it through to any sub-state(-machine)
```

However, information such as face data is stored using the Weaviate vector database described earlier as it will also provide a way to query this information efficiently later.

5.3.2 Scripts & Launch Files

To prepare for the execution of the task, we need to setup a bunch of required services. The `launch/services.launch` ile is a roslaunch file which simplifies running all of these. It performs the following configuration:

- Load the `parameters/motions.yaml` file into the parameter server, this contains necessary joint information for turning the robot's camera into certain positions such as: looking forwards, down, left, right, etc.
- Start the object detection service (preloading the YOLOv8 nano segmentation model), body segmentation service (preloading the BodyPix resnet50 model), face feature extraction service, face segmentation service, speech-to-text service (if a microphone is specified), stub services for text-to-speech (if we're not on the real robot), dialog intent recognition service (loading our custom-trained RASA model), and the vector database service.
- Start the health checker script (`scripts/check-all-running`). This scripts waits for all the aforementioned services to become available and then replaces the console with a message (this is achieved using the rich Python library [91]) indicated they have all started to eliminate the guesswork of waiting for everything to start.

The user executes this by running `roslaunch bsc_receptionist services.launch` and then can subsequently start the task or any unit tests by running `rosrun bsc_receptionist task/unit-0X/etc`.

5.3.3 State Machine

During the development of the project, I found that I had to make certain changes to the planned state machines. Figure 19 describes the final overarching state machine that I ended up implementing.

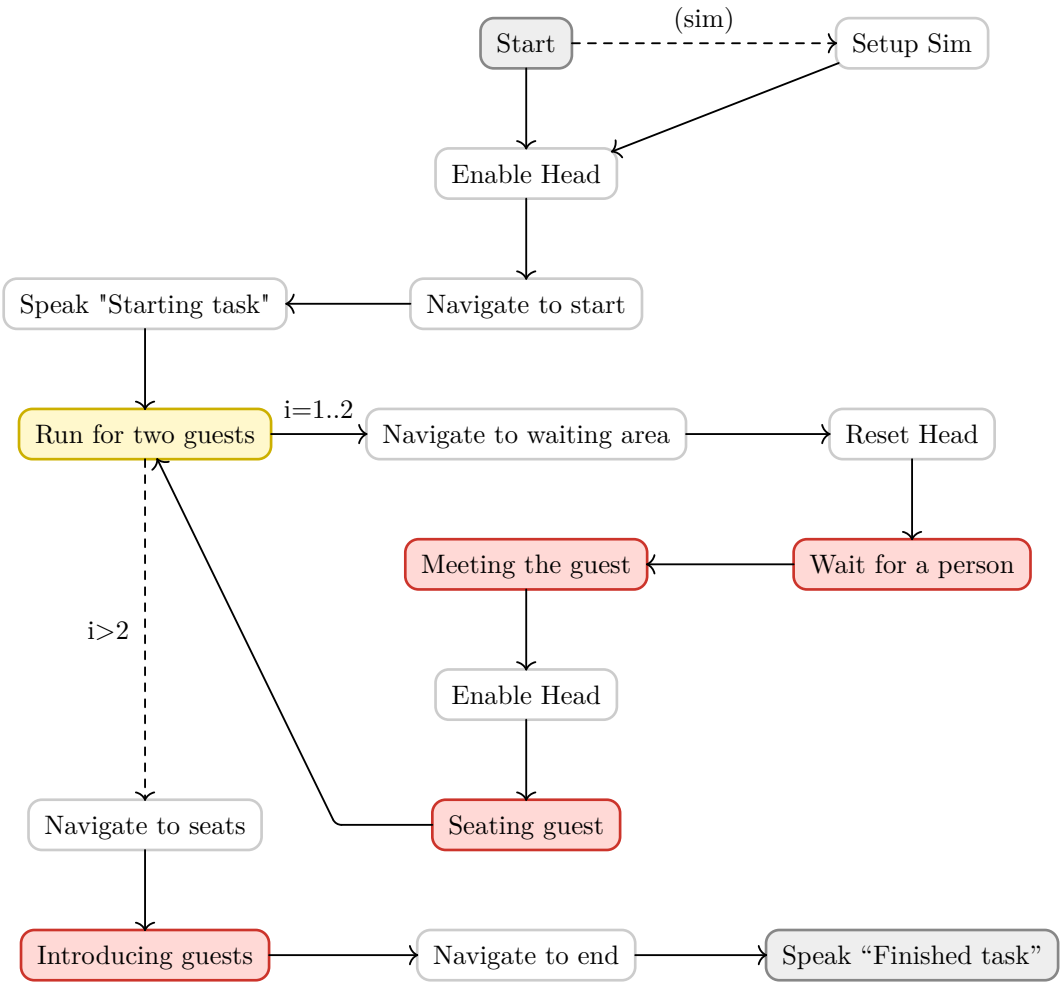


Figure 19: Overarching state machine for the task

5.3.3.1 Preparative Setup

Before we start the state machine, we need to prepare a few things. We begin by creating a new collection in the vector database with the name “Face” and we request that if the collection already exists, it is cleared. Afterwards, we check for the presence of host information within the parameter server. If information is present, we upload all of the vectors using the procedure described in Section 5.3.4.3.

5.3.3.2 Simulation Setup

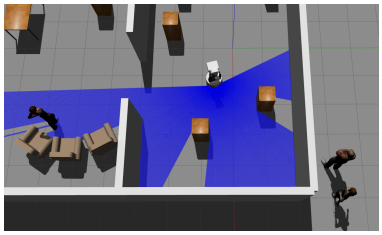


Figure 20: Automatic placement of objects in simulation

Initially, the entire project was built and tested within the Gazebo simulator. Manually setting up the scenario was cumbersome and annoying, and trying to setup a custom world was also infeasible due to the way TIAGo's simulation launch files were setup. So the solution was to investigate whether Gazebo offered any way to automate the spawning and deletion of objects as well as whether it could allow resetting the simulation as a whole.

To spawn new objects, Gazebo accepts SDF (Simulation Description Format) models, which are plain XML files that describe the mesh, physics, and other properties of the model. It accepts these on the `/gazebo/spawn_sdf_model` service along with a reference frame, name, and an initial pose. So we make a list of all the objects we want, e.g. 2 guests, 3 chairs, along with the target initial poses within the world. We can then iterate through this list and call the service to spawn them.

To then delete the object, we call `/gazebo/delete_model` with the name we specified earlier. To make things easier to later reset, all of the models we spawn are prefixed with `receptionist_`.

Each time the task is started (and we specify this when launching), the first state the machine starts at is the simulation setup. In this state we first reset the existing simulation by waiting for the next message on the topic `/gazebo/model_states` which provides us with a list of models in the world, we filter for those starting with our prefix and subsequently delete them, and finally we iterate through the list of predefined models and spawn all of them.

Gazebo also offers a way to reset the simulation directly, by calling `/gazebo/reset_simulation` service but this comes with two caveats making it unsuitable: models (spawned since load) do not despawn and the robot loses its localisation.

5.3.3.3 Head Manager

Head manager is a node that runs on the TIAGo robot that controls the looking direction of the head and automatically looks in the direction of travel/rotation when navigating. This allows us to automatically satisfy the direction of travel requirement however we must take care to disable/enable the head manager as appropriate, as it causes the robot to naturally look around when idle which is unintended behaviour when we need to look at guests.

To achieve this, we provide two states "Enable Head" and "Disable & Reset Head". The head manager allows us to control whether it is active via an action server that prevents it from working when there is an active goal, so when we disable it, we must send a disable action to `/pal_head_manager/disable`. But the head may still be looking in some random direction, so we also send a goal to "play motion" which is an action server running on the TIAGo robot that can play predefined motions. In our launch file, we loaded several motions onto the parameter server, we use "look_centre" which resets the head to look straight forward.

