ECE 445

SENIOR DESIGN LABORATORY

FINAL REPORT

Visual Chatting and Real-Time Acting Robot

Team #37

HAOZHE CHI
(haozhe4@illinois.edu)
MINGHUA YANG
(minghua3@illinois.edu)
JIATONG LI
(jl180@illinois.edu)
ZONGHAI JING
(zonghai2@illinois.edu)

TA: Enxin Song

May 31, 2024

Abstract

This project focuses on the development and implementation of an AI-enhanced robotic service system designed to assist blind individuals in navigating large public spaces and safely interacting with water dispensers. The system comprises three main components: the navigation system, the Raspberry Pi auxiliary system, and the PCB water dispenser system. The navigation system utilizes advanced large language models (LLMs) and large visual language models (LVLMs) to process visual inputs from a head-mounted camera and verbal commands from the microphone, providing real-time guidance and safety instructions. The Raspberry Pi auxiliary system integrates object detection, hardware control, and automation, leveraging a fine-tuned SSD-MobileNet-V2 model in TFLite format to achieve high accuracy in detecting and handling water bottles and cups. This system is connected with a UR3e robot arm, MegaPi controller, and various sensors to enable seamless automated operations. The PCB water dispenser system coordinates with the robot arm to ensure precise and safe water bottle refilling. Extensive testing confirms the system's reliability, efficiency, and effectiveness in real-world scenarios, demonstrating the feasibility of deploying complex AI models and robotics for practical applications in assistive technology.

Keywords: AI-Enhanced Robotics, LLM, LVLM, Raspberry Pi, Object Detection, SSD-MobileNet-V2, TFLite, UR3e Robot Arm, MegaPi, PCB, Real-Time Automation

Contents

1 Introduction **1**
- 1.1 Problem Statement 1
- 1.2 Solution Overview 1
- 1.3 Visual Aid 2

2 Design **3**
- 2.1 Block Diagram 3
- 2.2 Subsystems Overview 3
  - 2.2.1 Camera Subsystem 3
  - 2.2.2 Image Encoder Subsystem 4
  - 2.2.3 Q-Former Subsystem 4
  - 2.2.4 Large Language Model Subsystem 4
  - 2.2.5 Text Tokenizer Subsystem 5
  - 2.2.6 Speech-to-Text Subsystem 5
  - 2.2.7 Microphone Subsystem 5
  - 2.2.8 ROS Subsystem 5
  - 2.2.9 Text-to-Speech Subsystem 6
  - 2.2.10 Voice Player Subsystem 6
  - 2.2.11 Universal Robot UR3e Robot Arm Subsystem 6
  - 2.2.12 Raspberry Pi Auxiliary Subsystem 7
  - 2.2.13 PCB Water Dispenser Subsystem 7

3 Design Details **8**
- 3.1 Navigation System 8
  - 3.1.1 Speech-to-Text and Text-to-Speech Modules 8
  - 3.1.2 Navigation Algorithm 8
  - 3.1.3 BLIP-2-Based Visual Language Model Deployment 9
  - 3.1.4 Apply Acceleration Module 11
  - 3.1.5 Design of Real-Time Screenshot Program 11
- 3.2 Raspberry Pi Auxiliary System 13
  - 3.2.1 System Architecture Overview 13
  - 3.2.2 Block Diagram of Overall Work Flow 14
  - 3.2.3 Water Bottle Object Detection Task 15
  - 3.2.4 Robot Gripper Control 18
  - 3.2.5 Communication Setup with ROS System 19
  - 3.2.6 Model Selection for Text-to-Speech 21
  - 3.2.7 Communication Setup with PCB Water Dispenser System 21
  - 3.2.8 Shell Script for the Whole Raspberry Pi Auxiliary System 23
- 3.3 PCB Water Dispenser System 23

4 Verification **26**
- 4.1 Performance of the LVLM 26
- 4.2 Performance of the SSD-MobileNet-V2 model 27

4.2.1 Loss Figures 27
4.2.2 Learning Rate Figure 28
4.2.3 Training Process Analysis 29
4.2.4 Test Results and Analysis 29
4.3 Performance of the PCB Board 32
4.3.1 Attempts to Improve Brightness 32
4.3.2 Possible Causes and Future Steps 32

5 Tolerance Analysis **33**

6 Cost Analysis **34**

7 Conclusion **35**
7.1 Ethics and Safety 35
7.1.1 Ethics 35
7.1.2 Safety 36

References **38**

Appendix A Team Photo and Introductions **40**

Appendix B Code and Algorithm **41**
B.1 Code for Taking Screenshots 41
B.2 Main ROS Code to Control the Robot Arm 41
B.3 Inference Script for TFLite Model 42
B.4 Bottle Position Detection Algorithm 43
B.5 Code for Robot Gripper 44
B.6 Raspberry Pi GPIO Control 45
B.7 Shell Script Implementation 46

1 Introduction

1.1 Problem Statement

Blind individuals often face significant difficulties when navigating unfamiliar environments, such as finding water dispensers in large public spaces. Additionally, there is a risk of injury from interacting with devices that dispense hot water. The emergence of large language models (LLMs) and large visual language models (LVLMs) offers a promising avenue for developing innovative solutions to these challenges.

1.2 Solution Overview

Our team develop an AI-enhanced robotic service system aimed at assisting blind individuals in navigating large public spaces to safely access and interact with water dispensers. This initiative addresses the significant challenges that visually impaired people face, such as the risk of injury from devices dispensing hot water and difficulty in locating such amenities.

The proposed system combines advanced technological components including large language models (LLMs) and large visual language models (LVLMs). These models process both visual inputs from a camera mounted on the user's head and verbal commands via speech-to-text AI technology. It provides real-time, actionable guidance and safety instructions. To enhance interaction experience between users and the automated bottle refilling facilities, we also develop a Raspberry Pi Auxiliary system that can provide auditory guidance through visual monitoring. Our system comprises the following key components:

- **Real-Time Visual and Verbal Input Processing:** A combination of a head-mounted camera and speech-to-text AI captures and analyzes the user's surroundings and voice commands.
- **Dynamic Guidance and Interaction:** The BLIP-2 model will provide navigation assistance, warn of potential dangers, and instruct on interacting with a water dispenser.
- **Autonomous Assistance:** A Universal Robot Arm UR3e, controlled by the Robot Operating System and instructed by the Raspberry Pi Auxiliary System, will autonomously refill the user's water bottle.
- **User Communication:** Audio feedback and instructions will be delivered through a Bluetooth headset, ensuring clear and effective communication.

Operational Process When a blind individual approaches a water dispenser, the system triggers a specific sequence of actions:

1. The Vision Language AI model guides the user to the water dispenser.

2. The Raspberry Pi Auxiliary System will then provide audio instructions to help user place their bottle in a designated location.
3. Subsequently, a robot arm, following instructions from the Raspberry Pi system, securely grasps the bottle, fills it with water from the dispenser, and then returns the filled bottle to the user.

1.3 Visual Aid

The visual illustration of our AI-enhanced robotic service system is shown in Figure 1.

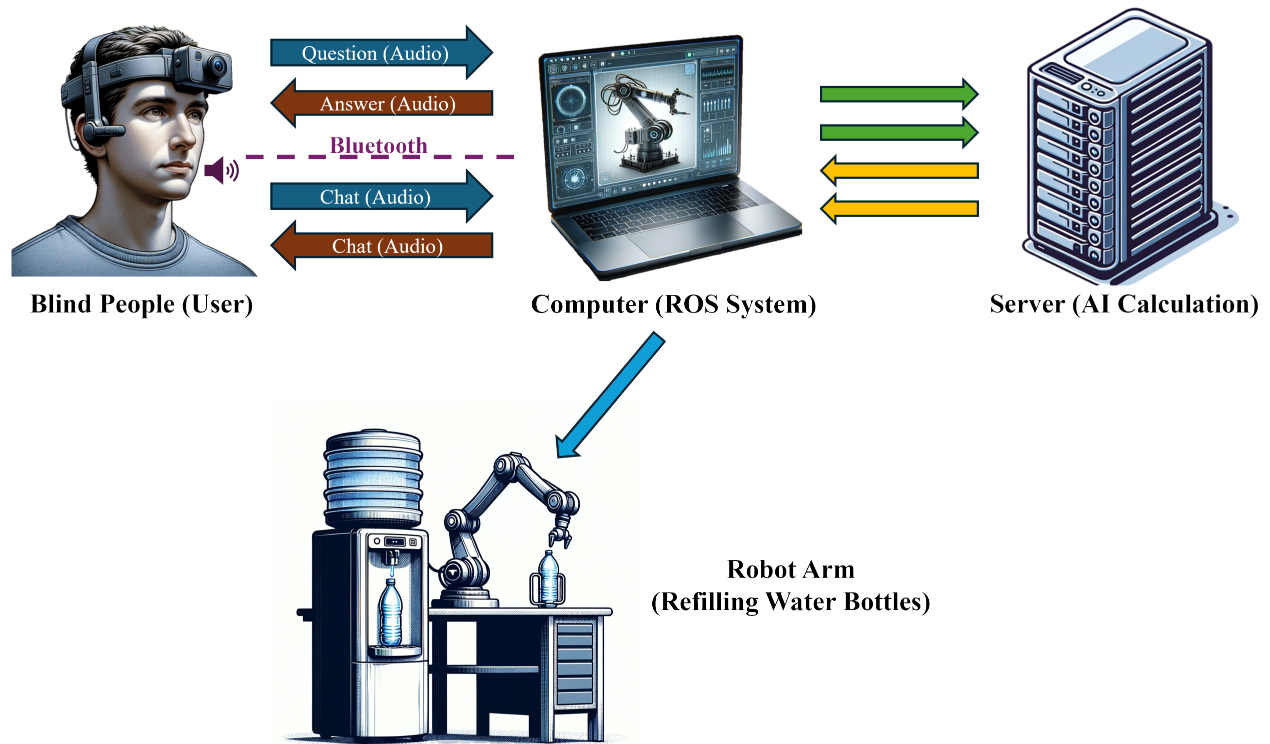


Figure 1: Visual Illustration of the AI-enhanced Robotic Service System

2 Design

2.1 Block Diagram

The general block diagram of our AI-enhanced robotic service system is shown in Figure 2. Our whole project can be mainly divided into two components, the navigation component and the automated bottle filling component. In the Navigation component, the LVLM takes the input from the head-mounted camera and the audio input from microphone, and then output corresponding answers to the voice player. In the automated bottle filling component, the Raspberry Pi acts as the central control core of this whole component, accepts visual input from cameras and output audio feedback to users as well as sending instructions to each hardware like robot arm, robot gripper and PCB water dispenser board to make them work together as expected.

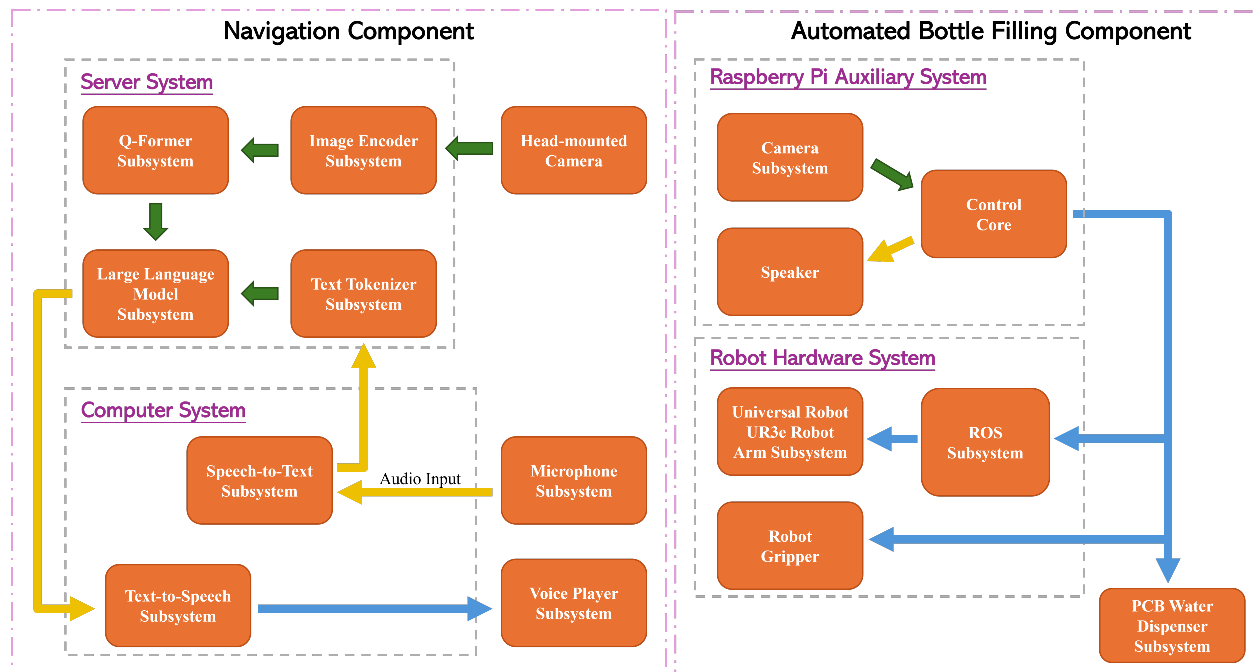


Figure 2: Block diagram of the AI-enhanced robotic service system (Green arrow: visual flow; Yellow arrow: text flow; Blue arrow: instruction flow).

2.2 Subsystems Overview

2.2.1 Camera Subsystem

The Camera Subsystem is a pivotal element in our robotic framework, acting as the primary data collection point. Utilizing an iPhone camera mounted on the user's head, this subsystem captures the user's environment and streams video in real-time to a connected Mac. This setup ensures a continuous flow of high-resolution visual data to the Image Encoder Subsystem. The system's contribution is quantified by its ability to deliver high-resolution video under varying lighting conditions and maintain a seamless frame rate

essential for subsequent processing stages. The interface with the Image Encoder Subsystem is defined by the video resolution, frame rate, and the real-time data transfer rate necessary for effective processing.

2.2.2 Image Encoder Subsystem

The Image Encoder Subsystem is an integral part of the robotics framework, responsible for converting visual input into a format suitable for advanced analysis. Leveraging the Vision Transformer (ViT) structure, specifically a **ViT-L/14** model, this subsystem provides a streamlined and feature-enriched representation of images captured by the Camera subsystem. It processes images through 32 queries of 768 dimensions each, aligning with the Q-Former's specifications for efficient interfacing. The subsystem's output, a compressed 32×768 matrix denoted as Z, presents a more efficient alternative to the initial 257×1024 ViT-L/14 image features. By reducing data dimensionality, it plays a crucial role in optimizing the computational workflow and facilitating swift data exchange with the Q-Former subsystem.

