

SMART COMPANION ROBOT FOR ELDERLY CARE

BY
TAN KOK FU

A REPORT
SUBMITTED TO
Universiti Tunku Abdul Rahman
in partial fulfillment of the requirements
for the degree of
BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMMUNICATIONS
AND NETWORKING
Faculty of Information and Communication Technology
(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

© 2025 Tan Kok Fu. All rights reserved.

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of **Bachelor of Information Technology (Honours) Communications and Networking** at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr Teoh Shen Khang who has given me this bright opportunity to engage in a Robotic project. It is my first step to establish a career in robot development field. Dr. Teoh always provided guidance and advice whenever I faced problems or issues in my project. A million thanks to you.

Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

With the aging population and an increasing number of elderly people living alone, their well-being, health, and safety have been a big concern. Traditional monitoring systems such as CCTV cameras and panic buttons are at a disadvantage since they are fixed and dependent on human initiation. This project proposes the use of a multipurpose companion robot based on TurtleBot3 with autonomous movement integration, AI-based fall detection, live monitoring, and voice control. The robot will move autonomously throughout the home, continuously keeping an eye on the health of the elderly via an AI-driven camera system to detect falls, inactivity, or distress. It will also provide voice companionship and reminders to alleviate loneliness and improve mental health. A basic web interface will enable caregivers to remotely keep an eye on their loved ones, watch live video streams, and receive emergency alerts. The project involves creating and deploying deep learning-based fall detection models, speech recognition for two-way communication, and IoT integration for facilitating seamless data transmission. By integrating AI, IoT, and robotics, the project aims to develop an intelligent, cost-effective, and scalable solution that enhances elderly care, allows real-time safety monitoring, and facilitates emotional interaction, thus bridging the gap between technology and elderly care.

Area of Study: Robotics AI

Keywords: Voice Interaction, Autonomous Navigation, User Interface, Real Time Fall Detection, IoT

TABLE OF CONTENTS

TITLE PAGE	i
COPYRIGHT STATEMENT	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	xiv
LIST OF ABBREVIATIONS	xvi
CHAPTER 1 INTRODUCTION	1
1.1 Problem Statement and Motivation	2
1.2 Objectives	3
1.2.1 To Achieve Autonomous Home Navigation and Mapping	3
1.2.2 To Integrate a Real Time Fall Detection System	3
1.2.3 To Develop an AI-Driven Voice Interaction	3
1.2.4 To Develop a Web-Based Remote Monitoring Interface	3
1.3 Project Scope and Direction	4
1.4 Contributions	5
1.5 Background Information	6
1.6 Report Organization	7
CHAPTER 2 LITERATURE REVIEW	8
2.1 Review of Technologies	8
2.1.1 Hardware Platform	8
2.1.2 Firmware/OS	9
2.1.3 Programming Language	10
2.1.4 Algorithm	11
2.1.5 Summary of the Technologies Review	14
2.2 Previous Works on Companion Robots for Elderly	15
2.2.1 Follow Me: A Personal Robotic Companion System for the Elderly	15
2.2.2 PARO Therapeutic Robot	17

2.2.3	ElliQ	18
2.2.4	Comparison between previous works and proposed works	20
2.3	Chapter Summary	22
CHAPTER 3 SYSTEM MODEL (FOR RESEARCH-BASED PROJECT)		23
3.1	Methodology	23
3.2	System Design Diagram	25
3.3	Development Tools And Communication Interfaces	27
3.4	Timeline	29
3.4.1	FYP 1 Timeline	30
3.4.2	FYP 2 Timeline	31
3.5	Estimated Cost	
CHAPTER 4 SYSTEM DESIGN		32
4.1	System Block Diagram	32
4.2	System Components Specifications	34
4.2.1	Hardware	34
4.2.2	Firmware	41
4.2.3	Software/Framework	43
4.3	Circuits and Components Design	45
4.4	System Components Interaction Operations	47
4.4.1	Overall System Design	47
4.4.2	Autonomous Navigation	48
4.4.3	Real Time Fall Detection	49
4.4.4	AI Driven Voice Interaction	50
4.5	System Architecture Diagram	51
4.6	System Workflow	52
4.6.1	Overall System Workflow	52
4.6.2	SLAM Navigation Workflow	54
4.6.3	Real Time Fall Detection Workflow	54
4.6.4	AI-Driven Voice Interaction Workflow	54
4.7	Chapter Summary	55