To then re-enable the head, we can cancel all goals sent to the action server.

5.3.3.4 Navigation to beacons

Navigation is relatively straightforward, we load the points defined earlier from the parameter server and then send an action to the `/move_base` action server (navigation stack) with the desired pose. If navigation fails due to poor localisation, the robot will automatically enter recover

behaviour which includes actions such as spinning the robot in order to collect information about the surroundings and relocalise.

It is unlikely the navigation action will catastrophically fail within these constrained environments, so any failure is handled as a total task failure.

5.3.3.5 Text-to-speech

Text-to-speech is also relatively straightforward as this is already provided by the robot, we simply have to send a goal to the `/tts` action server with our desired text and wait for completion.

5.3.3.6 Running for two guests

To run a set of states twice, we use an Iterator container from SMACH. It is composed of two parts, the iterator itself which holds a Python iterator and provides the current value through a chosen key in the userdata and an inner container in which we build our desired states into.

To implement this state, we create an iterator that runs over the range $1 \leq i < 3$, we ask that the current value of i is provided through the `index` key on the userdata, and we build the substate machine as in Figure 19 in the container. Once the iterator is exhausted, we transition to the next phase and go to the seating area.

5.3.3.7 Waiting for a person

This is a relatively straightforward state that first waits for the next PointCloud information from the robot, sends it to the object detection `/detect3d` service discussed earlier, filters the results by those whose points are within the predefined waiting area, and finally has a successful outcome if it finds someone or a failure outcome if it does not.

Figure 21: State machine for waiting for a new person

In order to filter for detections that are in the area, we use the shapely Python library [92] which provides algorithms for polygon and point intersections. We load the list of points for the waiting area and convert it to a shapely Polygon. Then we take the x,y components of the objects' points, convert them to shapely Points, and call `Polygon.contains(point)` for each.

This state lives in the lasr_skills package [37].

5.3.4 SM: Meeting the guest

To run multiple state machines at once, we use a Concurrence container from SMACH. Once all the sub-state machines have reached the success outcome, the concurrence as a whole will succeed. It may likewise fail and preempt all other states.

For the purposes of this task, we want to indefinitely collect face reidentification data until we finish generating a description and talking to the guest. To achieve this, we must provide a mechanism for preempting the infinite loop and allowing the state machine to finish. SMACH provides a way to add a callback which determines whether the state machine should be preempted based on the termination of its children. We implement the following:

Algorithm 9: Child Termination Callback

input: states: $str \rightarrow str$
output: terminate: *bool*

```
1 terminate ←⊥                          ▷ allow states to continue running by default
2 if states.DESC.succeeded and states.TALK.succeeded then
3     terminate ← ⊤                     ▷ signal that we want to preempt all other states
4 end
```

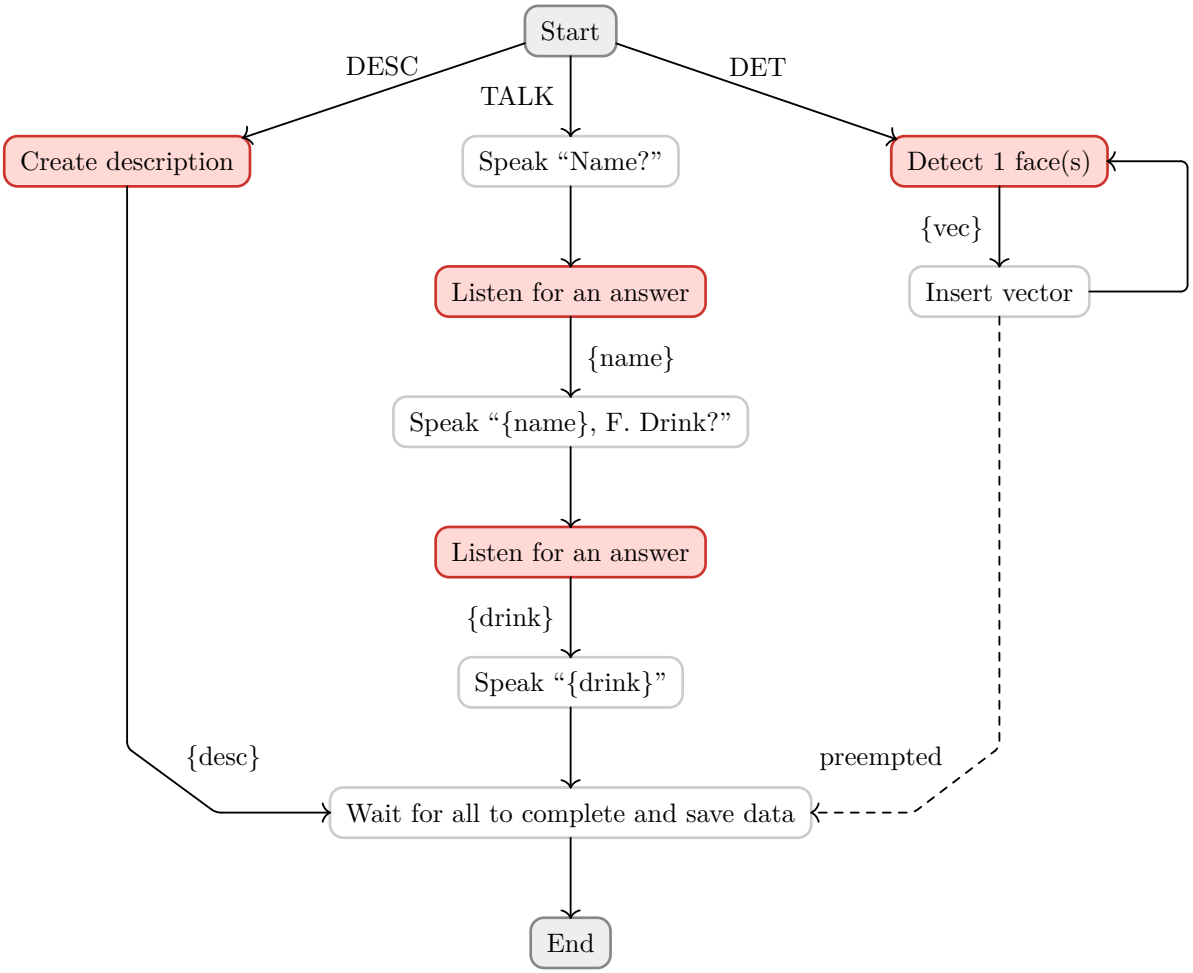


Figure 22: State machine for meeting and learning about a new guest

5.3.4.1 Facilitating preemption

The loop we've created will not automatically understand when it must break out and complete execution. So we introduce a new state, we call it the PreemptionHandler, which has two outcomes: continue or preempt. It is implemented as follows:

Algorithm 10: PreemptionHandler State

output: outcome: *str*

```
1 if self.preempt_requested() then          ▷ ask SMACH if we need to preempt
2     self.service_preempt()                ▷ notify SMACH that the preemption has been handled
3     outcome ← "preempted"                 ▷ notify our code that we must stop execution of this state
4 else
5     outcome ← "continue"
6 end
```

5.3.4.2 Detecting faces

This is also quite a simple state: we fetch the next image from the camera, send it to the DeepFace service to find all of the faces in the image, and finally we may optionally check the count and fail if it is not what we expected. The check count state simply asserts that the length of `faces` is correct.