2.2.3 Q-Former Subsystem

The Q-Former Subsystem serves as a crucial component in the processing pipeline of our design, utilizing transformer architecture to elevate visual data into abstract representations. This subsystem ingeniously employs attention mechanisms to enhance the visual features received from the Image Encoder subsystem before passing them on to the Large Language Model (LLM) subsystem. Quantitatively, it boasts two transformer submodules that share self-attention layers to refine features from varying image resolutions. The image transformer submodule is tasked with the visual aspect, and the text transformer handles the encoding and decoding of textual information. Through this setup, the Q-Former ensures that the interaction between visual and textual data is not only seamless but also optimized for the highest efficiency in real-time processing.

2.2.4 Large Language Model Subsystem

The Large Language Model (LLM) Subsystem is a sophisticated computational unit within our robotics architecture, integral for synthesizing both visual and textual data into actionable text outputs. It processes embeddings from the Image Encoder and Text Tokenizer Subsystems using an advanced **Llama** model, a choice inspired by the **BLIP-2** [1] architecture which ensures comprehensive and nuanced text generation. This subsystem's outputs are specifically formatted to instruct the Robot Operating System (ROS) for executing tasks or providing responses. The quantitative measure of this subsystem's performance is assessed by the quality and relevance of text outputs generated, as well as the speed and accuracy with which it processes input embeddings into these outputs. The interface with the Image Encoder and Text Tokenizer is marked by the standardized embedding vectors received, while its output interface with the ROS subsystem is quantified by the command strings dispatched for robotic control.

2.2.5 Text Tokenizer Subsystem

The Text Tokenizer Subsystem is a pivotal component tasked with converting raw textual inputs into structured embeddings. It serves as an intermediary, translating spoken language captured by the Speech-to-Text subsystem into a format amenable to computational analysis. Utilizing a **BERT** model tokenizer ensures compatibility with advanced Large Language Models like Llama, enabling robust text interpretation. This subsystem significantly contributes to the overall design by ensuring that linguistic information is accurately represented and processed, facilitating the system's ability to comprehend and act upon user commands. The interface with the Speech-to-Text subsystem is defined by the text input stream, while the output interface with the Large Language Model consists of tokenized text embeddings.

2.2.6 Speech-to-Text Subsystem

The Speech-to-Text Subsystem is a key interface that translates auditory information into a digital text format, bridging human interaction and machine processing. Using an advanced open-source model, this subsystem decodes spoken language with high accuracy and low latency, making it an essential component for real-time applications. It quantitatively contributes to the overall design by providing accurate text conversion, serving as the initial processing step for voice commands. The efficacy of this subsystem is measured by its transcription accuracy and speed, which directly impacts the performance of the downstream Text Tokenizer subsystem.

2.2.7 Microphone Subsystem

The Microphone Subsystem is an integral component of our robotic system, tasked with capturing audio input from users in a clear and reliable manner. Utilizing a Bluetooth microphone, this subsystem offers flexibility and enhances the robot's ability to interact with its environment by ensuring high-quality audio capture. This audio is then transmitted to the Speech-to-Text subsystem, where it is converted into textual data for further processing. The performance of this subsystem is quantitatively measured by its audio capture fidelity, noise reduction capability, and the latency in transmitting the captured audio to the Speech-to-Text subsystem. Its seamless integration and reliability are critical for the effective operation of the robot's interactive capabilities.

2.2.8 ROS Subsystem

The ROS (Robot Operating System) Subsystem acts as the central control unit within our robotics framework, crucial for interfacing with both software components and hardware mechanisms. Leveraging MQTT, a lightweight messaging protocol, it facilitates real-time communication with the Raspberry Pi Auxiliary System, which manages the operational commands for the robot arm. This shift enhances the system's responsiveness and reliability, especially in low-bandwidth environments. The ROS Subsystem translates these commands into precise physical actions, coordinating closely with the Raspberry Pi to ensure seamless execution. Additionally, it continues to relay necessary responses back

to the Text-to-Speech subsystem for user interaction. The system's performance is quantitatively assessed by its command execution latency, reliability in task execution, and the efficiency of inter-process communication.

2.2.9 Text-to-Speech Subsystem

The Text-to-Speech Subsystem is an essential communicative bridge in our robotics architecture, enabling the robot to convert textual responses into spoken words, thus facilitating a natural interaction with users. Utilizing the ***pyttsx3***[1], a versatile and open-source text-to-speech Python library, this subsystem translates textual data received from the ROS subsystem into audible speech, which is then relayed through the Voice Player subsystem for output. The choice of pyttsx3 not only supports a broad range of voices and languages but also ensures functionality without the need for internet connectivity. This subsystem's contribution to the overall design is quantitatively marked by its speech synthesis speed, clarity of the generated audio, and the seamless interface with the ROS and Voice Player subsystems, enabling the robot to provide timely and intelligible responses to user inquiries.

2.2.10 Voice Player Subsystem

The Voice Player Subsystem is a critical component for enabling the robot to audibly communicate with users, functioning as the final step in the interactive feedback loop. It takes the audio files generated by the Text-to-Speech Subsystem and plays them through a Bluetooth headset, ensuring clear and understandable speech output. This subsystem is essential for the robot's ability to provide audible responses to user queries or commands, enhancing the overall user experience. Its contribution to the design is quantified by its audio output clarity, playback latency, and compatibility with the Bluetooth headset, facilitating effective human-robot interaction.

2.2.11 Universal Robot UR3e Robot Arm Subsystem

The Universal Robot UR3e Robot Arm Subsystem, enhanced with a Makeblock Robot Gripper, is a critical component of our robotics architecture, providing high precision and flexibility for physical tasks. The gripper allows the arm to grasp and handle objects like water bottles more effectively. The gripper is controlled by a Raspberry Pi, which interfaces directly with the ROS subsystem to receive and execute detailed instructions. Equipped with six rotational joints, the subsystem executes movements with high precision, crucial for the accurate positioning and handling of objects within its operational environment. This subsystem's performance metrics include its reach, payload, repeatability, and the added functionality of the gripper's grasping capabilities.

[1] https://pyttsx3.readthedocs.io/en/latest/

2.2.12 Raspberry Pi Auxiliary Subsystem

The Raspberry Pi Auxiliary Subsystem serves as a sophisticated monitoring and interaction enhancer between the user and the robotic system, utilizing three cameras and a speaker for precise control and feedback. The first two cameras are dedicated to monitoring the water bottle on the desk, ensuring it is within the Robot Arm's reach and providing precise audio instructions for adjustments along the x-axis and y-axis, such as "Move your bottle right/left/forward/back a bit." The third camera checks the positioning of the bottle at the water dispenser to guarantee accurate filling. This subsystem leverages advanced object recognition technology to pinpoint the bottle's location and provide verbal guidance accordingly. Connected to the PCB Water Dispenser Subsystem, the Robot Gripper, and the ROS Subsystem, it orchestrates a seamless interaction flow and enhances operational efficiency. This integration is critical for the system's functionality, offering real-time and intuitive user guidance.

2.2.13 PCB Water Dispenser Subsystem

The PCB Water Dispenser Subsystem is integral to the functionality and user interface of our robotic system, acting as a visual communicator for the operational status of the water dispensing process. By employing a light control mechanism with programmable LEDs, this subsystem indicates when the water is being dispensed (green light) and when the process is complete (red light), based on the input from the Raspberry Pi Auxiliary System. This direct, visual feedback mechanism is crucial for coordinating the actions of the robot arm, especially in guiding it to retrieve and return the filled water bottle to the user. The subsystem's design is quantitatively defined by the accuracy of signal reception, the precision of the internal timer for light transitions, and the reliability of sending completion signals back to the Raspberry Pi Auxiliary System. This ensures seamless integration within the broader system, enhancing the robot's interactive capabilities.

3 Design Details

Our AI-enhanced robotic service system has three main components, the Navigation System, the Raspberry Pi Auxiliary System, and the PCB Water Dispenser System.

3.1 Navigation System

The Navigation System includes the following part of work:

1. Deploy the speech-to-text and text-to-speech modules. This is essential for the AI model to process blind people's vocal input and give corresponding guidance.
2. Deploy the depth-map generation module and complete scripts for navigation algorithm. Since we do not use depth camera, we need a depth-map generation module to generate depth map from original visual image. Then the navigation algorithm would use this depth map to detect whether there is any potential danger.
3. Establish stable connection between my personal computer and the AI server, and realize efficient data transmission. This connection is important for stable real-time chatting.
4. Deploy Large Visual Language Models on the AI server. This is the core of our design, as the LVLM provides real-time guidance and instructions for blind people.
5. Apply model acceleration technique to the AI model. This is also important for reducing the delay of processing data with AI model.
6. Design real-time visual screenshot program using Zoom App and transfer real-time visual input to the AI server.

3.1.1 Speech-to-Text and Text-to-Speech Modules

We deploy both speech-to-text and text-to-speech modules on PC (Mac OS). Specifically, we use a virtual machine VMware to install the Ubuntu system and deploy open-source modules on it. To realize the speech-to-text process, we use the pre-trained ***silero***[2] model.

Firstly we use the pyaudio library in Python to record the voice and save it as a wave file. Then we use the pre-trained silero model to process the wave file and turn it into text. To realize the text-to-speech process, we use the open-source project ***pyttsx3***. After receiving the JSON file from the AI server, our program would read the answer message from the JSON file and use pyttsx3 to turn it into audio and play it through Bluetooth devices.

3.1.2 Navigation Algorithm

The navigation algorithm has different priorities as shown in Figure 3. For the highest priority, the system should remind the potential danger ahead immediately. The potential

[2] https://github.com/snakers4/silero-models

danger is detected through the depth map analysis. If there is no potential danger, then it comes to the second priority. For the second priority, the system should respond to blind people's vocal inputs. If there is also no vocal input from blind people, then it comes to the third priority. For the third priority, the system should repeat the navigation route that blind people should follow.

For the generation of depth map, we use NYU FCRN network. It's based on Resnet50 and several depth blocks are added. Also, since the route in our design is fixed, the repeated guidance is also set to be fixed. We set a threshold K and would let the LVLM remind the danger only if there are object in the depth map owning depth smaller than K, like the cup in the figure.

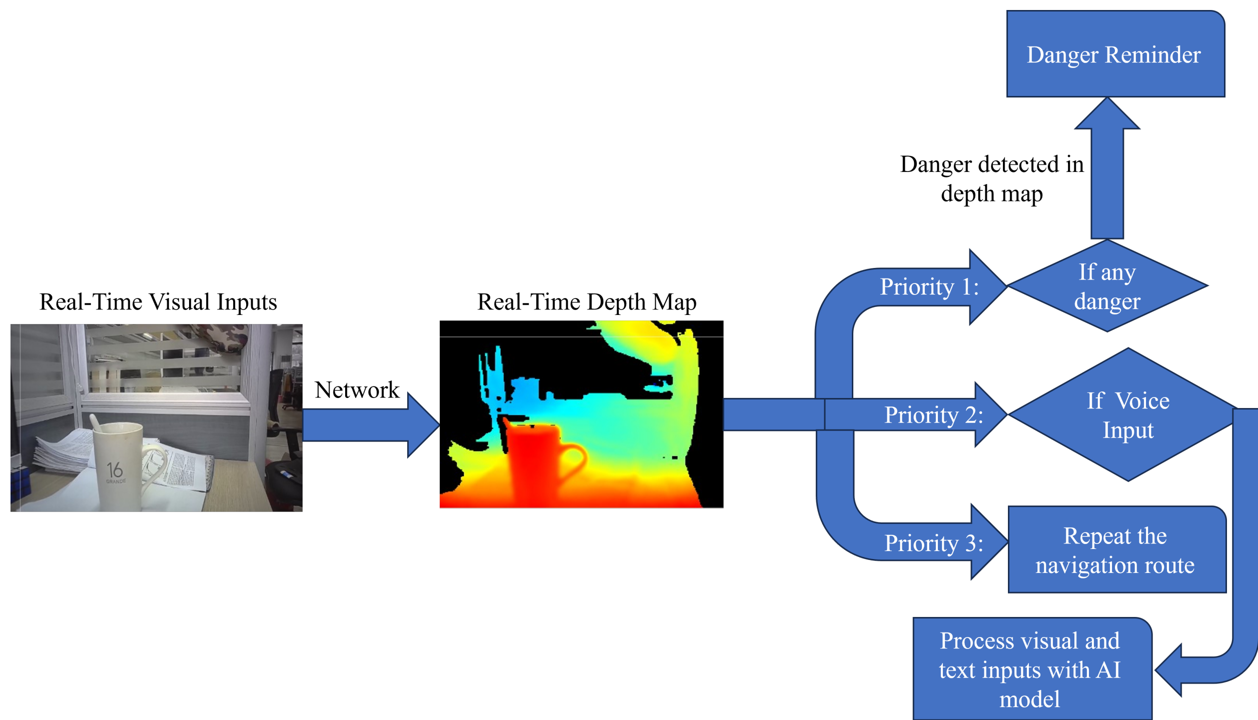


Figure 3: Navigation Algorithm Illustration

3.1.3 BLIP-2-Based Visual Language Model Deployment

We establish a stable connection between our PC and the AI server. To be specific, we use the paramiko library in Python to establish an SSH connection. We use SFTP protocol to realize data transmission. The overall transmission is fast and stable. Each time when new short video or text prompt is generated, we use sftp.put('remotepath') and sftp.get('remotepath') methods to send and fetch files. The new videos are generated every second and thus are sent to the server every second. Whenever the blind people ask a question, a question json file would be generated and sent to the server. Then the LVLM model would process the newest text file along with the newest video and give response. Whenever a new response json file is generated at the server, we would fetch the newest

response using sftp.get('remotepath') method. Since the file transmission is under ZJU WLAN, the overall delay is within 1ms for both short videos and text files.

We deploy the Large Visual Language Model on the AI server. To be specific, we deploy the modifed VideoChatgpt Model on the AI server. VideoChatgpt [2] is one of the common video understanding models. We use the BLIP-2-based Visual Language Model for vision-language interaction. We would send real-time visual and audio data through a connection to the server. Then the LVLM model would process it and send the updated answering messages back to the personal computer. The detailed architecture of the BLIP-2-based model [1] is shown in Figure 4.