CHAPTER 5	SYSTEM IMPLEMENTATION (FOR DEVELOPMENT- BASED PROJECT)	56
5.1	Hardware Setup	56
5.1.1	Robot Platform Setup	56
5.1.2	Network Infrastructure	58
5.2	Software Setup	60
5.2.1	PC Setup	60
5.2.2	SBC Setup	61
5.2.3	OpenCR Setup	63
5.2.4	Raspberry Pi Camera Setup	64
5.2.5	Pan Tilt Servo Setup	66
5.2.6	Audio and Speaker Setup	69
5.2.7	Telegram Setup	71
5.3	Setting and Configuration	74
5.3.1	System Launch and Robot Bringup	74
5.3.2	SLAM Navigation and Mapping	76
5.3.3	Fall Detection AI Model	82
5.3.4	STT/TTS Model	88
5.3.5	Web Interface	92
5.3.6	Telegram Bot	98
5.4	System Operation (with Screenshot)	102
5.5	Implementation Issues and Challenges	111
5.6	Concluding Remark	113
CHAPTER 6	SYSTEM EVALUATION AND DISCUSSION	114
6.1	System Testing and Performance Metrics	114
6.1.1	Fall Detection Performance	114
6.1.2	Navigation Performance	118
6.1.3	STT/TTS Performance	119
6.1.4	Web UI Performance	121
6.1.5	System Reliability	121
6.1.6	Meditation Scheduling Performance	121

6.1.7	Telegram Bot Performance	122
6.2	Testing Setup and Result	123
6.3	User Opinion Survey	124
6.4	Project Challenges	128
6.5	Objectives Evaluation	129
6.6	Concluding Remark	130
CHAPTER 7	CONCLUSION AND RECOMMENDATION	131
7.1	Conclusion	131
7.2	Recommendation	132
REFERENCES		133
APPENDIX		A-1
POSTER		B-1

LIST OF FIGURES

Figure Number	Title	Page
Figure 2.1.1	TurtleBot3 Burger	8
Figure 2.2.1	Follow Me: Robot Car Platform	15
Figure 2.2.2	Flow Diagram of System Connection of Follow Me	16
Figure 2.2.3	PARO Therapeutic Robot	17
Figure 2.2.4	Testing on PARO	18
Figure 2.2.5	ElliQ	18
Figure 3.1.1	Agile Methodology	23
Figure 3.2.1	Overall System Design Diagram	25
Figure 3.3.1	Oracle VM VirtualBox	27
Figure 3.3.2	MobaXterm	27
Figure 3.3.3	Telegram Bot	28
Figure 3.3.4	Visual Studio Code	28
Figure 3.3.5	Simple Commander API	29
Figure 3.4.1	FYP 1 Timeline and Milestone	29
Figure 3.4.2	FYP 2 Timeline and Milestone	30
Figure 4.1.1	System Block Diagram	32
Figure 4.2.1	TurtleBot3 Burger	34
Figure 4.2.2	OpenCR 1.0	35
Figure 4.2.3	Raspberry Pi 3B+	37
Figure 4.2.4	LiDAR Sensor LDS-01	38
Figure 4.2.5	Raspberry Pi Camera Module v2	39
Figure 4.2.6	USB Microphone For Raspberry Pi	39
Figure 4.2.7	USB Speaker on Raspberry Pi	40
Figure 4.2.8	Pan Tilt Servo Kit for Camera	40
Figure 4.2.9	ROS2 Humble	41
Figure 4.2.10	Ubuntu	42
Figure 4.2.11	Rviz	43
Figure 4.2.12	Flask	43
Figure 4.2.13	Vosk	43