Figure 23: State machine for detecting faces

5.3.4.3 Inserting vectors

This state takes the index and vector from userdata and uploads it to the vector database using the `/database/vectors/weaviate/insert` service with the following format:

Collection Name	Properties	Vector
"Face"	`{"guest": userdata.index}`	userdata.vector

5.3.4.4 Creating a description of the guest

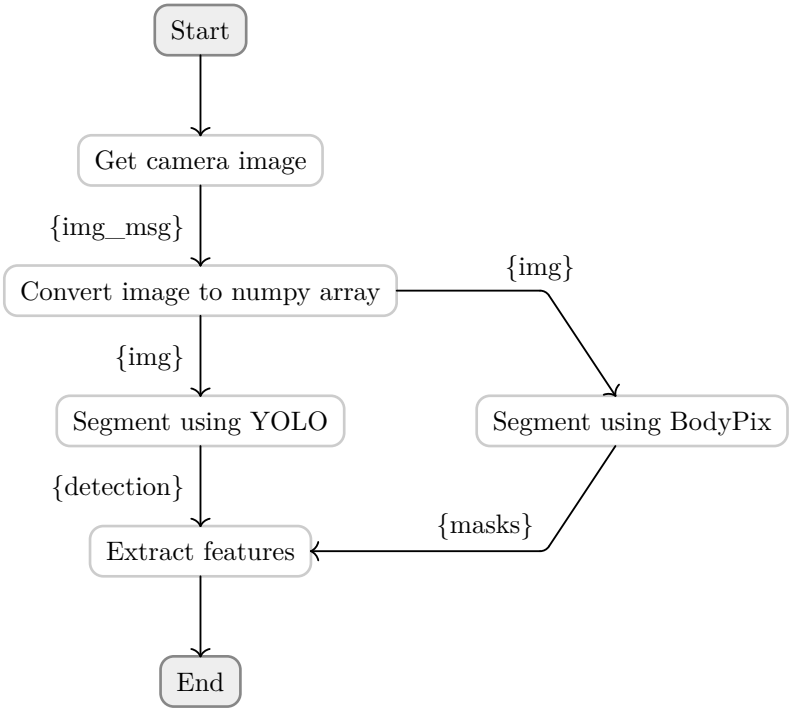


Figure 24: State machine for generating a description of the guest

This state takes the latest image from the robot's camera and segments it using both object detection and body segmentation services. For the object segmentation service, we request only people so we receive back a list of segmentation masks for each person in the scene. For the body segmentation service, we request the torso and face.

Next, for each person, we look at each body part mask and either extract the colour for the torso or perform additional inference using the face features service for the face mask. We need to also segment each body part mask using the person mask to avoid processing the wrong part of the image which isn't relevant to the person we want. In essence, it works as follows:

Algorithm 11: Extract Features State

input: detection: *YoloDetection[]*, masks: *BodyPixMask[]*, image: *numpy.array*
output: people: *YoloDetection* → *FeatureWithColour[]*
requires: numpy: *numpy [93]*, cv2: *OpenCV2 [94]*, colours: *colour_estimation*

```
1  people ← []
2  height, width, channels ← image.shape          ▷ determine the size of the image
3  for person in detection do
4      mask_image ← numpy.zeros((height, width))          ▷ create an empty image
5      cv2.fillPolygon(mask_image, person.xyseg, color=white)  ▷ fill in the outline of person
6      mask_bin ← mask_image > 128      ▷ create a binary mask using 50% brightness as threshold
7      features ← []
8      for index, part_mask in masks do
9          part_mask[mask_bin == 0] ← 0      ▷ remove mask areas that belong to other people
```

```
10      if index is 0 then torso mask
11          median ← numpy.median(image[part_mask == 1])
12          features ← FeatureWithColour("torso", colours.nearest_name(median))
13      else head mask
14          features ← result from face feature service
15      end
16    end
17  end
```

5.3.4.5 Listening for answers from the guest

In this state, we use the Whisper transcription action server to wait for a new phrase being spoken by the user, this is sent to RASA to figure out the intent of what the user is saying, and then we can try to interpret the given intent with the detected phrase entities. If we are in simulation, we instead simulate a waiting period of a few seconds and then pass example speech through userdata. If we fail to understand what the user says, we tell them to try again.

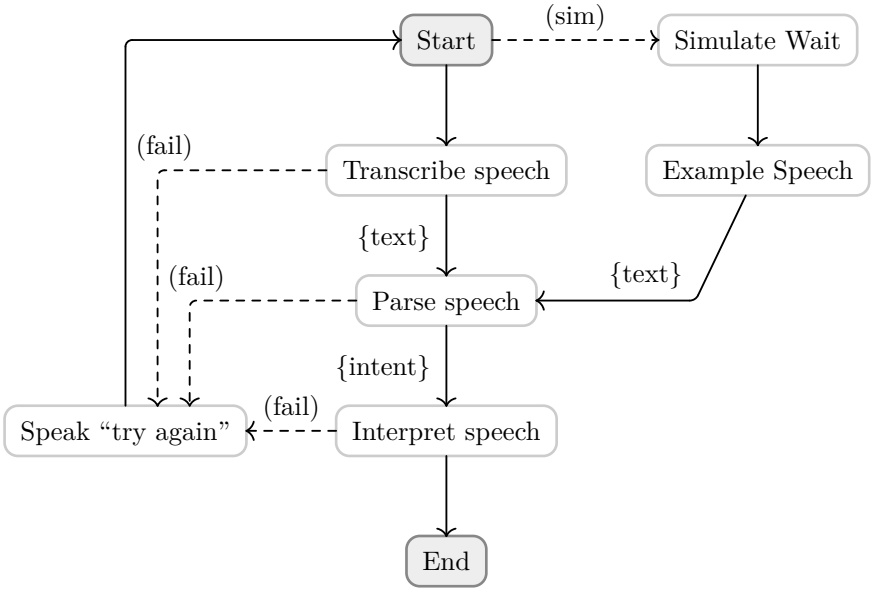


Figure 25: State machine for listening for a guest's spoken answer

Here's an example of how the intent is interpreted:

Algorithm 12: Interpret Intent (for person name)

input: intent: *JSON Object*
output: name: *str*

```
1  assert intent['intent']['name'] == "person_name"   ▷ ensure we are looking at correct intent
2  assert "name" in intent['intent']                  ▷ ensure a name has been detected by RASA
3  assert length intent['entities']['name'] is 1      ▷ expect there to only be one name
4  name ← intent['entities']['name'][0]
```

5.3.5 SM: Seating the guest

This state consists of previously defined states with one new sub-state machine.

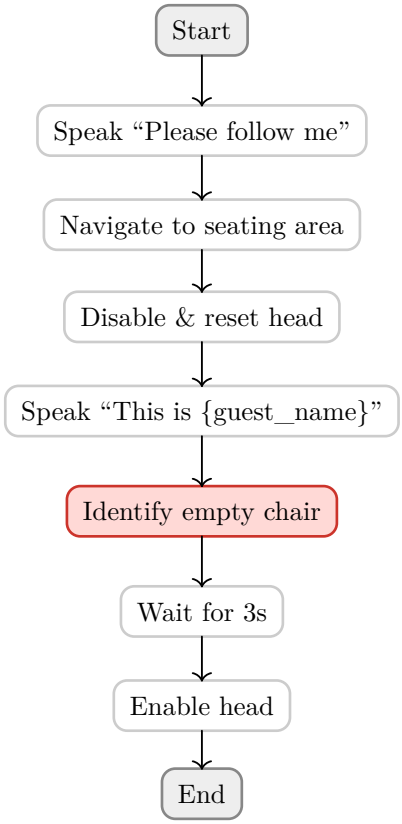


Figure 26: State machine for seating a guest

To identify a chair, we segment out all the chairs and people in the scene and find those that have nobody overlapping them. Any failure results in the robot asking guest to sit anywhere.