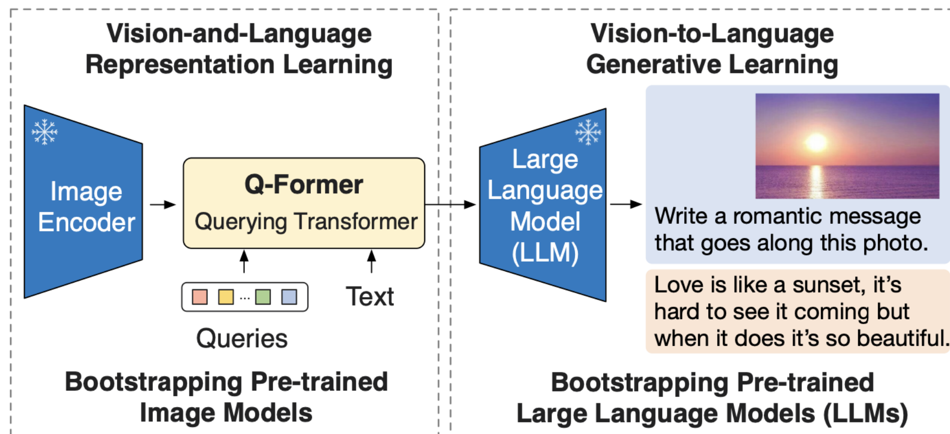


Figure 4: Architecture of the BLIP-2-based Model

And VideoChatgpt do not contain a Q-former module for cross-modality processing. The detailed architecture of VideoChatgpt model is shown in Figure 5.

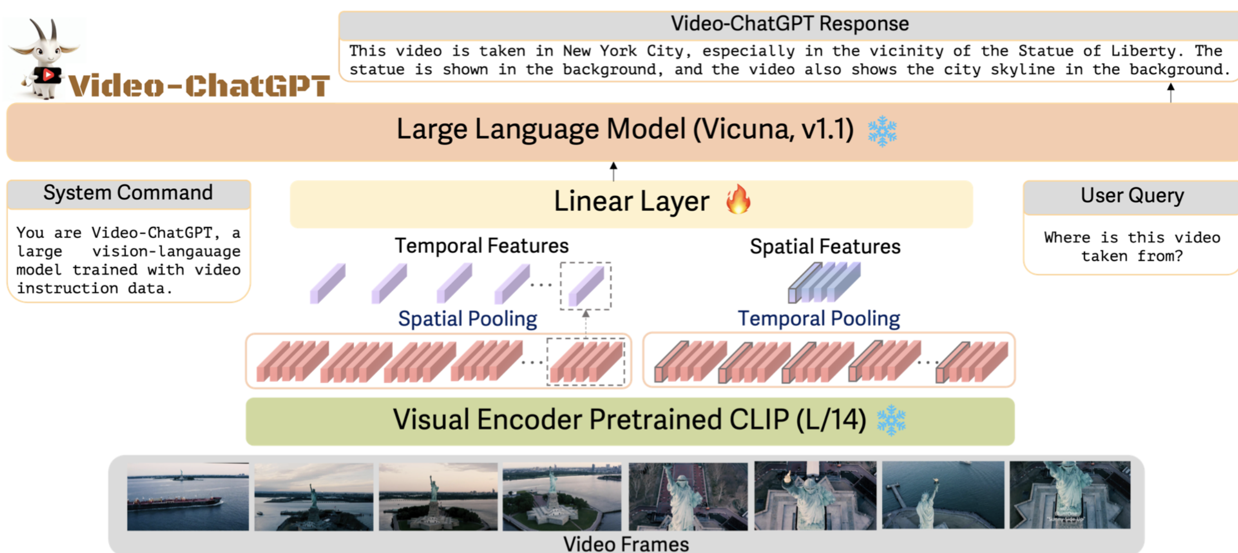


Figure 5: Architecture of the Video-ChatGPT Model

To adapt Q-former structure to VideoChatgpt architecture, we remove the spatial pooling stage and replace it with a pre-trained Q-former from VideoLlama. In this way, the temporal features are generated by Q-former while the spatial features are still generated by temporal pooling. With Q-former and position embedding applied, more temporal relations could be obtained through attention mechanism. The detailed architecture of modified VideoChatgpt is shown in Figure 6.

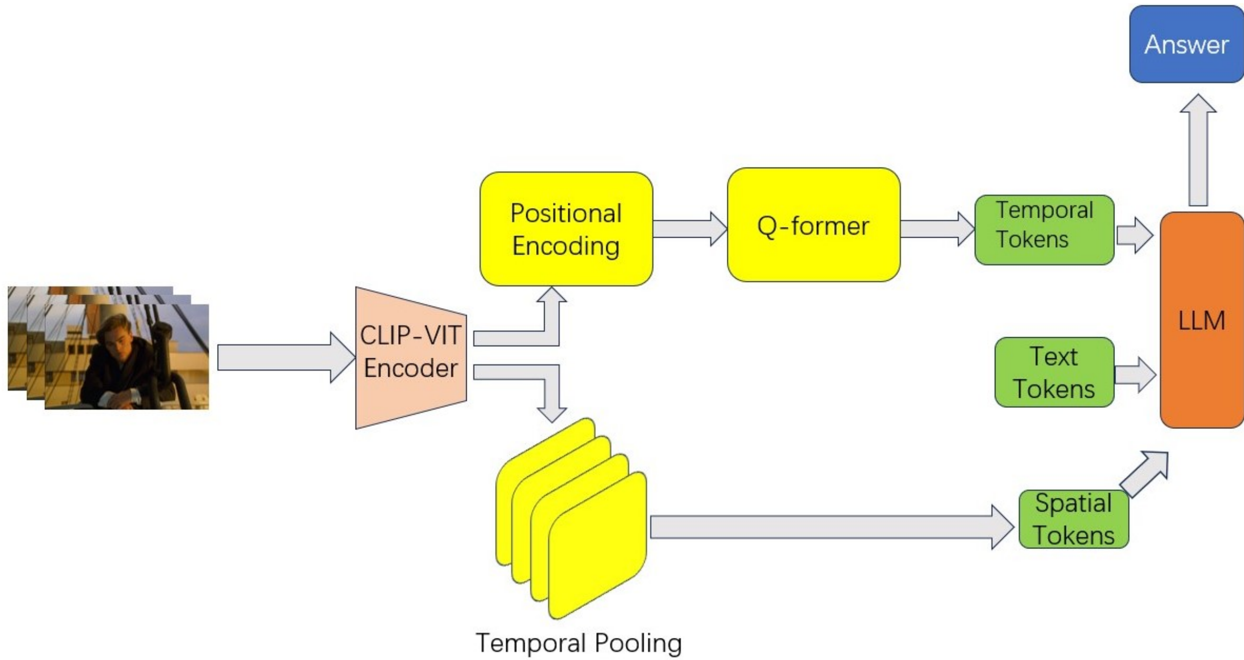


Figure 6: Architecture of the Modified Video-ChatGPT Model

3.1.4 Apply Acceleration Module

To make inference of LVLM much faster, we add flash-attention module to the Llama model, which is the LLM decoder at downstream of the architecture. To be detailed, flash-attention uses tiling strategy to calculate the attention blocks in transformers in parallel. The key idea behind is that the softmax function could be decomposed into tiles and traditional tiling strategy could be applied. Figure 7 shows the detailed algorithm of flash-attention.

As mentioned in the flash-attention paper [3], flash-attention method speeds up the training of GPT-2 model a lot and better utilize the bandwith of GPU and CPU. The layout of flash-attention is shown in Figure 8, which demonstrates the idea of looping and tiling.

3.1.5 Design of Real-Time Screenshot Program

To get real-time visual information, the blind people uses an iPhone to capture images every second. To better utilize existing softwares and make the process easier and faster,

we use Zoom to connect the personal computer to iPhone. We write a script to take screenshot every second in the zoom meeting, turn the shot image into short videos and upload them to the AI server.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.
1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\text{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

Figure 7: Algorithm of the Flash-attention

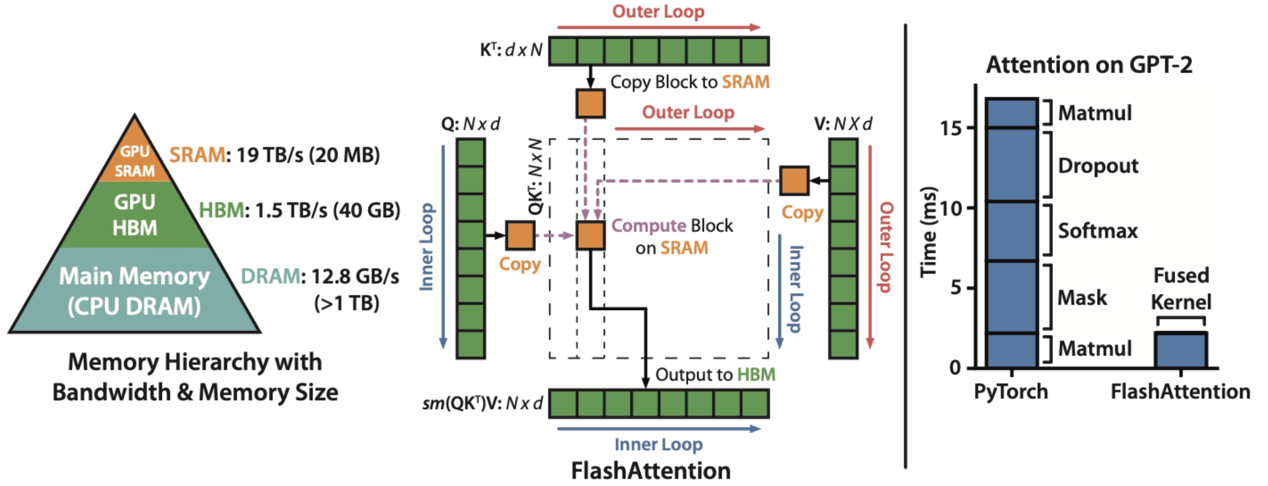


Figure 8: Layout of the Flash-attention

To be more precise, we use pyautogui library to take screenshot automatically. Then we crop the informative area of Zoom screen and save it to an image. After that, we use ImageSequenceClip class from moviepy library to generate short videos using the image

shot. Finally, we upload the generated video to the AI server through sftp protocol every second. The detailed code is shown in Appendix B.1.

3.2 Raspberry Pi Auxiliary System

The integration of the Raspberry Pi Auxiliary System into our project was driven by the need to address significant challenges observed in the initial design, which relied heavily on a Vision Language AI model for controlling all operations. Here, we outline the reasons behind adopting the Raspberry Pi system to enhance functionality and efficiency.

Initial Design and Challenges: Our initial setup tasked the Vision Language AI model with guiding interactions entirely, from bottle placement to operation of the robot arm and water dispenser. This centralized approach led to considerable processing delays due to the model's inherent latency and added complexity, impacting system responsiveness and precision.

Rationale for the Raspberry Pi Auxiliary System: The adoption of the Raspberry Pi Auxiliary System was aimed at mitigating these issues through several key improvements:

1. **Reduced Latency:** By decentralizing control, the Raspberry Pi significantly cuts down on overall system latency, facilitating near real-time interactions between the user and the robotic arm.
2. **Simplified Processes:** The Raspberry Pi manages direct control over the robot arm and water dispenser, simplifying operations and allowing the Vision Language AI model to focus on providing high-level navigational aid rather than managing minute details.
3. **Enhanced Precision and Interaction:** With dedicated object detection and text-to-speech models, the Raspberry Pi system provides precise localization of the water bottle and clear auditory instructions, improving the guidance provided to users.
4. **Improved System Efficiency and Usability:** These modifications ensure smoother, more convenient operations and enhance the user experience, particularly in facilitating effective interaction for visually impaired users.

Introducing the Raspberry Pi Auxiliary System was a strategic choice to optimize our project's structure, enhancing operational efficiency, responsiveness, and user interaction. This approach not only improved system performance but also better aligned with our objectives of supporting visually impaired individuals in public spaces.

3.2.1 System Architecture Overview

The Raspberry Pi Auxiliary System is strategically designed to enhance interaction convenience for visually impaired users. As shown in Figure 9, it integrates three cameras

and a speaker to facilitate user interaction with the robot arm:

- **Two Web-Cameras:** Positioned to monitor the placement of the water bottle on the desk (both X and Y directions), ensuring it is within the operational range of the robot arm.
- **Pi-Camera:** Used to confirm the bottle's correct positioning at the water dispenser, ensuring accurate filling.
- **Speaker:** Provides real-time vocal instructions to guide the user in adjusting the bottle's placement.

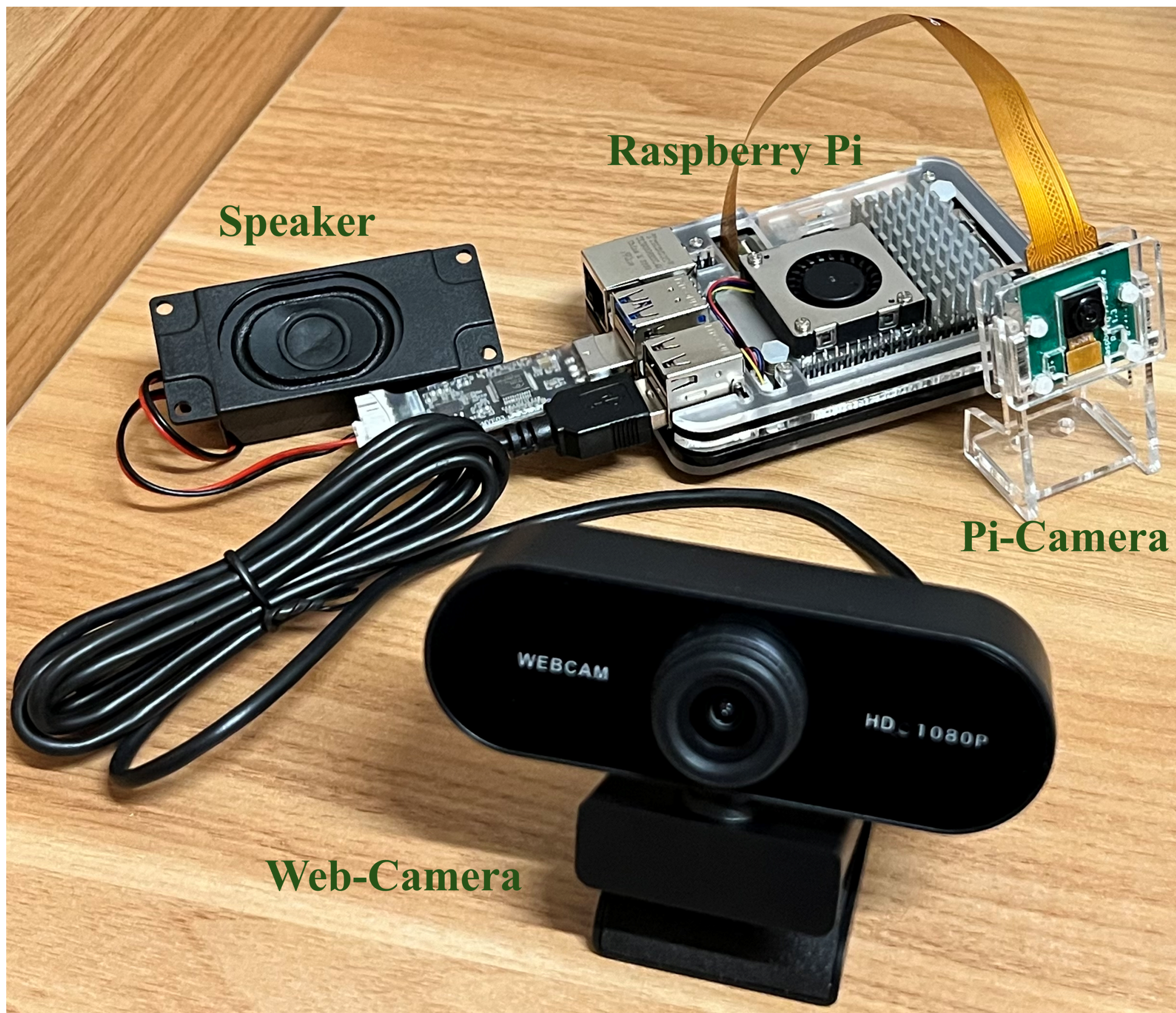


Figure 9: Architecture Overview of the Raspberry Pi Auxiliary System

3.2.2 Block Diagram of Overall Work Flow

The overall workflow of the Raspberry Pi Auxiliary System is illustrated in Figure 10. The process consists of six control steps:

1. The Raspberry Pi core control center captures visual input from camera-x and camera-y. It uses the object detection model to locate the position of the water bottle placed on the desk.
2. Based on the detected position of the water bottle, the control center outputs audio instructions to help the user move the bottle (right/left/forward/back) to the designated location so that the robot gripper can grasp it. (Steps 1 and 2 are repeated until the water bottle is correctly placed.)
3. The robot gripper grabs the water bottle.
4. The control center, through the ROS system, directs the robot arm to move the water bottle to the water dispenser.
5. The control center takes visual input from camera3, mounted on the water dispenser, to verify the presence of a bottle at the dispenser, ensuring accurate filling.
6. If the check in the previous step is successful, the control center sends a start signal to the PCB board to begin filling the bottle.