Figure 4.2.14	Whisper	44
Figure 4.2.15	Google Text to-Speech	44
Figure 4.3.1	Raspberry Pi Connections	45
Figure 4.4.1	Overall System Design Diagram	47
Figure 4.4.2	SLAM Navigation	48
Figure 4.4.3	Real Time Fall Detection	49
Figure 4.4.4	AI Driven Voice Interaction	50
Figure 4.5.1	Network Architecture Diagram	51
Figure 4.6.1	Overall System Activity Diagram	52
Figure 5.1.1	Raspberry Pi 3 GPIO Header	57
Figure 5.1.2	Ellie Front View	58
Figure 5.1.3	Ellie Back View	58
Figure 5.1.4	Network configuration setup	59
Figure 5.1.5	50-cloud-init.yaml file configuration	59
Figure 5.2.1	Setting Up of Virtual Machine	60
Figure 5.2.2	Ensure locale supporting UTF-8 exists	60
Figure 5.2.3	Ensure Ubuntu Universe Repository enabled	60
Figure 5.2.4	Add ROS 2 GPG key with apt	60
Figure 5.2.5	Add repository to sources list	61
Figure 5.2.6	Install ROS 2 packages	61
Figure 5.2.7	Install Gazebo	61
Figure 5.2.8	Install Cartographer	61
Figure 5.2.9	Install Navigation2	61
Figure 5.2.10	Install TurtleBot3 Packages	61
Figure 5.2.11	Setup ROS environment	61
Figure 5.2.12	Installing Ubuntu Server	62
Figure 5.2.13	Open network configuration file	62
Figure 5.2.14	Example of network configuration file settings	62
Figure 5.2.15	Install and Build ROS packages	63
Figure 5.2.16	USB Port Settings for OpenCR	63
Figure 5.2.17	Setting ROS Domain ID	63
Figure 5.2.18	Setting LDS Model	63
Figure 5.2.19	Install packages for OpenCR	63

Figure 5.2.20	Specifying the model for OPENCR	63
Figure 5.2.21	Download firmware and required loader for OpenCR	63
Figure 5.2.22	Upload firmware to OpenCR	63
Figure 5.2.23	Example of successful firmware upload	64
Figure 5.2.24	Successful libcamera build and installation process	66
Figure 5.2.25	Meson build configuration for libcamera	66
Figure 5.2.26	Successfully run camera node	66
Figure 5.2.27	Ros2 topic before launching camera node	66
Figure 5.2.28	Ros2 topic after launching camera node	66
Figure 5.2.29	Main Function of Pan Tilt Servo Setup	67
Figure 5.2.30	Driver Node Initialization	68
Figure 5.2.31	Teleop Node Key Handling	68
Figure 5.2.32	Output of arecord -l	69
Figure 5.2.33	Output of aplay -l	69
Figure 5.2.34	Volume Control through alsamixer	70
Figure 5.2.35	MicPub function	70
Figure 5.2.36	SpkSub function	71
Figure 5.2.37	Main Function of audio_bridge.py	71
Figure 5.2.38	Ros2 topic list before running audio node	72
Figure 5.2.39	Ros2 topic list after running audio node	72
Figure 5.2.40	Create Bot	73
Figure 5.2.41	Edit the Details	73
Figure 5.2.42	Example of Bot Page	73
Figure 5.2.43	Create Telegram bot command with functionalities	73
Figure 5.2.44	Telegram bot with AI reply function	74
Figure 5.3.1	Environment setup commands in the bashrc file (Turtlebot3)	75
Figure 5.3.2	robot_start.sh script	75
Figure 5.3.3	The tmux session after a successful launch, showing all active nodes	76
Figure 5.3.4	Environment setup commands in the .bashrc file (RemotePC)	76
Figure 5.3.5	Launch teleoperation on another terminal of Remote PC	77