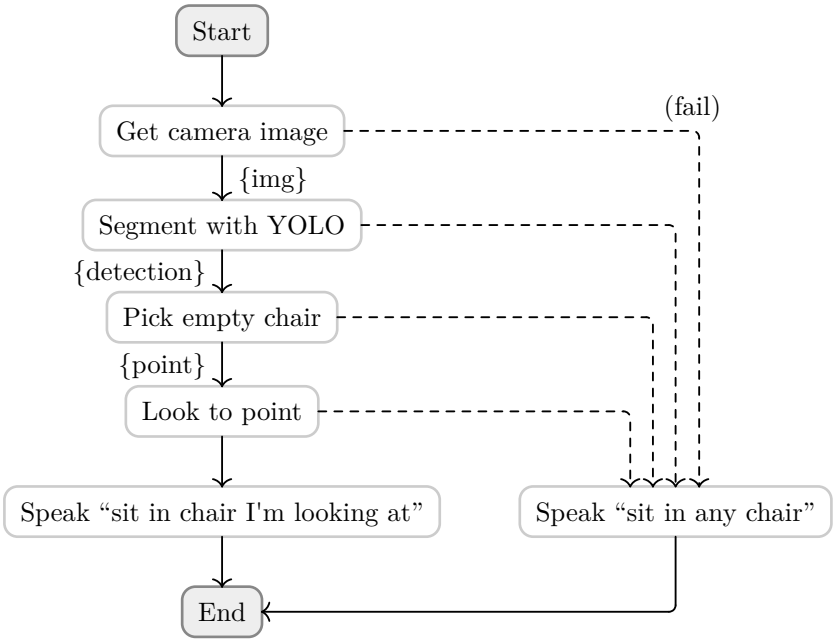


Figure 27: State machine for identifying an empty chair

Algorithm 13: Pick Empty Chair

input: detection: *YoloDetection[]*
output: point: *geometry_msgs/Point | None*
requires: shapely: *Shapely [92]*

```
people ← []
for object in detection do
    if object.name is "person" then        ▷ generate polygon data structures for all people in frame
        people ↞ shapely.Polygon(object.xyseg)
    end
end
for object in detection do
    if object.name is "chair" then
        polygon ← shapely.Polgyon(object.xyseg)
        intersects ←⊥
        for person in people do
            if person.intersects(polygon) then
                intersects ← ⊤              ▷ someone is sitting in the chair, so ignore it
            end
        end
        if intersects is ⊥ then
            point ← object.point           ▷ pick this chair's centroid as the point to look at
        end
    end
end
```

5.3.5.1 Look to point

Look to point is a state provided by the lasr_skills package [37]. It is a primitive wrapper around the action server `/head_controller/point_head_action` which accepts a frame to compute relative to and a target to look at. The goal is defined as so: (with sample values)

Data Structure 26: Point Head Action Implementation

PointHeadGoal.msg	Values
- pointing_frame: **string**	head_2_link
- pointing_axis: **geometry_msgs/Point**	[1.0, 0.0, 0.0]
- max_velocity: **float**	1.0
- target.header: **std_msgs/Header**	frame_id=map
- target.point: **geometry_msgs/Point**	userdata.point

5.3.6 SM: Introducing guests

This is the final phase of the task, during which we find where everyone is and then describe the first guest to the second guest by turning to look at where the second guest is and recalling the description generated for the first guest earlier.

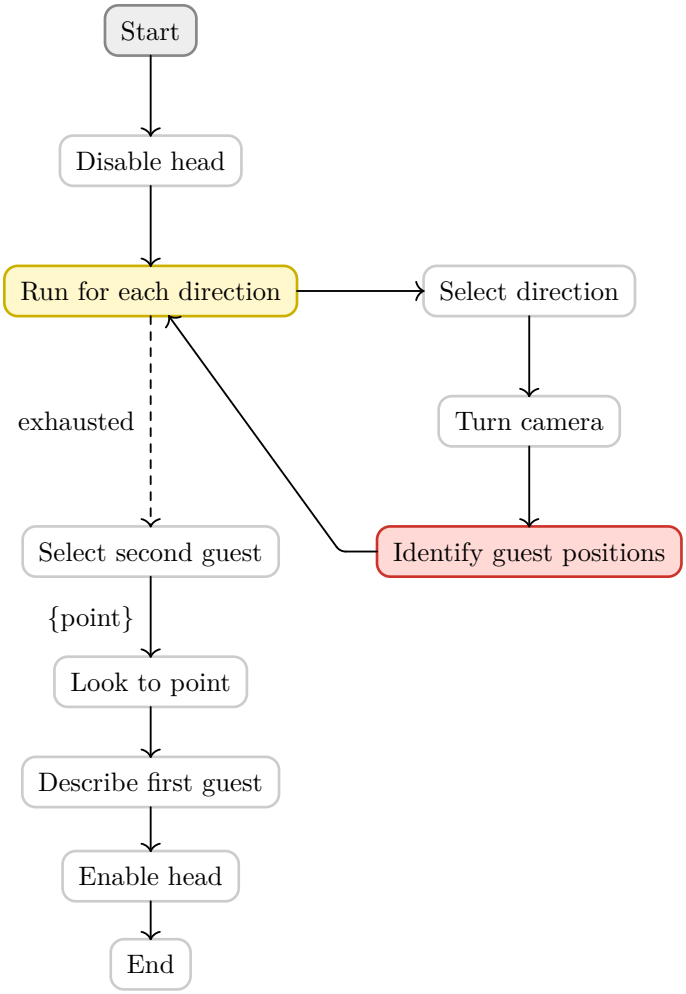


Figure 28: State machine for introducing guests to each other

5.3.6.1 Turning the camera

We use the play motion action server to send a goal to look left, right, and then to the centre. (we sweep across from left to right to then avoid having to correct from a side to the centre and add complexity) Each direction also overlaps, which means guests should typically be scanned at least twice if they're in front of the robot and allows people sitting to the side of the robot to have a more accurate front-facing scan of their face which increases our accuracy.

5.3.6.2 Identifying guest positions

This state detects people in 3D much like we did earlier when finding empty chairs but also finds the faces present in the frame. The face vector is queried against the vector database using the `/database/vectors/weaviate/query` service. Each item in the list of results is a potential match with the confidence and properties (which includes the guest index). Since this state runs multiple times, this allows us to pick the highest confidence results for each guest.

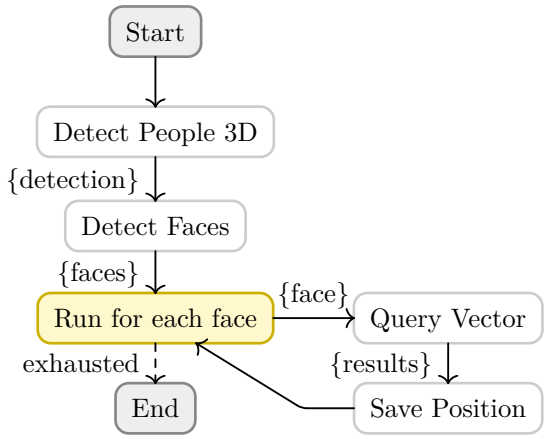


Figure 29: State machine for identifying guest positions

Algorithm 14: Save Guest Position

input: face: *Embedding*, detection: *YoloDetection[]*, result: *SearchResult[]*
requires: shapely: *Shapely [92]*, state: *EphemeralState*

```
face_area ← shapely.Polygon([ [face.x, face.y], [face.x + face.w, face.y], [face.x + face.w, face.y + face.h], [face.x, face.y + face.h] ])   ▷ convert xywh to polygon
result ← results[0]   ▷ results is sorted by confidence as it comes from database
confidence ← result.confidence
for person in detection do
    polygon ← shapely.Polgyon(person.xyseg)
    if polygon.intersects(face_area) then   ▷ this face corresponds to this person
        current_confidence ← state.get_guest_point_confidence(result.index)
        if confidence > current_confidence then
            state.set_guest_point(person.point)   ▷ take this as the new assumed position of the guest
            state.set_guest_point_confidence(confidence)   ▷ update best confidence
```

5.3.6.3 Look at the second guest

In this state, we re-use Section 5.3.5.1, but instead of simply passing the position of the detected person, we first update the z component of the point to be 1.0 which means the robot will look roughly eyelevel for someone sitting down on a chair.