After filling, the robot arm moves the water bottle back to the user, and the robot gripper releases the bottle. Finally, the control center outputs an audio message indicating that the user can now pick up the bottle.

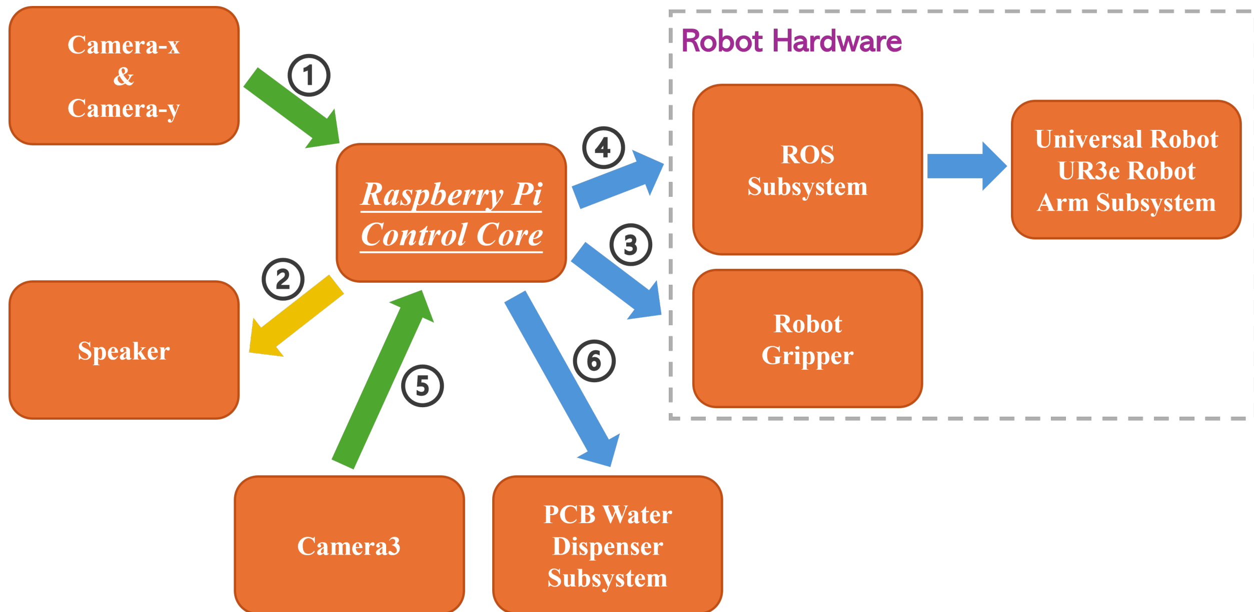


Figure 10: Overall Workflow Block Diagram of the Raspberry Pi Auxiliary System

3.2.3 Water Bottle Object Detection Task

Model Selection For the task of detecting water bottles, we selected the **SSD-MobileNet-V2** model [4] because it offers an optimal balance between speed and accuracy, which

makes it well suited for applications in near real time on our edge device, the Raspberry Pi. SSD-MobileNet-V2 is designed specifically for mobile devices and edge devices, combining the Single Shot Multi-Box Detector (SSD) framework [5] with MobileNetV2 [6]. MobileNetV2 builds on its predecessor, MobileNetV1 [7], by introducing an inverted residual structure with linear bottlenecks. This enhancement significantly improves computational efficiency by maintaining crucial depth information for performance while reducing model size and computational cost. This efficiency is crucial in reducing latency and ensuring that the system can operate in real time, which is essential for user interaction.

Model Training and Finetuning For the water bottle detection task, we selected the pretrained SSD-MobileNet-V2 model from the **TensorFlow 2 Object Detection Model Zoo**[3] [8]. Specifically, we used the following configuration:

```
'ssd-mobilenet-v2-fpnlite-320': {
   'model_name': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8',
   'base_pipeline_file': '
      ↪ ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.config',
   'pretrained_checkpoint': '
      ↪ ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz',
}
```

This pretrained model is initially trained on the COCO dataset [9], which includes a wide variety of objects, allowing the model to learn robust feature representations. Utilizing a pretrained model provides several benefits for our project. Firstly, it reduces the computational resources and time required for training from scratch. The SSD-MobileNet-V2 model is particularly advantageous due to its efficiency and accuracy, making it suitable for deployment on resource-constrained devices such as the Raspberry Pi.

Motivation for Finetuning and Transfer Learning Method The primary task of our project is to detect water bottles. Although the pretrained SSD-MobileNet-V2 model is effective, fine-tuning it specifically for water bottle detection can significantly enhance its performance for this particular task. Fine-tuning involves adjusting the pretrained model on a smaller, task-specific dataset, which in our case, is a custom dataset of water bottles.

We used the transfer learning method to fine-tune the pretrained model on our custom water bottle dataset. Transfer learning leverages the knowledge gained from the large-scale COCO dataset and applies it to our specific task, enabling the model to adapt to the nuances of water bottle detection. This approach is more efficient than training a model from scratch, as it requires less data and computational power while providing better performance due to the pre-learned features from the larger dataset.

[3]TensorFlow2-Object-Detection-Model-Zoo

Custom Water Bottle Dataset As shown in Figure 11, my custom water bottle dataset consists of 260 images of water bottles and cups, collected from the Web and daily life environments. These images represent common bottles and cups that users are likely to carry, ensuring the relevance and applicability of the model.

Using the labelling tool **LabelImg**[4] [10] from GitHub, we manually labeled all the images by drawing bounding boxes around the water bottles and cups and assigning the appropriate labels. Once labeled, the dataset was randomly split into training, validation, and test sets. We then created a Labelmap and TFRecords, which are required by TensorFlow for the training process.

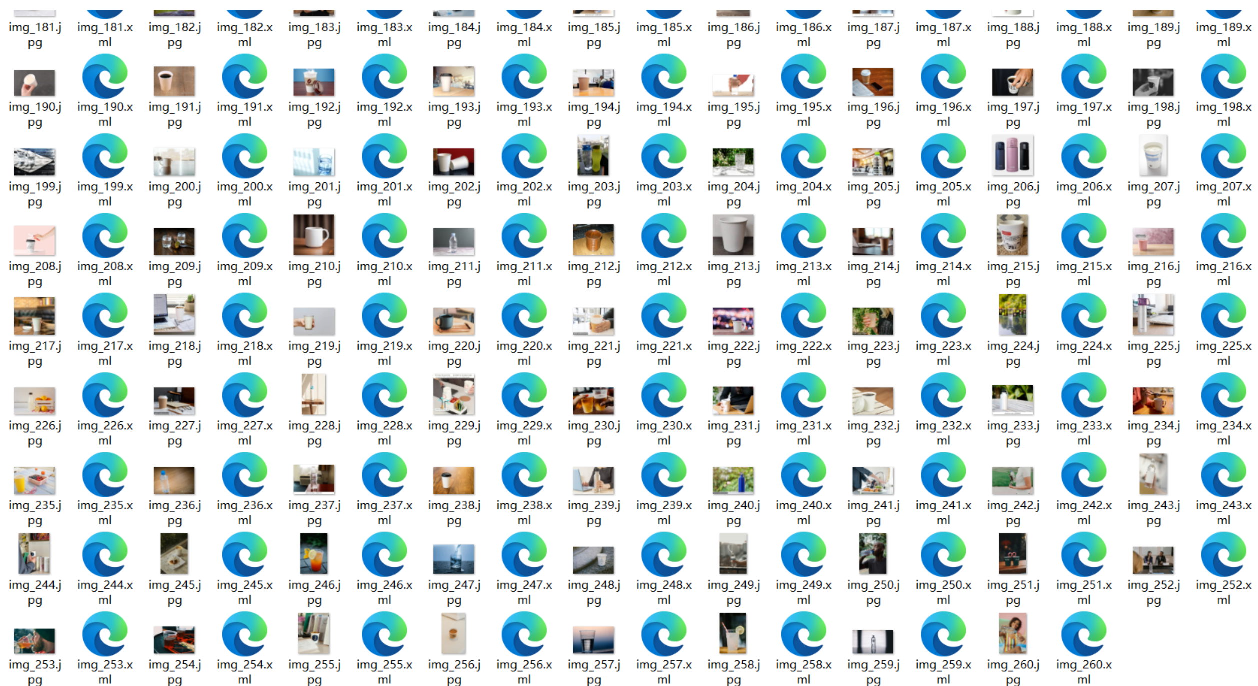


Figure 11: Screenshot of My Custom Water Bottle Dataset

Model Deployment After the fine-tuned model is trained, the next crucial step is to export the model graph, which contains information about the architecture and weights, to a TensorFlow Lite (TFLite) format. This conversion is essential for several reasons, particularly when deploying models on edge devices such as the Raspberry Pi. The basic outline of the inference script is shown in Appendix B.3.

The primary motivation for converting the model to the TFLite format is to optimize it for performance on resources-constrained devices. The TFLite format offers several benefits over the standard TensorFlow (TF) model format:

1. **Reduced Model Size**: TFLite models are significantly smaller in size compared to their TF counterparts. This reduction is achieved through various optimization tech-

[4]https://github.com/HumanSignal/labelImg

niques such as quantization, which helps in reducing the model footprint and making it suitable for devices with limited storage.

2. **Faster Inference**: TFLite models are optimized for speed. They are designed to perform efficient inference on devices with limited computational power. This optimization ensures that the model can run in real time, providing quick responses, which is critical for applications requiring immediate feedback, such as water bottle detection.

3. **Lower Latency**: By converting to TFLite, the inference latency is minimized. This is particularly important for interactive applications where user experience depends on how quickly the system can process input and provide output.

4. **Energy Efficiency**: TFLite models are designed to be energy-efficient, making them ideal for battery-powered edge devices. Reduced energy consumption ensures that these devices can operate longer without frequent recharges.

Bottle Position Detection Algorithm In our water bottle detection task, determining whether the detected water bottle is correctly positioned within the frame is crucial for providing accurate guidance to the user. The Algorithm shown in Appendix B.4 was developed to judge the position of the detected water bottle relative to the center of the screen. There are two key concepts used in the algorithm:

Central Position Threshold The *central_position_threshold* is a predefined value that determines the allowable range around the center of the screen within which the water bottle is considered correctly positioned. If the center of the detected water bottle's bounding box lies within this threshold, the bottle is deemed to be in the center. This threshold helps in reducing sensitivity to minor movements and ensures that the detection is robust.

Position Stability Threshold The *position_stability_threshold* is another predefined value that ensures the detected position of the water bottle is stable over a series of frames. This stability check prevents the system from reacting to transient or noisy detections, thereby providing a more reliable indication of the bottle's position.

3.2.4 Robot Gripper Control

The robot gripper used to grab the water bottle is the Makeblock Robot Gripper, mounted on the UR3e Robot Arm as shown in Figure 12. This gripper is operated through the MegaPi [11] controller, which is programmed and interfaced with a Raspberry Pi. The setup and control of the robot gripper involve several steps to ensure precise manipulative operations. The detailed code is shown in Appendix B.5.