Figure 5.3.6	Robot Mapping the environment	78
Figure 5.3.7	map.pgm and map.yaml	78
Figure 5.3.8	Final map.pgm after processing	79
Figure 5.3.9	Provide navigation goal	80
Figure 5.3.10	Goal Reached	80
Figure 5.3.11	Map to Pixel Conversion	81
Figure 5.3.12	Nav2 Simple Commander setup	81
Figure 5.3.13	Publish Initial Pose	81
Figure 5.3.14	Navigate to goal function	81
Figure 5.3.15	Flask API for Navigation Control	82
Figure 5.3.16	Save and get saved points	82
Figure 5.3.17	WebUI startup, subscribed to Nav2	82
Figure 5.3.18	Configuration for PoseNet model	83
Figure 5.3.19	Three-Frame Temporal Analysis	84
Figure 5.3.20	Real-time Fall Detection with PoseNet	84
Figure 5.3.21	fall_worker function in app.py	85
Figure 5.3.22	Initialization for Human FallDetection Model	85
Figure 5.3.23	Pose Analysis in the OpenPifPaf-based Model	86
Figure 5.3.24	Real-time Fall Detection with OpenPifPaf	86
Figure 5.3.25	Video Processing Pipeline	87
Figure 5.3.26	Optimized Frame Processing	87
Figure 5.3.27	Environment Variables	88
Figure 5.3.28	Telegram Alert Function	89
Figure 5.3.29	Ros2 Topic Integration	90
Figure 5.3.30	Precheck on RoS2 related topic	90
Figure 5.3.31	Hybrid STT Model Implementation	91
Figure 5.3.32	API Key Management & Rotation Algorithm	91
Figure 5.3.33	Voice Logging Database	91
Figure 5.3.34	Main Voice Interaction Pipeline	92
Figure 5.3.35	Wake Words Detection Algorithm	92
Figure 5.3.36	Example of bringup WebSocket of Turtlebot	94
Figure 5.3.37	Sample File Hierarchy	95
Figure 5.3.38	WebUI Integration	95

Figure 5.3.39	Sample Route to each pages	96
Figure 5.3.40	Home Page	97
Figure 5.3.41	STT/TTS Page	97
Figure 5.3.42	Control Page	97
Figure 5.3.43	Navigation Page	97
Figure 5.3.44	Meditation Page	98
Figure 5.3.45	Fall Detection Page	98
Figure 5.3.46	Telegram Bot Page	98
Figure 5.3.47	Telemetry Page	98
Figure 5.3.48	Telegram Bot Message Function	99
Figure 5.3.49	Telegram Status Function	99
Figure 5.3.50	Telegram Command Handling Flow	100
Figure 5.3.51	Telegram AI Chat Interaction	101
Figure 5.3.52	Telegram Polling Worker Function	102
Figure 5.4.1	Launching Turtlebot3 with ./robot_start.sh	103
Figure 5.4.2	Launching RemotePC with ./remote_start.sh	103
Figure 5.4.3	Home Page of WebUI	103
Figure 5.4.4	Control Page of WebUI	104
Figure 5.4.5	Alert Message Received in Telegram	104
Figure 5.4.6	Navigation Page of WebUI	105
Figure 5.4.7	Rviz after startup of map	105
Figure 5.4.8	Navigation In Progress	106
Figure 5.4.9	Goal Reached	106
Figure 5.4.10	Selection of Voice	106
Figure 5.4.11	Example of AI Conversation	106
Figure 5.4.12	STT/TTS Page	107
Figure 5.4.13	Fall Detection Test	107
Figure 5.4.14	Example of Fall Detection Test Result	108
Figure 5.4.15	Telemetry Dashboard	108
Figure 5.4.16	Meditation Schedule	109
Figure 5.4.17	Meditation Schedule Reminder from Telegram	109
Figure 5.4.18	Telegram Bot Management Page	110
Figure 5.4.19	Telegram Bot Start	111

Figure 5.4.20	Telegram Bot Commands	111
Figure 5.4.21	Telegram Bot /status command	111
Figure 5.4.22	Telegram Bot Interactive Chat	111
Figure 6.1.1	Result of test2.mp4 for both model (People falling)	116
Figure 6.1.2	Result of test1.mp4 for both model (People on floor)	116
Figure 6.1.3	Result of output.mp4 for PoseNet	116
Figure 6.1.4	Result of test1.mp4 for HumanFallDetection model (People on floor)	116
Figure 6.1.5	Navigation ETA	118
Figure 6.1.6	Navigation Successfully	118
Figure 6.1.7	Example of Vosk Failed	120
Figure 6.1.8	Example of Transcription	120