5.3.6.4 Describe the first guest

We use the ephemeral state to populate information about the guest into the following:

Hi guest 1, guest 2 is here with you, they are wearing a torso colour top, they have hair colour hair, they are / are not wearing glasses, and they are / are not wearing a hat.

Figure 30: String Interpolation

5.4 Approaches to Testing

This project follows a concise testing methodology, functional requirements (Section 4.1.1) are proved through running and recording the results of the task on the real robot and in simulation, while non-functional requirements (with the exception of N1, N2) are qualitatively assessed using specially designed experiments that measure how well a certain aspect performs.

5.4.1 Testing the whole task

Running the task follows a simple procedure (which will be detailed more thoroughly in Appendix Section A). We must first start all of the required services for the intended configuration. In simulation, we omit loading the transcription server (responses are mocked) and load a mock text-to-speech server. Once it is confirmed they are all running, we may load or create the appropriate configuration for the arena, and finally run the task.

To be able to later assess the results: The outputs from all relevant topics are saved using the rosbag utility (this includes inference results, text-to-speech goals, and camera information). The output of the task is logged to a local file. A screen cast is taken of the machine running the task. And if running on the robot, a video recording is taken of the physical space.

5.4.2 Experiments

The project includes three 'scripts' (Python modules that either directly interface with various services to emulate parts of the task or invoke parts of the state machine we've built) that provide data to qualitatively assess requirements N3 through N5 (Section 4.1.2).

Unit 02 addresses requirement N3 "robot can reliably detect and reidentify faces": this script creates three collections in the vector database into which we insert 1, 2, or 5 faces for each subject we involve from which to learn from. We can then use the remainder of the face images for each subject to determine whether they can be reidentified and with what confidence.
Dataset collected: 5 subjects with 15 images each

Unit 03 addresses requirement N4 "reliably recognise speech and intent": this script reads in a given set of audio clips with their associated spoken phrases transcribed by hand and feeds them into various speech-to-text models available within this project. The resulting phrases are fed into the custom RASA assistant to see if the correct intent and information can be extracted. We can calculate the success rates for both components.
Dataset collected: 112 spoken phrases, 50 of which contain background noise representing a busy coffee shop, 12 of which are phrases that aren't included in the training set for the assistant.

Unit 04 addresses requirement N5 "reliably generate selected characteristics": this script takes in a given folder of pictures of people with associated metadata (known torso colour, hair colour, hat presence, glasses presence) and executes the describe guest (Figure 24) state machine. The results can be compared to the actual known information to calculate the success rate of each feature extraction.
Dataset collected: 30 unique images of people with associated metadata; resized to 640px

For completeness, Unit 01 was a script used for testing face reidentification during development but it is not an appropriate source of data.

6 Evaluation

6.1 Experimental Results

6.1.1 Findings from full task runs

The following findings are proven by the runs on the actual robot and in simulation that are included in the supplementary presentation video.

The robot successfully navigates between predefined points in the arena (T1) without failure. The robot can identify when a person is or is not in the waiting area and does not move preemptively even if a person is visible in the background (T2). The robot can generate a description of the guest and recite these features during the final stage of the task (T3). The robot can engage in conversation with a newly arrived guest and learn about their name and favourite drink, it recites these back to the guest as confirmation (T4). The robot is able to learn the guest's face features and later looks at the correct guest during the final phase (T5). The robot directs guests to the seating area and then looks towards an empty chair to indicate they should sit in it (T6).

The robot also maintains gaze with the person they talk to during setup by virtue of the guest always being in front of the robot, and turns to look at the person they intend to talk to during the final phase (N1). Finally, the robot always looks in the direction of travel except when in simulation due to limitations with the simulated TIAGo (N2).

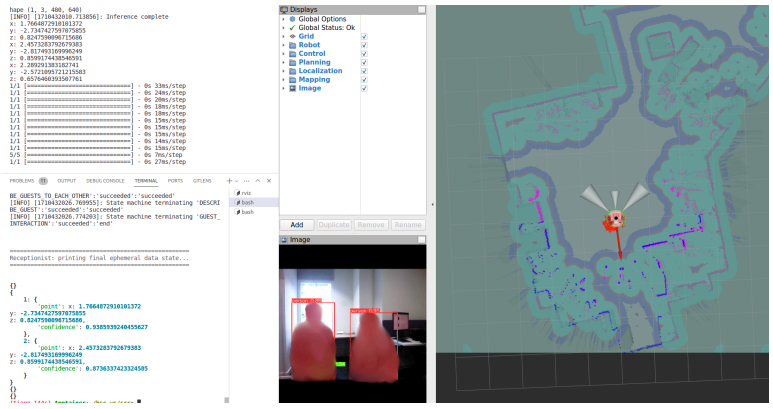


Figure 31: Robot correctly reidentifies a guest and logs their position

6.1.2 Assessing face reidentification

From the collected experimental data, the mode of the face detection confidence is 99.9%, with lows of 99.7%. However, we find that in some circumstances it is challenging to detect faces when the image quality is poor as seen with subject C.

Table 1: Face Detection Results

Subject	Total	Detected
A	15	100%
B	15	100%
C	15	53%
D	15	93%
E	15	100%

In general, the success rate of reidentification is almost always 100%. In the context of the task, we can confidently say that at least 5 images will be collected on average which should certainly guarantee that we reliably reidentify guests.

Table 2: Face Reidentification Results

	Success Rate			**Min. Confidence**			**Avg. Confidence**		
Subject	1:14	2:13	5:10	1:14	2:13	5:10	1:14	2:13	5:10
A	0.93	0.62	1	0.67	0.68	0.77	0.72	0.74	0.8
B	1	1	1	0.64	0.67	0.67	0.77	0.77	0.78
B	1	1	1	0.68	0.77	0.82	0.78	0.81	0.84
C	1	1	1	0.6	0.67	0.67	0.75	0.78	0.79
D	1	1	1	0.66	0.66	0.75	0.75	0.77	0.79

6.1.3 Assessing speech recognition

In general, the results show that chance of failure isn't astoundingly high and given that we stick to the medium-sized model, we can assume that we are unlikely to mishear someone.

Table 3: Transcription Success Rate

		Clean Audio			**Busy Coffee Shop**		
Phrase	**Samples**	tiny	small	medium	tiny	small	medium
"i like to drink cola"	5	0	1	1	0	0.6	1
"i'm robin"	5	1	1	1	1	1	1
"it's finn"	5	0.6	0.4	1	0.2	0.6	0.8
"it's me, mason"	5	0.8	1	1	0.4	1	1
"jane"	5	1	1	1	1	1	1
"milk"	5	1	1	1	1	1	1
"my favourite drink is iced tea"	5	1	1	1	1	1	1
"my name is charlie"	5	0.6	1	0.6	0.8	1	1
"orange juice"	5	1	1	1	1	1	1
"tropical juice"	5	0.6	1	1	0.6	1	1
"i like to drink hot chocolate"	3	1	1	1			
"i'm izzy"	3	1	1	1			
"my favourite drink is lemonade"	3	0.67	1	1			
"my name is adam"	3	1	1	1			
Total	112						

The tiny-sized model was surprisingly good, but it often hallucinated phrases or was just completed wrong. For example, it could interpret "It's finn" as "it's been" or "it's fun", "cola" as "color", "lemonade" as "laminate", and so on. The worst hallucinations seen were "tropical juice" as "drop good juice", "trump goodness", and "trump could just".