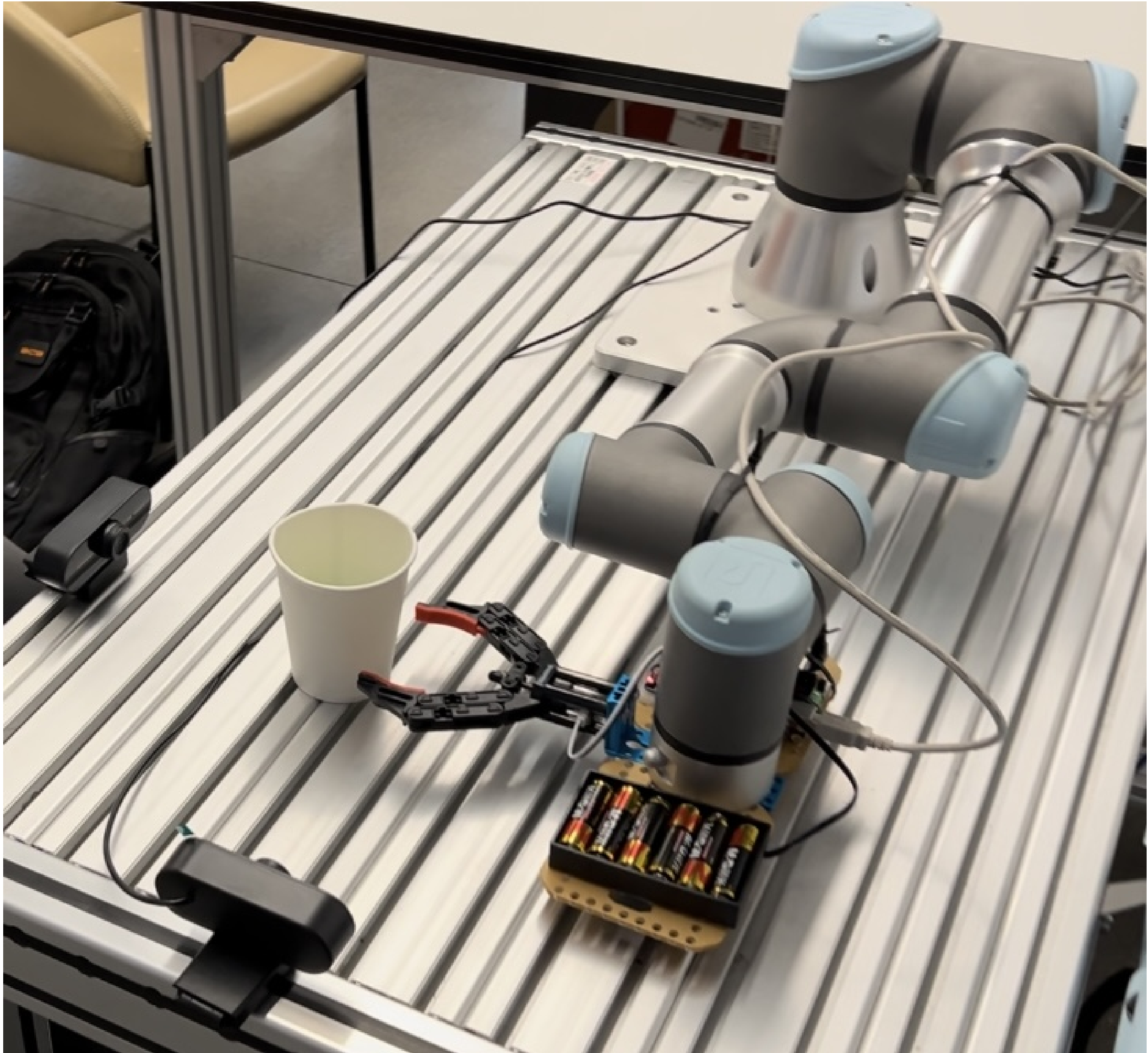


Figure 12: The robot gripper is mounted on the robot arm, with two cameras controlled by the Raspberry Pi to detect the position of the water bottle.

3.2.5 Communication Setup with ROS System

To control the UR3e robot arm [12], a robust communication setup involving a Virtual Machine (VM) and the Robot Operating System (ROS) is established. This setup allows for the control of the robot arm from a Raspberry Pi via SSH, providing a streamlined method of sending instructions remotely.

Setup and Control Procedures

1. **Virtual Machine Configuration**: An Ubuntu VM (version 20.04) is set up on a PC using virtualization software. This VM serves as the host for the ROS environment

and interfaces with the UR3e robot arm.

2. **ROS Installation**: ROS Noetic [13], which is compatible with Ubuntu 20.04, is installed on the Ubuntu VM. This version of ROS provides the necessary packages and libraries to interface with the robot hardware and execute control commands.

3. **Network Configuration**: To ensure seamless communication between the VM and the robot arm, the VM's network is configured to use a bridged adapter. This setup utilizes the Realtek PCIe GbE Family Controller, allowing the VM to connect directly to the same network as the host machine and the robot arm.

4. **Physical Connection**: The UR3e Robot Arm is connected to the PC using a network cable. This direct connection minimizes latency and ensures reliable communication.

5. **Automated Control Script**: On the Raspberry Pi, a shell script is developed to automate the control of the robot arm. This script uses SSH to securely connect to the Ubuntu VM and execute ROS commands. The SSH setup allows remote and secure command transmission from the Raspberry Pi to the VM.

6. **Enable SSH Communication**: Enable the hotspot on the PC, then set up a new bridge network with Microsoft Wi-Fi Direct Virtual Adapter #2 on VM. Connect both the Raspberry Pi and the VM to the PC hotspot to ensure that they are under the same subnet.

On the Ubuntu VM:

Set up communication with the robot arm by executing the following commands in two separate terminals:

```
# Terminal 1
roslaunch ur_robot_driver ur3_bringup.launch robot_ip
   ↪ :=192.168.1.120

# Terminal 2
rosrun ur3_driver ur3_driver _robot_ip:=192.168.1.120
```

On the Raspberry Pi:

Use a shell script to automate the control of the robot arm. The following example script connects to the VM via SSH and runs ROS commands to reset the robot arm, move it to the water dispenser, and move it back to the user.

```
# Shell script to control the UR3e robot arm
ssh user@vm-ip-address << EOF
rosrun RobotArm_pkg_py RobotArm_reset.py
rosrun RobotArm_pkg_py RobotArm_forward.py
rosrun RobotArm_pkg_py RobotArm_back.py
EOF
```

- The *RobotArm_reset.py* script resets the robot arm to its original position.
- The *RobotArm_forward.py* script sends instructions to the robot arm to move to the water dispenser.
- The *RobotArm_back.py* script commands the robot arm to return to the user after filling the water bottle.

The main ROS code to control the robot arm is shown in Appendix B.2.

3.2.6 Model Selection for Text-to-Speech

For converting text outputs into audible instructions, we chose the **eSpeak**[5] module. eSpeak is known for its simplicity, lightweight design, and wide language support, making it suitable for real-time speech synthesis on constrained devices. Its compact size and efficient speech generation capabilities ensure minimal impact on the system's resources, aligning with the need for a fast response time in our application.

3.2.7 Communication Setup with PCB Water Dispenser System

The communication between the Raspberry Pi and the PCB water dispenser system is established through GPIO pins, as shown in Figure 13 [14]. This setup is crucial for indicating the operational status of the water dispenser system using LEDs and for managing the water dispensing process based on camera detections.

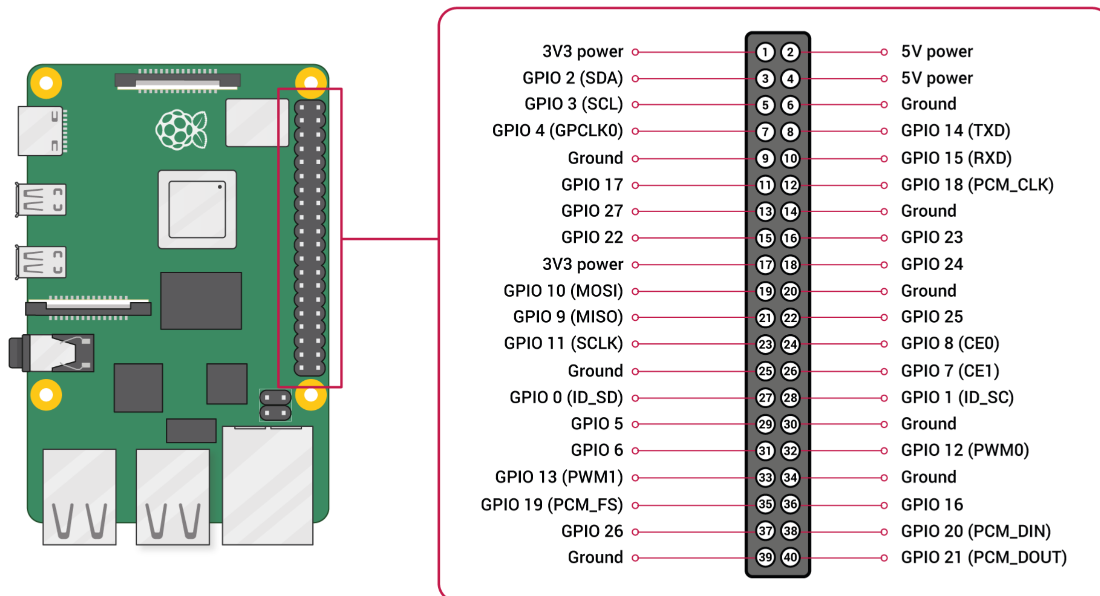


Figure 13: Raspberry Pi GPIO 40-Pin Layout

[5] https://espeak.sourceforge.net/

LED Indicators The PCB water dispenser board is equipped with two LEDs:

- **Green LED**: Indicates that the entire water dispenser system is operational. This LED turns on when the system is powered up and remains on during normal operation.
- **Red LED**: Indicates that hot water is being dispensed. This LED turns on only when the camera mounted on the water dispenser detects that a water bottle is placed at the dispenser. The LED turns off once the filling process is complete.

GPIO Pin Configuration The communication with the LEDs is managed through GPIO pins on the Raspberry Pi. Specifically, two pins are utilized:

- **Green LED Pin**: Connected to GPIO pin 27.
- **Red LED Pin**: Connected to GPIO pin 17.
- **Ground Pin**: Connected to GPIO Ground pin (No.6 in the layout diagram).

These pins control the LEDs on the PCB board, providing a straightforward and low-latency interface to manage the water dispensing actions based on the camera's detections. The detailed code is shown in Appendix B.6.

Bottle Detection and Red LED Control As shown in Figure 14, the system checks if a water bottle is placed at the dispenser using the *Detect_Bottle* function. If a bottle is detected, the red LED is turned on to indicate that hot water is being dispensed. The LED remains on during the filling process and turns off once the process is complete. If no bottle is detected, the system provides an appropriate audio message, and then the robot arm will move back to the original position.

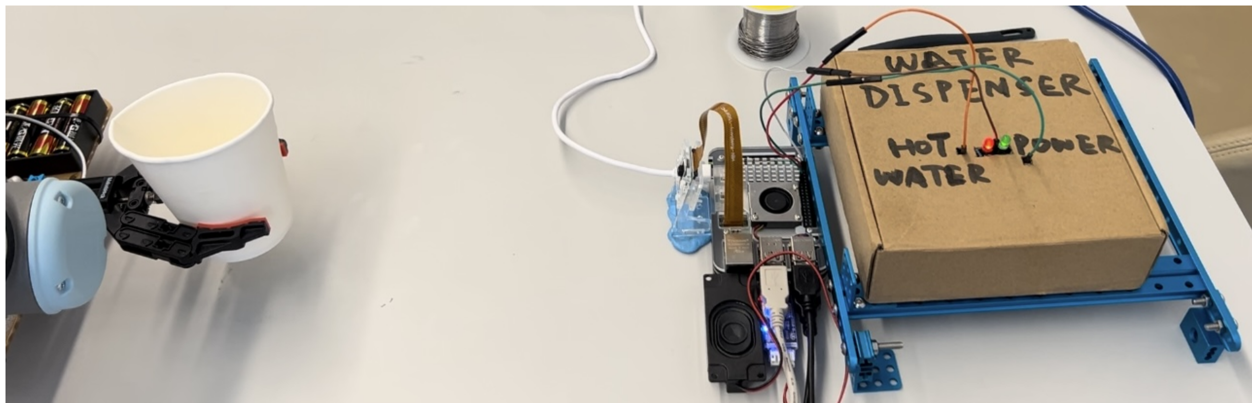


Figure 14: The camera3 detects that a water bottle is placed at the water dispenser, then the red LED is turned on to indicate that hot water is now being dispensed.

3.2.8 Shell Script for the Whole Raspberry Pi Auxiliary System

To integrate the above all components of the Raspberry Pi Auxiliary System and achieve automatic control, a comprehensive shell script is used. This script orchestrates the actions of the camera detection, robot gripper, UR3e robot arm, and LED indicators on PCB board to provide a seamless and automated process for water bottle handling. The following is an introduction to the shell script and its functionalities.

Overview of the Shell Script The shell script is designed to perform the following sequence of operations:

1. Reset the robot arm to its initial position.
2. Turn on the green power LED light to indicate that the system is operational.
3. Wait for the camera to detect the water bottle.
4. Grab the water bottle using the robot gripper.
5. Move the robot arm with the bottle to the water dispenser.
6. Check if the water bottle is correctly placed at the dispenser and fill it with water.
7. Move the robot arm back to the user.
8. Release the water bottle.

Using a shell script to control each component together provides several benefits, including streamlined automation, reduced manual intervention, and improved synchronization between components. And the detailed implementation is shown in Appendix B.7.

3.3 PCB Water Dispenser System

The design details of our PCB board of the simulated water dispenser are shown in the following figures. On the board there are two LED lights with several gate-controlled logic. Based on the input from the Raspberry Pi Auxiliary System, when the whole system is powered up (green light on), and when the hot water is being dispensed (red light on).

The Design of the PCB Board As shown in Figure 15, inputs include voltage sources and signal sources from the Raspberry Pi, with NAND gates to control the red and green lights to come on at different times. In the whole circuit, most of the devices are intuitive, but we still want to illustrate the 74LS00 for U1U2 and the 74LS04 for U3. As a basic building block in digital circuits, the 74LS00 can be used to implement NAND logic. And 74LS04 is used to act as voltage regulator.

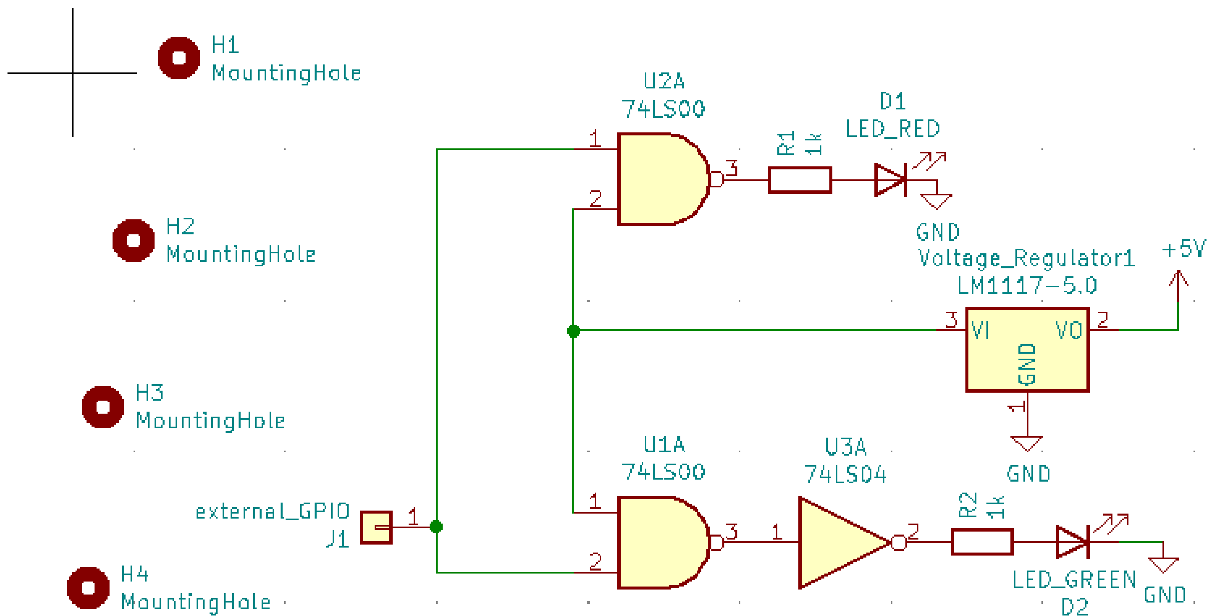
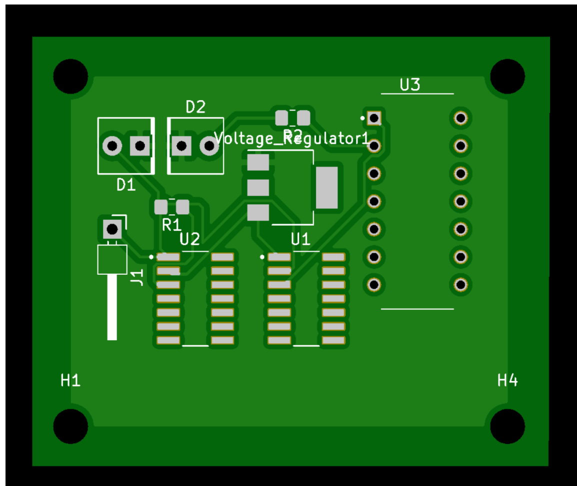