LIST OF TABLES

Table Number	Title	Page
Table 2.1.1	Summary of the Technologies Review	14
Table 2.2.1	Comparison between previous works and proposed works	20
Table 3.5.1	Estimated Cost	31
Table 4.2.1	Specifications of Turtlebot3	34
Table 4.2.2	Specifications of OpenCR 1.0	35
Table 4.2.3	Role of OpenCR 1.0 in TurtleBot3	36
Table 4.2.4	Specifications of Raspberry Pi 3B+	37
Table 4.2.5	Role of Raspberry Pi 3B+ in TurtleBot3	38
Table 4.2.6	Role of LiDAR Sensor LDS-01 in TurtleBot3	38
Table 4.2.7	Role of Raspberry Pi Camera Module v2 in TurtleBot3	39
Table 4.2.8	Role of USB Microphone For Raspberry Pi	39
Table 4.2.9	Role of USB Speaker on Raspberry Pi in TurtleBot3	40
Table 4.2.10	Pan Tilt Servo Kit for Camera in TurtleBot3	40
Table 4.2.11	Specifications of laptop	41
Table 4.2.12	Role of laptop	41
Table 4.3.1	Components Description and Connection	45
Table 5.1.1	Pan-Tilt Servo Kit Connection	56
Table 5.1.2	Summary of Hardware Components	57
Table 5.3.1	Navigation Points	80
Table 6.1.1	Confidence Score and Detection Result	115
Table 6.1.2	Additional Performance Metrics for Fall Detection Model	115
Table 6.1.2	Navigation Performance	118
Table 6.1.3	STT Performance Metrics	119
Table 6.1.4	Transcription Test Results	119
Table 6.1.5	WebUI Performance Table	121
Table 6.1.6	Meditation Scheduling Performance Metric	122
Table 6.1.7	Telegram Bot Performance Metrics	122
Table 6.3.1	Average Scores for User Opinion Survey	124

LIST OF ABBREVIATIONS

<i>WHO</i>	World Health Organization
<i>DOSM</i>	Department of Statistics Malaysia
<i>ESCAP</i>	Economic and Social Commission for Asia and the Pacific
<i>CCTV</i>	Closed-Circuit Television
<i>AI</i>	Artificial Intelligence
<i>SLAM</i>	Simultaneous Localization and Mapping
<i>NLP</i>	Natural Language Processing
<i>LLM</i>	Large Language Model
<i>ROS</i>	Robot Operating System
<i>LDS</i>	Laser Distance Sensor
<i>SBC</i>	Single Board Computer
<i>MCU</i>	Microcontroller unit
<i>OpenCR</i>	Open-source Control Module
<i>LiDAR</i>	Light Detection and Ranging
<i>IMU</i>	Inertial Measurement Unit
<i>VM</i>	Virtual Machine
<i>SSH</i>	Secure Socket Shell
<i>AMCL</i>	Adaptive Monte Carlo Localization
<i>HTTP</i>	Hypertext Transfer Protocol
<i>OS</i>	Operating System
<i>ML</i>	Machine Learning
<i>UI</i>	User Interface
<i>IOT</i>	Internet of Things
<i>IDE</i>	Integrated Development Environment
<i>STT</i>	Speech To Text
<i>TTS</i>	Text To Speech
<i>PWM</i>	Pulse Width Modulation

Chapter 1

Introduction

Southeast Asia faces a rapidly aging population, with Malaysia projected to see its elderly population of peoples aged above 65 doubles to 14.5% by 2044 from 7.2% on 2022 based on the analytics from DOSM [1]. Based on ESCAP, By 2050 estimated 996 million people in Asia-Pacific will be over 65, intensifying demand for sustainable elderly care solutions [2]. Traditional reliance on family caregiving is strained by modern work demands, leading to isolation, unaffordable professional care, and passive monitoring systems that lack intervention.