All of these issues are resolved in the larger models. However, the medium-sized model still struggled with recognising the name “Finn”, with this issue being less prevalant in medium. The only outlier in medium was interpreting “charlie” as “chani”.

In terms of then recognising the intent, the assistant was always able to recognise what we meant when the phrases generated by the transcription were correct. And even then, it was more often right in picking the correct intent and picking out the information.

Table 4: RASA Intent Recognition Results

		Correct Inferences			
		Valid Phrases		Invalid Phrases	
Total Phrases	**Valid Phrases**	**Intents**	**Entities**	**Intents**	**Entities**
48	38	100%	100%	70%	60%

6.1.4 Assessing guest feature extraction

This was the least reliable component of the project as seen in the results below.

Table 5: Person Description Results

Correct Inferences			
Torso Colour	**Hair Colour**	**Glasses Presence**	**Hat Presence**
53%	60%	80%	90%

This can be put down to a combination of factors involving poor image quality (causing colours to tend to extremes), poor colour estimation (the code took the mean RGB value and compared it to the closest known value, this is relatively primitive and likely why colours are often wrong), and unrefined feature segmentation model. This will not be considered to satisfy its respective requirement due to the poor overall performance.

6.2 Summary

All of the raw data used to assess the experiments is available in Appendix Section C.

Table 6: List of project requirements

Requirement	**Satisfied**
T1 Able to navigate to preconfigured beacons in arena	R,S
T2 Wait for a person to appear and approach them	R,S
T3 Generate a description of a guest including at least 4 characteristics	R,S
T4 Engage in conversation with a guest asking for name & favourite drink	R,S
T5 Learn a guest's face features and then later reidentify them	R,S
T6 Direct guests to sit in empty seats in the waiting area	R,S
N1 Robot should maintain gaze with any person it talks to	R,S
N2 Robot should look in the direction of travel when moving around	R
N3 Robot can reliably detect and reidentify faces	E
N4 Robot can reliably recognise speech and intent	E
N5 Robot can reliably generate the selected characteristics for a guest	

Requirements labeled with **R**, **S**, **E** are satisfied by evidence of the behaviour being observed on the actual robot, n imulation, or in experimentation respectively.

7 Legal, Social, Ethical, and Professional Issues

7.1 Legal Issues

This project requires the processing of personal data which is subject to data protection laws, specifically it handles personally identifiable information such as names, face feature vectors, and a textual description of the person's appearance. This PII is associated with non-PII such as their favourite drink and their position within the arena at the time of phase 3.

In order to legally process this data, we must achieve reasonable consent, we must make the user aware of their data being collected, how long it is being collected for, and their rights. For the purposes of this project, we will refer to the General Data Protection Regulation [95] which is implemented in the UK under the Data Protection Act 2018 [96].

Reasonable consent and legal purpose for processing this PII is gathered when the user consents to participate in the task after (in accordance with Art. 15 [97]):

- They are notified that facial recognition is in use during the execution of the project, their facial information is only stored from the beginning to when the task is next invoked or the database is manually cleared during clean-up, whichever comes first. With the exception of the host, whose information is temporarily stored between invocations until the robot is turned off or a new host is enrolled.
- They are notified that any information they provide or is gathered from their appearance is only stored for the duration of the task.
- They are made aware of their right to be forgotten (Art. 17 [98]) and withdraw consent at any time (Art 18. [99]) by talking to the operator of the task.
- They are made aware of who the data controller is, this is usually the operator of the task. Their responsibility is to ensure this regulation is followed adequately (Art. 24 [100]).

The operator of the task should follow the guidance provided in the user manual to ensure they operate within this legislation, see Appendix Section A.

There are no specific laws in the UK with regards to facial recognition [101]. The ICO points to responsibilities set out in the DPA 2015.

7.2 Health & Safety

There are many considerations for health & safety when dealing with the robot, for this task, you should use the stock TIAGo configuration, i.e. you should not mess with the default robot acceleration and cost layer inflation. This will ensure the robot is operating within its specifications.

This task involves no arm movement as it has been deemed too risky. In the context of this project, we instead choose to look in a particular direction in order to indicate what we are referring to. This task involves no changes to the map cost layers. Here the laser is used for obstacle avoidance and is rather accurate so collisions are not a risk. This task also involves no changes to the maximum robot speed or acceleration so we are unlikely to overshoot and collide with any objects.

The operator is advised to learn the TIAGo's safety features, specifically the ability to emergency stop the robot if it is putting someone else in danger or it is at risk of serious damage if it continues to operate, this is achieved by pressing the E-STOP button which kills all motors, but may damage the arms if they are actively moving (not applicable for our task). Other

safety features include the ability to instantly freeze the robot using a special combination on the controller, this is detailed in the handbook for TIAGo.

Finally, the operator should take care to ensure someone is always present with the robot who understands the safety rocedures, and they must never leave the robot unattended while charging or risk a potential fire.

7.3 Ethics & Professional Issues

This project involves the use of various "AI models" which are trained on a variety of datasets, due diligence should be taken to ensure that the datasets being used for these models are ethically sourced, specifically images and audio snippets must be collected with appropriate consent from the authors.

This project involves the recognition of features of different people as well as full-body person detection. Care should be taken to ensure the datasets used to train the models in use on the project are appropriately diverse and would not fail to recognise people of various origin.

This project may be used by any number of different groups of people so any human-robot interactions should aim to be inoffensive to a broad number of cultures, this would involve avoiding slang, certain phrases, or gestures which may be appropriate in our culture but inappropriate in another.

With further work, this project may have the potential to replace a human hospitality worker, care should be taken by those wishing to deploy this or derivative solutions to ensure any replaced workers are able to continue work in another field or otherwise are provided with an appropriate safety net if they must find another job.

This project is implemented in accordance with the British Computing Society's Code of Conduct [102], specifically, it has been developed with "due regard for public health, privacy, security and wellbeing of others and the environment", does not violate any legitimate rights of Third Parties, conducts due diligence to avoid discriminating against any one person, and aims to promote equal access by the virtue of the project being available publicly under the MIT license [103].

8 Conclusion

This project has involved the development of a complex state machine that can perform the role of a robot receptionist, composed of three distinct phases in which we greet new guests, guide them to the lobby, and later come back to talk to them. It has involved the implementation of various object detection, body segmentation, speech recognition, dialog intent recognition, and custom-trained feature recognition models. This project demonstrates that, with the exception of the face feature model, the technology required to implement robots that interact in an natural manner is available and robust.

To a greater extent, this project has met the criteria set out by its requirements. It misses the mark in its ability to describe people and their textual facial features, which arguably can be considered a project in itself. As such, I think the project has been successful.

Some takeaways and observations from the development of the project:

- Following the development of and implementing state-of-the-art models in your project is the least path of resistance to a robust foundation for any sort of complex behaviour. Even if the data being used with these models is sub-par.
- The development and training of custom vision models is a great enough commitment that it doesn't make sense in the timeframe of this project, as such that component of the project has suffered accordingly. More research should have been done here into other more robust solutions for extracting textual facial features and colours.