Figure 15: PCB Schematics

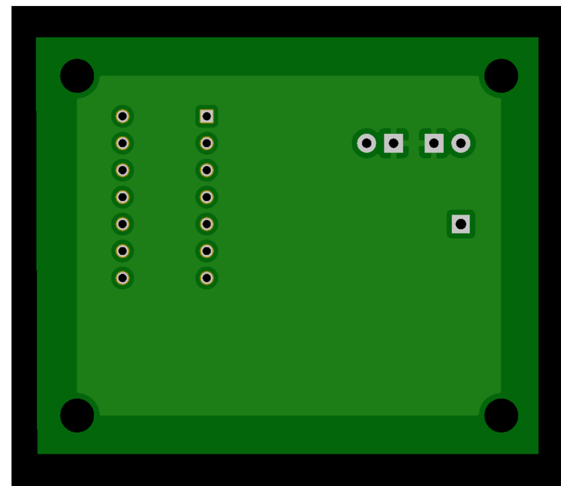
```
符号:封装分配
  1          D1 -          LED_RED : LED_THT:LED_Rectangular_W5.0mm_H5.0mm
  2          D2 -        LED_GREEN : LED_THT:LED_Rectangular_W5.0mm_H5.0mm
  3          H1 -     MountingHole : MountingHole:MountingHole_3.2mm_M3
  4          H2 -     MountingHole : MountingHole:MountingHole_3.2mm_M3
  5          H3 -     MountingHole : MountingHole:MountingHole_3.2mm_M3
  6          H4 -     MountingHole : MountingHole:MountingHole_3.2mm_M3
  7          J1 -    external_GPIO : Connector_PinHeader_2.54mm:PinHeader_1x01_P2.54mm_Horizontal
  8          R1 -               1k : Resistor_SMD:R_0805_2012Metric_Pad1.20x1.40mm_HandSolder
  9          R2 -               1k : Resistor_SMD:R_0805_2012Metric_Pad1.20x1.40mm_HandSolder
 10          U1 -           74LS00 : TTL:SOIC127P600X175-14N
 11          U2 -           74LS00 : TTL:SOIC127P600X175-14N
 12          U3 -           74LS04 : TTL:DIP794W45P254L1969H508Q14
 13 Voltage_Regulator1 -       LM1117-5.0 : Package_TO_SOT_SMD:SOT-223
```

Figure 16: PCB Footprints Details

Figure 17: PCB Schematics in KiCad



(a) PCB Front



(b) PCB Back

4 Verification

4.1 Performance of the LVLM

To verify the effectiveness of the modified VideoChatgpt model, we conduct comprehensive experiments on Video QA datasets. We also quantatively test the inference delay of model to show the effectiveness of the added acceleration module flash-attention.

Model	Size	MSVD	MSRVTT	ActivityNet
FrozenBiLM [15]	1B	32.2	16.8	24.7
VideoChat [16]	7B	56.3	45.0	26.5
Video-ChatGPT [2]	7B	64.9	49.3	35.2
Video-LLaMA [17]	7B	51.6	29.6	12.4
Video-LLaVA [18]	7B	70.7	59.2	45.3
Ours	7B	65.7	51.6	34.1

Table 1: Short Video Question Answering Results (Accuracy) on MSVD, MSRVTT, and ActivityNet Dataset

Model	Size	Time
VideoChatgpt	7B	5.6s
VideoChatgpt + flash-attention	7B	4.1s
Ours	7B	5.9s
Ours + flash-attention	7B	3.8s

Table 2: Inference Time Comparison

We conduct experiments on short video QA datasets MSVD, MSRVTT and NextQA. MSVD is a short video dataset collected by Microsoft company that contains 1970 short video clips in total. The average length of each video clip is 10 seconds. These clips are from Youtube and cover various themes like animals, music, sports and travelling. Each short video clip has a text describing the contents in the video. The Video QA file is generated from these video descriptions. MSRVTT dataset is also collected by Microsoft, containing 10,000 short video clips. The average length here is 20 seconds, which means it's longer compared to MSVD. MSRVTT covers more topics and show more variety of contents. Furthermore, MSRVTT include more scene switches, which make the video clips harder to understand. ActivityNet dataset is a benchmark that contains many types

of human activities. ActivityNet dataset provides various types of videos for advance video understanding, including temporal action reasoning and common scene understanding. It comprises of different motion information for further action recognition and analysis.

The modified VideoChatgpt model shows competitive results in QA tasks, as shown in Table 2. First, we can see that flash-attention reduces the total time delay by 2.1 seconds. Furthermore, we can see that flash-attention module only reduces original VideoChatgpt inference time by 1.5 seconds. This is because I add Q-former module to original model, and flash-attention module can significantly reduce the processing time of Q-former as well. Second, it's shown in Table 1 that the modified VideoChatgpt achives better accuracy than original model in MSVD dataset QA task, and shows competitive result with respect to Video-LLaVA. Moreover, the modified VideoChatgpt also shows better accuracy in MSRVTT dataset QA task. Lastly, although it does not show superior result in ActivityNet dataset QA task, it still has high accuracy compared to other baselines. The reason of lower accuracy might due to the flexibility of questions in ActivityNet, as more complex reasoning questions are included. In general, the performance of modified VideoChatgpt is impressive and inspiring. This also shows the effectiveness of Q-former structure in BLIP-2 architecture.

4.2 Performance of the SSD-MobileNet-V2 model

The fine-tuning training results of the SSD-MobileNet-V2 model are illustrated through several key figures that show the behavior of different loss functions and the rate of learning during the training process.

4.2.1 Loss Figures

The training process is evaluated using four types of loss functions as shown in Figure 19, each representing a different aspect of the model's performance.

1. **Classification Loss**: The first figure shows the classification loss, which measures the model's accuracy in classifying detected objects. This loss represents the error in predicting the correct class labels for the detected objects. A decrease in classification loss over time indicates that the model is improving in its ability to correctly classify objects.

2. **Localization Loss**: The second figure shows the localization loss, which measures the accuracy of the bounding box predictions. This loss accounts for the differences between the predicted bounding boxes and the ground truth boxes. A decrease in localization loss signifies that the model is getting better at predicting the precise locations of the objects.

3. **Regularization Loss**: The third figure shows the regularization loss, which helps to prevent overfitting by adding a penalty for large weights. This loss component ensures that the model generalizes well to unseen data. A smooth decrease in regularization loss indicates effective regularization, contributing to better generalization.

4. **Total Loss**: The fourth figure shows the total loss, which is the sum of the classification, localization, and regularization losses. This combined loss gives an overall measure of the model's performance during training. A consistent decrease in total loss suggests that the model is learning effectively, balancing classification accuracy, localization precision, and regularization.

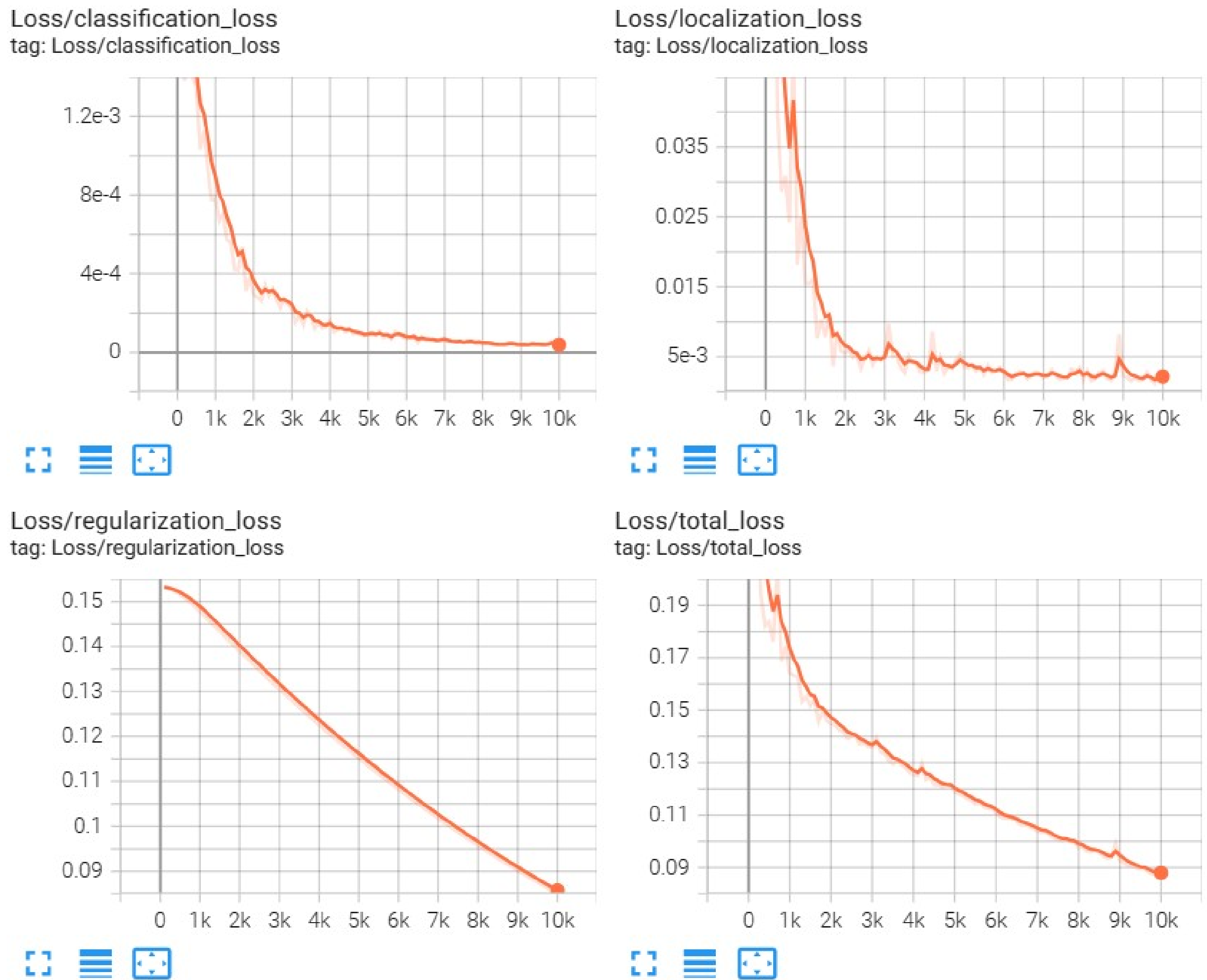


Figure 19: Four Types of Loss Functions During Training Progress

4.2.2 Learning Rate Figure

The learning rate in Figure 20 shows how the learning rate changes over the training process. The learning rate controls the step size during gradient descent optimization. Initially, the learning rate increases to allow rapid learning, then it stabilizes and gradually decreases to fine-tune the model's weights. The shape of the curve indicates that the training process starts with aggressive learning and transitions to fine-tuning as training progresses.

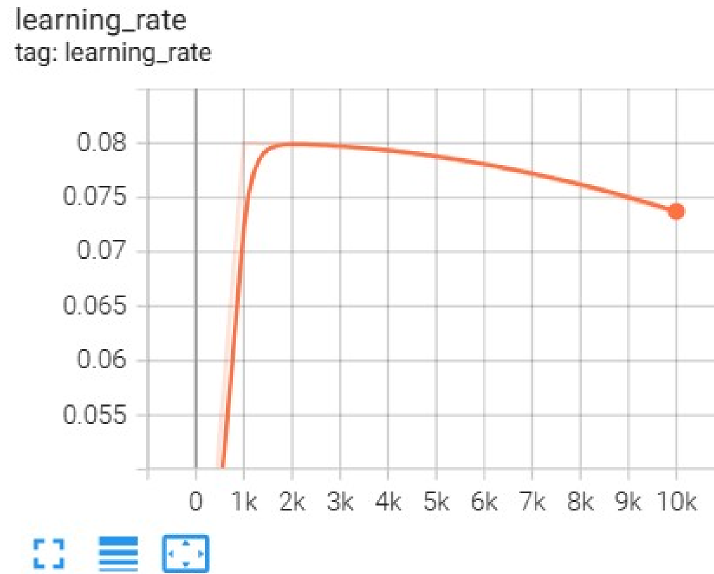


Figure 20: Learning Rate During Training Progress

4.2.3 Training Process Analysis

The general training process can be analyzed from the above loss functions and learning rate figures:

- **Early Stage**: The initial part of the training shows a steep decline in classification and localization losses, indicating a rapid improvement in both classifying objects and predicting their locations. This is supported by a high initial learning rate that facilitates quick learning.
- **Mid Stage**: As training progresses, the decrease in losses starts to slow down, showing that the model is entering a phase of fine-tuning. The learning rate stabilizes and then gradually decreases, indicating a shift from rapid learning to careful adjustment of the model's parameters.
- **Late Stage**: In the later stages of training, the losses continue to decrease at a slower rate, and the learning rate decreases further. This suggests that the model is making smaller and more precise adjustments to improve performance without overfitting.

Overall, the figures demonstrate a successful training process in which the model improves significantly in terms of classification and localization, with effective regularization to ensure generalization. The learning rate adjustments contribute to a balanced training process, optimizing both speed and accuracy.

4.2.4 Test Results and Analysis

After converting the fine-tuned SSD-MobileNet-V2 model to TensorFlow Lite (TFLite) format, I evaluated its performance on my custom test dataset discussed in Section 3.2.3.

Two examples of inference results and the mAP results on the test dataset are analyzed in the following two sections.

Inference Results Two example inference results are shown in Figure 21, captured in the actual lab environment where we have set up our robot arm and the water dispenser. In both test images, cups are placed in front of the robot arm. The green bounding boxes accurately identify the position and label of the cups. These results demonstrate the TFLite model's high accuracy in detecting and labeling the cups. The bounding boxes are well-aligned with the objects, indicating the model's ability to accurately locate and classify the items.