Individuals working overseas or in other metropolitan areas may have limited opportunities to visit, and it is hard to offer continuous care. This was a factor in rising cases of lonely death in Malaysia [3]. Elderly individuals living alone face critical risks such as falls, a leading cause of injury death, it often goes unreported due to unreliable emergency devices, while loneliness exacerbates mental and physical decline. Falls are a leading cause of injury among the elderly, yet conventional detection methods like wearable devices and static cameras have significant limitations [4]. Companion robots address these gaps by offering real-time fall detection, emergency alerts, and meaningful social interaction.

This project will be presenting a companion robot, which acts as a companion robot for elderly in home while at the same time serves as monitoring purpose. This project aims to utilizes the capabilities of a robot, to provide real time alerts and care towards elderly through detection and voice interaction which provides elderly with proper care, attention and love and improving their quality of life.

1.1 Problem Statement and Motivation

In 2020, approximately 10.3% of Malaysians aged 60 and above were found to be “empty nesters” [5]. It is driven by urban migration and crowded work hours; thus, elderly parents are often left behind in homes without immediate family support. Afterwards, older people are prone to increased vulnerability from loneliness, depression, and ill health, thus, it highlights the need for a more effective and interactive elderly care solution. Traditional monitoring solutions, such as static CCTV cameras and wearable devices present several limitations. Static cameras often have blind spots, while wearables rely on user cooperation, which may not be feasible during emergencies as it could delay emergency responses. According to World Health Organization (WHO) [6], falls are the second highest leading cause of unintentional injury death globally. Notably, 14.1% of Malaysian elderly were reported experiencing at least one fall per year, underscoring the need for more effective monitoring systems [7]. Besides, most empty nest elderly individuals lack daily social interaction, and thus they become lonely and even develop mental illnesses. The elderly individuals often need regular social interaction to prevent suffering from cognitive problems, including conditions like Alzheimer’s disease [8]. Currently available technologies cannot provide substantial interaction or companionship. Additionally, caregivers and family members are restricted in monitoring their older relatives remotely since there are no embedded, simple-to-use interfaces offering real-time information and control features from IoT devices.

To overcome these constraints, this project proposes designing a mobile companion robot that is equipped with real-time mapping and navigation functionality to travel throughout the house and overcome static CCTV cameras limitations. The real-time video analysis and fall detection algorithms-loaded mobile robot can provide more timely and accurate alerts to improve safety performance. Enhanced with a web-based interface, this solution aims to enhance elderly care through continuous monitoring, immediate emergency response, and social interaction. Hence, the quality of life for the elderly can be improved and offer peace of mind to their caregivers. The motivation stands out from the needs of solving the future needs of elderly care issues since the rise of the population of elderly in the future and current technologies lacks sufficient or effective care.

CHAPTER 1

1.2 Objectives

The primary objective of this project is to develop a mobile companion robot equipped with intelligent systems to assist elderly individuals in a modern home environment. The specific objectives are:

1.2.1 To Achieve Autonomous Home Navigation and Mapping

The first objective is to develop a mobile robot with the capability to navigate and map home environments autonomously using Simultaneous Localization and Mapping (SLAM) techniques. This will enable the robot to dynamically move through different rooms, with complete monitoring and fewer blind spots.

1.2.2 To Integrate a Real Time Fall Detection System

Besides, this project aims to integrate real-time fall detection based on computer vision and deep learning methodologies such as pose estimation algorithms to accurately identify fall detected on the real-time video streaming. In case of detecting a fall, the system will immediately alert caregivers or relatives to allow timely assistance. For emergency situations, it aims to reduce false positives and late intervention.

1.2.3 To Develop an AI-Driven Voice Interaction

Other than that, this project aims to develop an AI-driven voice interaction module by employing speech recognition technologies, Natural Language Processing (NLP) and Large Language Model (LLM) to engage in meaningful conversations with the elderly using the mobile robot. This element of the project is seeking to provide companionship, respond to inquiries, and provide reminders of medication and routines for daily requirements with a view to providing emotional advantages.

1.2.4 To Develop a Web-Based Remote Monitoring Interface

The last goal of the project is to create a simple web interface by which users can monitor robot activity in real-time. The interface will also display live video feeds, battery status, and

emergency notifications and will offer facilities for manual control of robot's navigation and interaction parameters based on user's preference.