- SMACH is an excellent choice as a modular platform to implement isolated behaviours. This proved immensely useful for testing and for being able to quickly iterate on how different behaviours link together.

8.1 Future Work

In the near future, this project may be repurposed by generalising and merging its utility states and supporting state machines into the LASR repository, it may also be merged back in preparation for the coming RoboCup@Home competition.

This project is in a state that anyone may adapt it to work in any number of environments that require dynamic interactions with humans.

Appendix omitted, provided in separate PDF.

Keep scrolling for references.

References

[1] K. Kaufmann *et al.*, "Social Robots: Development and Evaluation of a Human-Centered Application Scenario," in *Human Interaction and Emerging Technologies*, T. Ahram, R. Taiar, S. Colson, and A. Choplin, Eds., Cham: Springer International Publishing, 2020, pp. 3–9.

[2] J. Nakanishi, I. Kuramoto, J. Baba, K. Ogawa, Y. Yoshikawa, and H. Ishiguro, "Continuous Hospitality with Social Robots at a hotel," *SN Applied Sciences*, vol. 2, no. 3, p. 452–453, Feb. 2020, doi: 10.1007/s42452-020-2192-7.

[3] "Market Insights on Service Robotics - Worldwide." [Online]. Available: https://www.statista.com/outlook/tmo/robotics/service-robotics/worldwide

[4] K. Price, "Slim Chickens Guildford starts trialling...." [Online]. Available: https://www.thecaterer.com/news/boparan-restaurant-group-slim-chickens-trialling-service-robots

[5] J. Rear, "Are robo-waiters really the future of restaurants?." [Online]. Available: https://www.telegraph.co.uk/food-and-drink/features/robo-waiters-really-future-restaurants/

[6] [Online]. Available: https://group.hennnahotel.com/

[7] [Online]. Available: https://www.theguardian.com/world/2015/jul/16/japans-robot-hotel-a-dinosaur-at-reception-a-machine-for-room-service

[8] "RoboCup Scientific Papers." [Online]. Available: https://www.robocup.org/research/search

[9] J. Hart *et al.*, "RoboCup@Home 2024: Rules and Regulations." [Online]. Available: https://github.com/RoboCupAtHome/RuleBook/actions/runs/6710524336

[10] J. Pages, L. Marchionni, and F. Ferro, "TIAGo: the modular robot that adapts to different research needs." [Online]. Available: https://pal-robotics.com/wp-content/uploads/2020/08/TIAGo_-the-modular-robot-that-adapts-to-different-research-needs.pdf

[11] "Nodes - ROS Wiki." [Online]. Available: http://wiki.ros.org/Nodes

[12] "rospy." [Online]. Available: http://wiki.ros.org/rospy

[13] "std_msgs/Header." [Online]. Available: https://docs.ros.org/en/noetic/api/std_msgs/html/msg/Header.html

[14] "geometry_msgs/Point." [Online]. Available: https://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Point.html

[15] "geometry_msgs/Pose." [Online]. Available: https://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Pose.html

[16] "Topics - ROS Wiki." [Online]. Available: http://wiki.ros.org/Topics

[17] "Services - ROS Wiki." [Online]. Available: http://wiki.ros.org/Services

[18] "Parameter Server - ROS Wiki." [Online]. Available: http://wiki.ros.org/Parameter%20Server

[19] "actionlib - ROS Wiki." [Online]. Available: http://wiki.ros.org/actionlib

[20] "tf2 - ROS Wiki." [Online]. Available: http://wiki.ros.org/tf2

[21] "smach - ROS Wiki." [Online]. Available: http://wiki.ros.org/smach

[22] "smach_ros - ROS Wiki." [Online]. Available: http://wiki.ros.org/smach_ros

[23] "rviz - ROS Wiki." [Online]. Available: http://wiki.ros.org/rviz

[24] "Gazebo." [Online]. Available: https://gazebosim.org/home

[25] "rosbag - ROS Wiki." [Online]. Available: http://wiki.ros.org/rosbag

[26] "Foxglove." [Online]. Available: https://foxglove.dev/

[27] "roslaunch - ROS Wiki." [Online]. Available: http://wiki.ros.org/roslaunch

[28] G. Jocher, A. Chaurasia, and J. Qiu, "YOLO by Ultralytics." [Online]. Available: https://github.com/ultralytics/ultralytics

[29] C.-Y. Wang and H.-Y. M. Liao, "YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information," 2024.

[30] "deepface." [Online]. Available: https://github.com/serengil/deepface

[31] "tfjs-models/body-pix." [Online]. Available: https://github.com/tensorflow/tfjs-models/tree/master/body-pix

[32] "python-tf-bodypix." [Online]. Available: https://github.com/de-code/python-tf-bodypix

[33] "Weaviate - Vector Database." [Online]. Available: https://weaviate.io/

[34] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust Speech Recognition via Large-Scale Weak Supervision." [Online]. Available: https://arxiv.org/abs/2212.04356

[35] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open Source Language Understanding and Dialogue Management." 2017.

[36] "LASR." [Online]. Available: https://www.sensiblerobotsresearch.org/lasr/

[37] "Base/skills." [Online]. Available: https://github.com/LASR-at-Home/Base/tree/main/skills

[38] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A Review of Yolo Algorithm Developments," *Procedia Computer Science*, vol. 199, pp. 1066–1073, 2022, doi: https://doi.org/10.1016/j.procs.2022.01.135.

[39] W. Zhiqiang and L. Jun, "A review of object detection based on convolutional neural network," in *2017 36th Chinese Control Conference (CCC)*, 2017, pp. 11104–11109. doi: 10.23919/ChiCC.2017.8029130.

[40] C. Li *et al.*, "YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications." 2022.

[41] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2023, pp. 7464–7475.

[42] "COCO - Detection Evaluation." [Online]. Available: https://cocodataset.org/#detection-eval

[43] "yolo-comparison-plots.png." [Online]. Available: https://raw.githubusercontent.com/ultralytics/assets/main/yolov8/yolo-comparison-plots.png

[44] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," *CoRR*, 2014, [Online]. Available: http://arxiv.org/abs/1405.0312

[45] O. Mubin, M. I. Ahmad, S. Kaur, W. Shi, and A. Khan, "Social Robots in Public Spaces: A Meta-review," in *Social Robotics*, S. S. Ge, J.-J. Cabibihan, M. A. Salichs, E. Broadbent, H. He, A. R. Wagner, and Á. Castro-González, Eds., Cham: Springer International Publishing, 2018, pp. 213–220.

[46] S. Sabanovic, M. Michalowski, and R. Simmons, "Robots in the wild: observing human-robot social interaction outside the lab," in *9th IEEE International Workshop on Advanced Motion Control, 2006.*, 2006, pp. 596–601. doi: 10.1109/AMC.2006.1631758.

[47] M. K. Lee and M. Makatchev, "How Do People Talk with a Robot? An Analysis of Human-Robot Dialogues in the Real World," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, in CHI EA '09. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 3769–3774. doi: 10.1145/1520340.1520569.

[48] C. Nakajima, M. Pontil, B. Heisele, and T. Poggio, "Full-body person recognition system," *Pattern Recognition*, vol. 36, no. 9, pp. 1997–2006, 2003, doi: https://doi.org/10.1016/S0031-3203(03)00061-X.

[49] C.-F. Juang, C.-M. Chang, J.-R. Wu, and D. Lee, "Computer Vision-Based Human Body Segmentation and Posture Estimation," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 39, no. 1, pp. 119–133, 2009, doi: 10.1109/TSMCA.2009.2008397.

[50] G. Papandreou *et al.*, "Towards Accurate Multi-person Pose Estimation in the Wild." 2017.