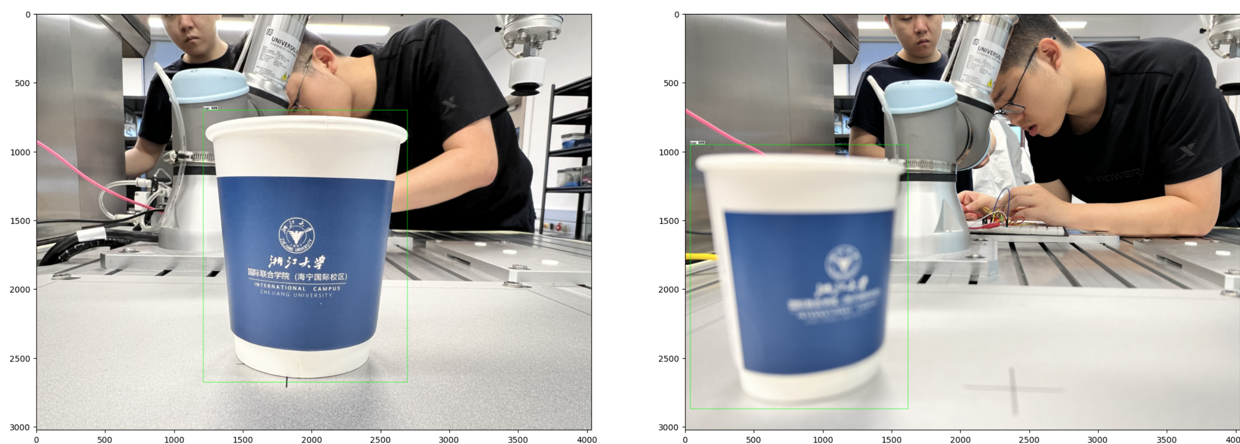


Figure 21: Example Inference Results of the TFLite Model on the Custom Test Dataset

Mean Average Precision (mAP) Results Mean Average Precision (mAP) is a standard metric used to evaluate the performance of object detection models. It considers both the precision and the recall of the detections.

- **Precision** is the ratio of true positive detections to the total number of detections made by the model.
- **Recall** is the ratio of true positive detections to the total number of actual objects present in the images.

The mAP is calculated by averaging the Average Precision (AP) for each class. AP is computed as the area under the Precision-Recall (PR) curve.

- **Precision:**

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall:**

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **Average Precision (AP):**

$$\text{AP} = \int_0^1 \text{Precision}(\text{Recall})\, d\text{Recall}$$

- **Mean Average Precision (mAP):**

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

COCO Metric for mAP @ 0.50:0.95: The mAP results for the TFLite model are determined using the **mAP calculator**[6] tool [19]. They are reported using the COCO metric for mAP @ 0.50:0.95 [20], as shown in Figure 22. This metric is the average AP across different Intersection over Union (IoU) thresholds ranging from 0.50 to 0.95 in increments of 0.05. It provides a comprehensive evaluation of the model's performance by considering varying levels of detection overlap.

```
***mAP Results***

Class           Average mAP @ 0.5:0.95
---------------------------------------
bottle          75.63%
cup             89.11%

Overall         82.37%
```

Figure 22: mAP Results for the TFLite Model on the Custom Test Dataset

- **Bottle Detection**: The model achieved an average mAP of **75.63%** for detecting bottles. This indicates a high level of accuracy, but also suggests room for improvement, possibly due to the variability in bottle appearances.
- **Cup Detection**: The model performed exceptionally well in detecting cups, with an average mAP of **89.11%**. This high accuracy demonstrates the model's robustness in recognizing and localizing cups.
- **Overall Performance**: The overall mAP of **82.37%** reflects the model's strong performance across both classes. This high mAP score indicates that the fine-tuned SSD-MobileNet-V2 model is highly effective for the water bottle detection task.

In conclusion, the TFLite model exhibits excellent detection capabilities, with high mAP scores indicating reliable performance for both bottles and cups. The conversion to TFLite format did not compromise accuracy, making it suitable for deployment on edge devices such as the Raspberry Pi.

[6] https://github.com/Cartucho/mAP

4.3 Performance of the PCB Board

We conducted tests to analyze the behavior of the LEDs in our circuit, focusing on their brightness at different voltages.

Red LED The red LED begins to shine brightly at approximately 2.2 V and reaches full brightness at around 3.0 V, indicating efficient operation within this voltage range.

Green LED The green LED, however, remains dimmer even with increased voltage.

4.3.1 Attempts to Improve Brightness

To enhance the green LED's brightness, we replaced the LED and reduced the resistance in its branch from 1000 Ω to 680 Ω. These changes did not significantly improve its brightness, making the green LED less visible during demonstrations in dim lighting.

4.3.2 Possible Causes and Future Steps

The difference in brightness may be due to varying forward voltage requirements and efficiencies between the LEDs. Future steps to balance the brightness include:

- Using LEDs with matched forward voltage and efficiency.
- Implementing precise current control.
- Exploring alternative circuit configurations.

These optimizations aim to achieve a balanced and effective LED display in our circuit.

5 Tolerance Analysis

A key design consideration is the latency in data transfer, which is critical to real-time interaction and control. We meticulously assess the latency focusiing on two main channels: user to computer, and computer to server.

User to Computer Data Transfer Analysis: A pivotal design concern is the latency during Bluetooth transmission of captured images and audio from head-mounted cameras and headsets to the computer. Assuming an operational distance of approximately 10 meters, we utilize the following formula to estimate Bluetooth transmission latency:

$$\text{Latency} = \frac{\text{Data Size}}{\text{Transmission Speed}} + \text{Propagation Delay}$$

Data Size is the total size of the data to be transmitted, measured in bits. Transmission Speed is the rate at which data is transmitted, measured in bits per second (bps). Propagation Delay is the time it takes for the signal to travel from the source to the destination, which can be calculated as the distance divided by the speed of the signal. However, for Bluetooth and similar short-range technologies operating at the speed of light, this delay is negligible compared to other factors.

Given Bluetooth 4.0's capability of up to 25 Mbps in high-speed mode and considering an average data packet size (1MB for a captured image), we can estimate the latency:

$$\text{Latency} = \frac{1 \times 10^6 \times 8 \text{ bits}}{25 \times 10^6 \text{ bits/sec}} = 0.32 \text{ seconds}$$

Computer to Server Data Transfer Analysis: Through simulations, we have estimated that data transfer delays between the computer and server can be confined to approximately 3-4 seconds. This latency is primarily influenced by network speed, server processing capabilities, and the data's complexity. Incorporating Python libraries like flash-attention has been instrumental in augmenting our AI models' processing speeds. These libraries enable more efficient handling of computations necessary for real-time analysis and decision-making based on the data received from user devices.

Conclusion: Experimental outcomes demonstrate that, despite variations, the entire processing duration stays within a few seconds, contingent on the complexity of the input data. This duration falls within our acceptable limits for real-time operations, underscoring the system's viability for responsive and effective user assistance. This analysis confirms our commitment to optimizing system performance while maintaining the real-time interaction that is vital for the success of our project.

6 Cost Analysis

Our fixed labor salary is estimated to be $10/hour, and 50 hours for each person. The total labor costs for all partners:

$$4 \cdot \$10/\text{hour} \cdot 2.5 \cdot 50 \text{ hours} = \$5000$$

The costs of all parts in our project are shown in Table 3.

Part	**Cost**
Personal Computer (Macbook)	$1200
Bluetooth Headset and iPhone Camera	$1000
Raspberry Pi System	$150
PCB Board (with Control Lights)	$20
Robot Arm and AI Server (Borrow from ZJUI)	$0

Table 3: Cost of Each Part

The grand total costs: $5000 + $1200 + $1000 + $150 + $20 = $7370.

7 Conclusion

In summary, our AI-enhanced robotic service system successfully addresses the challenges faced by blind individuals in navigating public spaces and safely interacting with water dispensers. This project has demonstrated the effective integration of advanced technologies, including large language models (LLMs), large visual language models (LVLMs), and robotics, to create a practical and reliable solution.

The key accomplishments of our project include:

- **Effective Real-Time Visual and Verbal Input Processing:** The head-mounted camera and LVLM work seamlessly to capture and analyze the user's surroundings and commands, providing real-time guidance.
- **Dynamic and Accurate User Guidance:** The BLIP-2 model provides precise navigation assistance, warning of potential dangers and instructing the user on interacting with water dispensers.
- **Autonomous Bottle Refilling:** The Universal Robot Arm UR3e, controlled by the Robot Operating System and instructed by the Raspberry Pi Auxiliary System, performs autonomous water bottle refilling, ensuring safety and efficiency.
- **Clear and Effective User Communication:** Audio feedback and instructions delivered through a Bluetooth headset guarantee clear communication, enhancing the overall user experience.

Our system has undergone rigorous testing, confirming its reliability and effectiveness in real-world scenarios. The navigation component, powered by LVLM, has proven capable of guiding users accurately to water dispensers, while the automated bottle filling component ensures a safe and efficient interaction with the dispensers.

In conclusion, the project demonstrates a significant step forward in assistive technology for blind individuals. This project showcases the potential of integrating AI and robotics to improve accessibility and safety for visually impaired individuals in public spaces. With further refinements and potential scalability, this system could be deployed widely to enhance the independence and quality of life for blind individuals, making public spaces more accessible and user-friendly.

7.1 Ethics and Safety

In the development of our robotics project, we rigorously adhere to the IEEE Code of Ethics [21] to uphold the highest standards of ethical practice and safety.

7.1.1 Ethics

Privacy (ACM 1.7: Respect the Privacy of Others) To protect privacy, we take strict measures to ensure the confidentiality and security of any personal information collected by the robot. We establish robust protocols based on industry standards to limit access

to this sensitive data to authorized individuals with a legitimate need to access it. In addition, we carefully design and implement secure storage and processing procedures to reduce the risk of unauthorized disclosure or misuse. By prioritizing the protection of personal information, we demonstrate our unwavering commitment to maintaining the privacy and trust of individuals who interact with our robots.

Fairness (IEEE - Avoiding Real or Perceived Conflicts of Interest) The possibility of bias in the decision-making process of artificial intelligence is a major ethical issue. Recognizing this, we will strive to provide robots with a comprehensive understanding of human diversity and societal nuances through rigorous training and careful refinement. By exposing robots to a variety of data, including different demographics, cultural backgrounds, and environmental scenarios, we aim to equip robots with the ability to impartially discern and understand complex social dynamics.

Being Open (ACM 1.2: Avoid Harm) We are committed to ensuring full transparency in the robotics decision-making process. Our goal is to provide clear and understandable information to all stakeholders so that they can fully understand how the robot operates and the factors that influence its decisions. To achieve this, we keep detailed records of the algorithms, data inputs and learning methods used by the robot. Additionally, we are committed to an open approach to making information about the robot's functioning, including its training data, learning outcomes, and decision logic, readily available. By increasing transparency, we aim to build trust and confidence among users, stakeholders, and the broader community, thereby promoting ethical behavior by individuals or organizations when using our robotics.

Professional Development (ACM 2.6) Adhering to ACM's principles, our team dedicates itself to the continual enhancement of our knowledge and understanding of the societal ramifications of robotics. We recognize the dynamic nature of ethical standards and proactively refine our systems to stay abreast of new developments, ensuring that our robots serve as a benchmark for responsible AI and robotics practice.

7.1.2 Safety

Avoiding Accidents (IEEE - Priority to Public Welfare) Our robots are carefully designed with safety as a top priority to ensure that they do not jeopardize the personal safety of others or the safety of property. Equipped with advanced emergency stops and a range of sophisticated sensors, the robots are able to operate with increased vigilance, effectively preventing collisions with people and objects. These safety features are carefully designed to prevent accidental collisions and provide peace of mind in dynamic environments where human-robot interactions are frequent.

Staying Secure (ACM 3.7: Recognize the Need to Protect Personal Data) Given the advanced functionality and interconnectedness of our robots, it is critical to protect their

integrity and guard against potential cyber threats. We are therefore building relevant security measures to strengthen its defenses and reduce the risks posed by malicious actors and cyberattacks. This requires the implementation of advanced encryption protocols, strict access controls and continuous monitoring mechanisms to detect and respond to any unauthorized attempts to compromise robotic systems or data. In addition, we prioritize regular security assessments and audits to identify vulnerabilities and weaknesses in our security infrastructure, enabling us to proactively address potential threats and ensure that our robots are resilient to evolving cyber threats.

Dealing with Mistakes (ACM 2.5 & IEEE - Acknowledge and Correct Mistakes) In the event of an unforeseen situation or error, the robot responds in a manner that prioritizes safety and reliability. The robot's operational framework incorporates fail-safe mechanisms and real-time monitoring capabilities to promptly identify and address any anomalies or deviations from expected behavior. By promptly notifying designated personnel or stakeholders of such occurrences, the robot facilitates rapid intervention to minimize potential risks and ensure continuity of safe and effective operations.

Responsibility (IEEE) In line with IEEE guidelines, our project is committed to the responsible deployment of robotics, ensuring they fulfill their intended roles effectively while safeguarding societal and environmental well-being. Our team maintains a vigilant approach to technology stewardship, regularly assessing and mitigating any negative impacts our robots may have, thereby ensuring our innovations contribute positively to society and operate sustainably within the environment.

Whistleblowing (ACM 1.4) Upholding ACM's ethical code, we foster an environment where whistleblowing is not just protected but encouraged, as it is crucial for maintaining the highest ethical standards. By promoting transparency and inviting scrutiny, we ensure any instance of misuse or ethical misconduct involving our robots is promptly addressed, reinforcing our commitment to integrity and the responsible use of technology.

References

[1] J. Li, D. Li, S. Savarese, and S. Hoi, "Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models," *arXiv preprint arXiv:2301.12597*, 2023.

[2] M. Maaz, H. Rasheed, S. Khan, and F. S. Khan, "Video-chatgpt: Towards detailed video understanding via large vision and language models," 2023.

[3] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022.

[4] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "Mobilenet-ssdv2: An improved object detection model for embedded systems," in *2020 International conference on system science and engineering (ICSSE)*. IEEE, 2020, pp. 1–5.

[5] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] TensorFlow, "Tensorflow 2 detection model zoo," 2021, accessed: 2024-05-25. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

[9] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.

[10] Tzutalin, "Labelimg," 2015, accessed: 2024-05-25. [Online]. Available: https://github.com/HumanSignal/labelImg

[11] Makeblock, "About megapi - programming guide," 2024, accessed: 2024-05-25. [Online]. Available: https://support.makeblock.com/hc/en-us/articles/12963818051991-About-MegaPi#6.%20Programming%20Guide

[12] U. Robots, "Ur3e robot arm technical details," 2024, accessed: 2024-05-25. [Online]. Available: https://www.universal-robots.com/media/1802780/ur3e-32528_ur_technical_details_.pdf

[13] O. S. R. Foundation, "Ros noetic installation on ubuntu," 2024, accessed: 2024-05-25. [Online]. Available: http://wiki.ros.org/Installation/Ubuntu

[14] R. P. Foundation, "Raspberry pi documentation - computers," 2024, accessed: 2024-05-25. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[15] A. Yang, A. Miech, J. Sivic, I. Laptev, and C. Schmid, "Zero-shot video question answering via frozen bidirectional language models," 2022.