1.3 Project Scope and Direction

The scope of this project encompasses the design and development of a mobile companion robot specifically tailored for elderly care within domestic environments. The robot will leverage robotics, sensors, real-time video streaming and analysis and real-time voice interaction with humans to address the issues encountered currently such as social isolation, falling risks, and the need for constant monitoring. This project focuses on designing, implementing and evaluating a prototype mobile companion robot. The hardware foundation for these projects will be the TurtleBot3 platform with the ROS2 framework, augmented with a camera module, speaker, microphone, and a remote PC for processing and communication. These modules were the foundations of the companion robot, enabling the robot functionalities such as real-time video streaming, voice interaction, and easy communication with the user.

On the software front, this project will implement SLAM techniques for robot control, navigation and localization within house environments. AI-based fall detection algorithms will be integrated with the camera system for facilitating real-time processing and immediate alerting in case of emergencies. At the same time, the robot will also have an AI-driven voice interaction module using speech recognition, NLP and LLM technologies to engage in meaningful conversation and communication with humans, while being a companion at the same time. In addition, an easy-to-use web interface will also be developed to offer real-time data visualization and control so that caregivers and family members can monitor robot status, be notified, and remotely interact with the system. The final product of this project will be an operational prototype demonstrating the integration of the systems with the objective of enhancing the quality of life of the elderly and providing them peace of mind as far as the caregivers are concerned through innovative technology solutions.

1.4 Contributions

The planned mobile companion robot project signifies enormous progress in elderly care and to convert robotics into elderly care by solving crucial issues such as social isolation, safety, and the need for continuous monitoring. With an integration of revolutionary technologies like real-time voice talk, fall detection, and auto-navigation, this robot is bound to provide not only functional support but also emotional reassurance to lonely old age individuals, all the while subjecting old age individuals to full-time surveillance and reducing the possibility of accidents going unnoticed. In addition, the robot companion will serve as an interactive communication device that will enable elderly individuals to communicate and receive reminders for critical activities such as medication and hydration. AI-powered natural speech recognition will enable the robot to respond naturally, thereby making the interaction more engaging and meaningful. The robot will further have an incident detection and emergency alert system. Through fall detection, the major cause of fatal and nonfatal trauma in the elderly [9] could be addressed, the system will notify caregivers in real-time, enabling faster reaction to emergencies. This significantly enhances the well-being and security of elderly individuals, reducing the risks of being alone. Further, this project presents a low-cost and scalable alternative to traditional caregiving services.

Unlike full-time caregivers, the robotic solution will be a cost-effective way of handling elderly care compared to full-time caregivers, which in most instances, will prove impractical for many families. Utilizing AI and IoT provides flexibility to accommodate various household settings and caregiving needed. By implementing this solution, the project can enhance the welfare of elderly patients while providing their caregivers with a working, remote monitoring system. The significance of this project is that it presents a whole solution strategy towards elderly care. Combining robots, artificial intelligence, speech interaction and intuitive dialogue, the social robot is a whole solution in a multi-aspect solution to enhance the life of the elderly person and make the caregivers' lives easier with peace of mind concerns. Brief and simply stated, this project is not merely an academic exercise but a genuine attempt that can potentially make a meaningful contribution to society.

1.5 Background Information

The global demographic shift towards an aging population has intensified the demand for innovative solutions in elderly care [10]. Traditional models of caregiving are typically not equipped to deal with the dual demands of the aged, particularly the empty nest elderly. This has necessitated a drive towards supportive technologies, and companion robots have emerged as a viable solution to improve the quality of life among the elderly.

Companion robots are designed to provide not just practical assistance but also emotional support to older adults [11]. These robots can perform tasks such as reminding users to take medications, assisting with mobility, and facilitating communication with family members and healthcare providers. Other than that, They also offer companionship, which plays a critical role in the avoidance of feelings of loneliness and social isolation, a prevalent condition for the elderly. Companion robot interactions have been found to lead to improved mood and cognitive performance among older adults [12].