[51] G. Papandreou, T. Zhu, L.-C. Chen, S. Gidaris, J. Tompson, and K. Murphy, "PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model." 2018.

[52] "bodypix-google-blog." [Online]. Available: https://blog.tensorflow.org/2019/11/updated-bodypix-2.html

[53] Mozilla, "Mozilla/DeepSpeech: DeepSpeech is an open source embedded (offline, on-device) speech-to-text engine which can run in real time on devices ranging from a Raspberry Pi 4 to high power GPU servers.." [Online]. Available: https://github.com/mozilla/DeepSpeech

[54] A. Hannun *et al.*, "Deep Speech: Scaling up end-to-end speech recognition." 2014.

[55] "vosk." [Online]. Available: https://alphacephei.com/vosk/

[56] Alphacep, "Alphacep/Ros-Vosk: VOSK node for Ros Robot Operating System." [Online]. Available: https://github.com/alphacep/ros-vosk

[57] N. Shmyrev, "OpenAI whisper accuracy and other recent models (Nemo Transducer XLarge, Gigaspeech)." [Online]. Available: https://alphacephei.com/nsh/2022/10/22/whisper.html

[58] G. W. J. D. W.-Q. Z. C. W. D. S. D. P. J. T. J. Z. M. J. S. K. S. W. S. Z. W. Z. X. L. X. Y. Y. W. Y. W. Z. Y. Z. Y. Guoguo Chen Shuzhou Chai, "GigaSpeech: An Evolving, Multi-domain ASR Corpus with 10,000 Hours of Transcribed Audio," in *Proc. Interspeech 2021*, 2021.

[59] "NVidia Conformer-Transducer X-Large." [Online]. Available: https://huggingface.co/nvidia/stt_en_conformer_transducer_xlarge

[60] N. Sabharwal and A. Agrawal, "Introduction to Google Dialogflow," in *Cognitive Virtual Assistants Using Google Dialogflow: Develop Complex Cognitive Bots Using the Google Dialogflow Platform*, Berkeley, CA: Apress, 2020, pp. 13–54. doi: 10.1007/978-1-4842-5741-8_2.

[61] A. F. Muhammad, D. Susanto, A. Alimudin, F. Adila, M. H. Assidiqi, and S. Nabhan, "Developing English Conversation Chatbot Using Dialogflow," in *2020 International Electronics Symposium (IES)*, 2020, pp. 468–475. doi: 10.1109/IES50839.2020.9231659.

[62] "DialogFlow." [Online]. Available: https://cloud.google.com/dialogflow

[63] "PyPI - RASA." [Online]. Available: https://pypi.org/project/rasa/

[64] "RASA Command Line Interface." [Online]. Available: https://rasa.com/docs/rasa/command-line-interface/

[65] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the Gap to Human-Level Performance in Face Verification," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708. doi: 10.1109/CVPR.2014.220.

[66] G. B. Huang, M. Mattar, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database forstudying face recognition in unconstrained environments," in *Workshop on faces in'Real-Life'Images: detection, alignment, and recognition*, 2008.

[67] A. Firmansyah, T. F. Kusumasari, and E. N. Alam, "Comparison of Face Recognition Accuracy of ArcFace, Facenet and Facenet512 Models on Deepface Framework," in *2023 International Conference on Computer Science, Information Technology and Engineering (ICCoSITE)*, 2023, pp. 535–539. doi: 10.1109/ICCoSITE57641.2023.10127799.

[68] "Visual Geometry Group - University of Oxford." [Online]. Available: https://www.robots.ox.ac.uk/~vgg/software/vgg_face/

[69] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep Face Recognition," in *British Machine Vision Conference*, 2015.

[70] F. Boutros, M. Huber, P. Siebke, T. Rieber, and N. Damer, "SFace: Privacy-friendly and Accurate Face Recognition using Synthetic Data," in *IEEE International Joint Conference on Biometrics, IJCB 2022, Abu Dhabi, United Arab Emirates, October 10-13, 2022*, IEEE, 2022, pp. 1–11. doi: 10.1109/IJCB54206.2022.10007961.

[71] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015. doi: 10.1109/cvpr.2015.7298682.

[72] "Build an Image Search Engine." [Online]. Available: https://fireship.io/lessons/image-search-engine/

[73] D. Qi, W. Tan, Q. Yao, and J. Liu, "YOLO5Face: Why Reinventing a Face Detector." 2022.

[74] "Visual Studio Code." [Online]. Available: https://code.visualstudio.com/

[75] "catkin_tools." [Online]. Available: https://github.com/catkin/catkin_tools

[76] "catkin_make." [Online]. Available: http://wiki.ros.org/catkin/commands/catkin_make

[77] "apptainer." [Online]. Available: https://apptainer.org/

[78] "robocup_container." [Online]. Available: https://github.com/LASR-at-Home/Containers/blob/main/robocup_container.def

[79] "catkin_virtualenv." [Online]. Available: https://github.com/locusrobotics/

[80] T. Foote, "tf: The transform library," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, in Open-Source Software workshop. Apr. 2013, pp. 1–6. doi: 10.1109/TePRA.2013.6556373.

[81] "House M.D.." [Online]. Available: https://www.imdb.com/title/tt0412142/

[82] M. R. Hannemose, "Recreated Lena Picture." [Online]. Available: https://mortenhannemose.github.io/lena/

[83] "OpenCV Tutorial: Cascade Classifiers." [Online]. Available: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[84] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks," *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1499–1503, 2016, doi: 10.1109/LSP.2016.2603342.

[85] "Releases - weaviate/weaviate." [Online]. Available: https://github.com/weaviate/weaviate/releases

[86] "weaviate/Dockerfile at master." [Online]. Available: https://github.com/weaviate/weaviate/blob/master/Dockerfile#L51

[87] "Docker Compose | Weaviate." [Online]. Available: https://weaviate.io/developers/weaviate/installation/docker-compose#configurator

[88] "PyPI - Whisper." [Online]. Available: https://pypi.org/project/openai-whisper/

[89] "PyPI - SpeechRecognition." [Online]. Available: https://pypi.org/project/SpeechRecognition/

[90] "PyPI - typer." [Online]. Available: https://pypi.org/project/typer/

[91] "PyPI - rich." [Online]. Available: https://pypi.org/project/rich/

[92] "PyPI - shapely." [Online]. Available: https://pypi.org/project/shapely/

[93] "PyPI - numpy." [Online]. Available: https://pypi.org/project/numpy/

[94] "PyPI - opencv-python." [Online]. Available: https://pypi.org/project/opencv-python/

[95] "General Data Protection Regulation - Legal Text." [Online]. Available: https://gdpr-info.eu/

[96] "Data Protection Act 2018." [Online]. Available: https://www.gov.uk/data-protection

[97] "Art 15. GDPR." [Online]. Available: https://gdpr-info.eu/art-15-gdpr/

[98] "Art 17. GDPR." [Online]. Available: https://gdpr-info.eu/art-17-gdpr/

[99] "Art 18. GDPR." [Online]. Available: https://gdpr-info.eu/art-18-gdpr/

[100] "Art 24. GDPR." [Online]. Available: https://gdpr-info.eu/art-24-gdpr/

[101] "Facial recognition cameras – what your rights are." [Online]. Available: https://www.daslaw.co.uk/blog/facial-recognition-cameras-what-your-rights-are

[102] "British Computing Society. Code of Conduct for BCS Members." [Online]. Available: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

[103] "LICENSE.md for this project." [Online]. Available: https://git.is.horse/insert/university/undergraduate-project/-/blob/main/LICENSE