[16] K. Li, Y. He, Y. Wang, Y. Li, W. Wang, P. Luo, Y. Wang, L. Wang, and Y. Qiao, "Videochat: Chat-centric video understanding," 2024.

[17] H. Zhang, X. Li, and L. Bing, "Video-llama: An instruction-tuned audio-visual language model for video understanding," 2023.

[18] B. Lin, Y. Ye, B. Zhu, J. Cui, M. Ning, P. Jin, and L. Yuan, "Video-llava: Learning united visual representation by alignment before projection," 2023.

[19] J. Cartucho, "map: Mean average precision for object detection," 2020, accessed: 2024-05-25. [Online]. Available: https://github.com/Cartucho/mAP

[20] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," 2015, accessed: 2024-05-25. [Online]. Available: https://cocodataset.org/#detection-eval

[21] Institute of Electrical and Electronics Engineers. (2016) IEEE Code of Ethics. Accessed on: 2024-03-07. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

Appendix A Team Photo and Introductions

Figure 23 was taken in the ZJUI Intelligent Robotics Laboratory of our team and Prof. Liangjing Yang. We are grateful to Prof. Liangjing Yang and the ZJUI Intelligent Robotics Laboratory for providing the UR3e Robot Arm and Robot Gripper for our project.

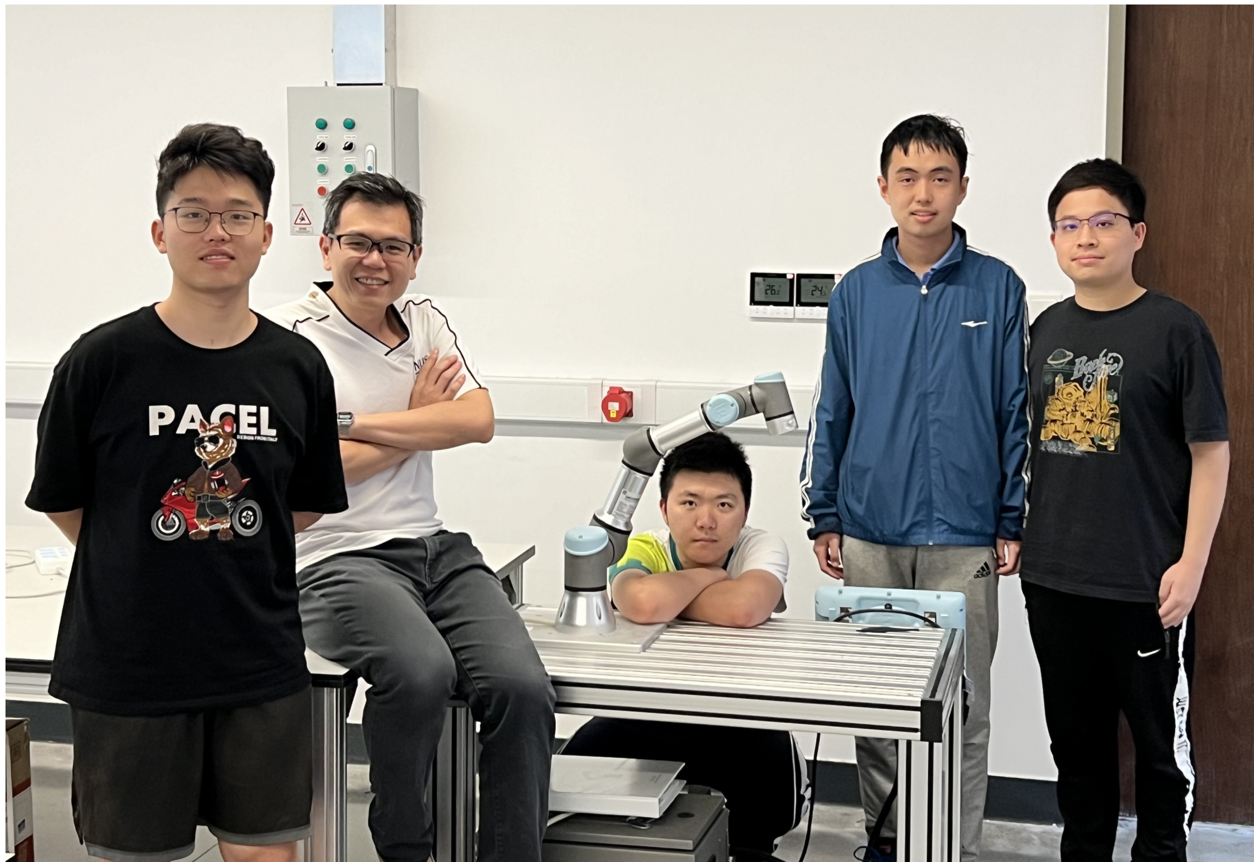


Figure 23: Team Photo

Team Members (From left to right):
Jiatong Li; Prof. Liangjing Yang; Zonghai Jing; Haozhe Chi; Minghua Yang

Appendix B Code and Algorithm

B.1 Code for Taking Screenshots

```
while True:
    time.sleep(1)
    demo_image = pyautogui.screenshot()
    demo = demo_image.crop((375, 393, 2000, 1600))
    #print(demo_image.size)
    demo.save("./demo.png")
    img_files = ['./demo.png']
    duration = 10.0
    fps = 30
    clip = ImageSequenceClip(img_files, fps=fps, durations=
        ↪ duration)
    clip.write_videofile('demo.mp4', fps=fps)
    sftp.put(filevideopth, remotevideopth)
    sftp.put(fileimgpth, remoteimgpth)
    #time.sleep(1)
```

B.2 Main ROS Code to Control the Robot Arm

```
def move_arm(pub_cmd, loop_rate, dest, vel, accel):
    global thetas
    global SPIN_RATE

    error = 0
    spin_count = 0
    at_goal = 0

    driver_msg = command()
    driver_msg.v = vel
    driver_msg.a = accel
    driver_msg.destination = dest
    driver_msg.io_0 = current_io_0
    pub_cmd.publish(driver_msg)

rospy.init_node('ECE445')

# Initialize publisher for ur3/command with buffer size of 10
pub_command = rospy.Publisher('ur3/command', command, queue_size
   ↪ =10)
sub_position = rospy.Subscriber('ur3/position', position,
   ↪ position_callback)
```

```
sub = rospy.Subscriber('/ur3/gripper_input', gripper_input,
   ↪ gripper_input_callback)
```

B.3 Inference Script for TFLite Model

```
from tflite_runtime.interpreter import Interpreter
import numpy as np
import cv2

# Load the TFLite model and allocate tensors
interpreter = Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Load the label map
with open('labelmap.txt', 'r') as f:
   labels = [line.strip() for line in f.readlines()]

# Function to perform inference
def detect_water_bottle(image_path, frame):
   # Load and preprocess each frame form camera
   input_data = np.expand_dims(frame, axis=0)

   # Set input tensor
   interpreter.set_tensor(input_details[0]['index'], input_data)

   # Run the model
   interpreter.invoke()

   # Class index of detected objects
   classes = interpreter.get_tensor(output_details[classes_idx]['
      ↪ index'])[0]

   # Get output label
   Object_name = labels[int(classes[i])]
```

B.4 Bottle Position Detection Algorithm

Algorithm 1 Bottle Position Detection

Require: Video stream from camera; screen center *screen_center_x*; central position threshold *central_position_threshold*; position stability threshold *position_stability_threshold*

1: Initialize *object_positions* $\leftarrow \{\}$, *object_stability_count* $\leftarrow \{\}$
2: Start video stream from camera
3: **while** True **do**
4: Capture frame from video stream
5: Perform object detection on the frame
6: **for** each detected object **do**
7: **if** object is a bottle or cup **then**
8: Calculate center coordinates (*center_x*, *center_y*) of the bounding box
9: Get object identifier *object_id*
10: **if** *object_id* in *object_positions* **then**
11: **if** Position is stable **then**
12: Increment stability count *object_stability_count*[*object_id*]
13: **else**
14: Update position and reset stability count
15: *object_positions*[*object_id*] $\leftarrow$ (*center_x*, *center_y*)
16: *object_stability_count*[*object_id*] $\leftarrow 1$
17: **end if**
18: **else**
19: Initialize position and stability count
20: *object_positions*[*object_id*] $\leftarrow$ (*center_x*, *center_y*)
21: *object_stability_count*[*object_id*] $\leftarrow 1$
22: **end if**
{if the object has been in the same position for 8 consecutive frames}
23: **if** *object_stability_count*[*object_id*] ≥ 8 **then**
24: **if** *center_x* $<$ *screen_center_x* $-$ *central_position_threshold* **then**
25: *position* $\leftarrow right$
26: **else if** *center_x* $>$ *screen_center_x* $+$ *central_position_threshold* **then**
27: *position* $\leftarrow left$
28: **else**
29: *position* $\leftarrow center$
30: **end if**
31: **if** *position* $== center$ **then**
32: Output audio instruction: "Hold your *object_id* in the current position and wait for the next instruction"
33: Stop video stream and return success
34: **else**
35: Output audio instruction: "Move your *object_id* *position* a bit"
36: **end if**
37: Reset stability count *object_stability_count*[*object_id*] $\leftarrow 0$
38: **end if**
39: **end if**
40: **end for**
41: **if** exit condition met **then**
42: Break
43: **end if**
44: **end while**
45: Stop video stream and clean up

B.5 Code for Robot Gripper

```
from time import sleep
from megapi import MegaPi

class Gripper:
    def __init__(self, usb_port='/dev/ttyUSB0', motor_port=4,
        ↪ motor_speed=100, run_time=1):
        """
        Initialize the gripper.
        :param usb_port: Specify the USB-port connection to the
            ↪ MegaPi.
        :param motor_port: specify the motor port on MegaPi.
        :param motor_speed: Specify the speed of the motor.
        :param run_time: Specify the time (in seconds) to run the
            ↪ motor for open/close operations.
        """
        self.bot = MegaPi()
        self.bot.start(usb_port)
        self.motor_port = motor_port
        self.motor_speed = motor_speed
        self.run_time = run_time

        # Ensure the motor is stopped initially.
        self.bot.motorRun(self.motor_port, 0)
        sleep(1) # Wait for the MegaPi to initialize properly.

    def release(self):
        """
        Open the gripper.
        """
        self.bot.motorRun(self.motor_port, -(self.motor_speed))
        sleep(self.run_time)
        self.stop()
        return 1

    def grab(self):
        """
        Close the gripper.
        """
        self.bot.motorRun(self.motor_port, self.motor_speed)
        sleep(self.run_time)
        self.stop()
        return 1
```

```
    def stop(self):
        """
        Stops any gripper motion.
        """
        self.bot.motorRun(self.motor_port, 0)
        sleep(1) # Pause to allow the motor to fully stop.

    def close(self):
        """
        Closes the serial connection and cleans up.
        """
        self.bot.close()
        self.bot.exit(0, 0)
```

B.6 Raspberry Pi GPIO Control

```
import time
import gpiod

LED_GREEN_PIN = 27
LED_RED_PIN = 17

def Led_green_ON(LED_GREEN_PIN):
   chip = gpiod.Chip('gpiochip4')
   led_line_green = chip.get_line(LED_GREEN_PIN)
   led_line_green.request(consumer="LED", type=gpiod.
      ↪ LINE_REQ_DIR_OUT)
   led_line_green.set_value(1)

def Led_green_OFF(LED_GREEN_PIN):
   chip = gpiod.Chip('gpiochip4')
   led_line_green = chip.get_line(LED_GREEN_PIN)
   led_line_green.request(consumer="LED", type=gpiod.
      ↪ LINE_REQ_DIR_OUT)
   led_line_green.set_value(0)
   time.sleep(1)
   led_line_green.release()

### Check if the water bottle is placed at the water dispenser
if (Detect_Bottle()):
    print("Detect␣bottle␣at␣the␣water␣dispenser...")
    chip = gpiod.Chip('gpiochip4')
    led_line_red = chip.get_line(LED_RED_PIN)
```

```
    led_line_red.request(consumer="LED", type=gpiod.
        ↪ LINE_REQ_DIR_OUT)
    led_line_red.set_value(1)

    # Start filling the bottle (dispensing hot water)
    play_audio_vlc("filling_bottle.mp3")

    led_line_red.set_value(0)
    led_line_red.release()
else:
    print("No␣bottle␣at␣the␣water␣dispenser!")
```

B.7 Shell Script Implementation

```
#!/bin/bash

# SSH into the Ubuntu VM to control the Robot Arm (Reset position
    ↪ )
ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_reset.py
logout
EOF
echo "Robot␣arm␣reset␣successful"

# Function to handle cleanup
cleanup() {
    echo "Cleaning␣up..."
    python3 water_dispenser_led.py OFF
    echo "LED␣cleanup␣complete.␣Exit!"
    exit 0 # Exit the script
}

# Catch the SIGINT (Ctrl+C), calling cleanup
trap cleanup SIGINT

# Setup holding state of water dispenser (Green LED ON)
python3 water_dispenser_led.py ON
echo "Water␣dispenser␣is␣ready..."

while true; do

    # Detect water bottle and then grab
```

```
    echo "Start␣detecting␣the␣location␣of␣water␣bottle..."
    usb_port=$(ls /dev/ttyUSB*)
    echo "USB␣port:␣$usb_port"
    camera1_index=$(v4l2-ctl --list-devices | grep -A 1 'XWF␣1080P
        ↪ ' | tail -n 1)
    echo "Camera␣1␣Index:␣$camera1_index"
    camera2_index=$(v4l2-ctl --list-devices | grep -A 1 'USB2.0
        ↪ _CAM2' | tail -n 1)
    echo "Camera␣2␣Index:␣$camera2_index"
    python3 detect_grab.py --webcam1 $camera1_index --webcam2
        ↪ $camera2_index --usb_port $usb_port

    # SSH into the Ubuntu VM to control the Robot Arm (forward to
        ↪ the water dispenser)
    ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_forward.py
logout
EOF
    echo "Robot␣arm␣is␣moving␣toward␣the␣water␣dispenser..."

    # Start the water dispenser
    python3 check_bottle_led.py

    # SSH into the Ubuntu VM to control the Robot Arm (back to the
        ↪  user)
    ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_back.py
logout
EOF
    echo "Robot␣arm␣is␣moving␣back␣to␣the␣user..."

    # Release the water bottle
    python3 release_bottle.py --usb_port $usb_port
    echo "Water␣bottle␣is␣released..."
    echo "Finish"

    sleep 60
done
```