The development of companion robots integrates innovations in several fields like robotics, artificial intelligence (AI), natural language processing (NLP), and human-computer interaction [13]. For example, Baby seal-sized robot Paro is used in care settings to soothe dementia patients. Similarly, ElliQ and Buddy robots are designed to converse with users, encourage exercise, and monitor health metrics [14].

In short, the combination of age profiles and technological developments has created possibilities for robots to become centre stage in caring for seniors. Combining helpful assistance with reassurance, the robots can stimulate enhanced autonomy and enhanced well-being among the elderly and burden caregivers less. Continued research and development in the area is necessary to overcome current challenges and realize the full potential of companion robots in supporting the ageing population.

1.6 Report Organization

This report is structured into seven chapters that guide the reader through the project from conception to conclusion. Chapter 1 introduces the project's background, objectives, and scope. Chapter 2 presents a literature review of relevant technologies and a comparative analysis of existing companion robots. Chapter 3 details the system's methodology, outlining the overall architecture and design through use case and activity diagrams. Chapter 4 provides an in-depth look at the system's technical design, including component specifications and interaction diagrams, offering a blueprint for replication. Chapter 5 describes the practical implementation, covering the hardware and software setup, configuration, and operational workflow. Chapter 6 evaluates the system's performance through rigorous testing, presenting the results and discussing the challenges faced. Finally, Chapter 7 concludes the report by summarizing the key achievements and providing recommendations for future development.

Chapter 2

Literature Review

2.1 Review of the Technologies

2.1.1 Hardware Platform

The core of the companion robot is the TurtleBot3 Burger, a versatile and compact mobile robot platform. It serves as the primary robotic platform for navigation and interaction in this project, providing a reliable foundation for building the system's autonomous capabilities.

TurtleBot3 Burger

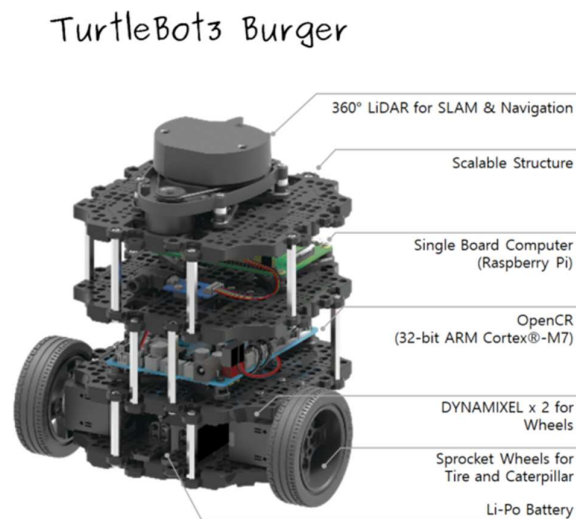


Figure 2.1.1: TurtleBot3 Burger [15]

The TurtleBot3 Burger is a ROS2-compatible mobile robot that provides a comprehensive set of features for autonomous navigation, perception, and interaction. The platform is equipped with an LDS-01 LiDAR, IMU, and motor controllers, all of which are essential for enabling Simultaneous Localization and Mapping (SLAM). This makes TurtleBot3 Burger an ideal choice for navigation and autonomous mapping tasks.

The LDS-01 LiDAR provides 360° environmental scanning, delivering real-time data crucial for obstacle detection and the creation of a 2D map of the robot's surroundings. The IMU offers orientation tracking, helping the robot maintain stability and accurately track its position during movement. Additionally, the platform incorporates an Arduino-based

CHAPTER 2

microcontroller for low-level motor control, ensuring precise movement and seamless communication with the sensors.

The TurtleBot3 Burger is specifically designed for research and development in ROS2, making it a preferred choice for robotics projects that require modularity, flexibility, and compatibility with a variety of sensors and actuators. Its open-source nature allows for easy integration with additional components, while its compact design makes it suitable for both indoor and real-world testing scenarios.

In this project, the TurtleBot3 Burger will serve as the mobile base for the robot, enabling it to autonomously navigate and perform tasks such as fall detection and human-robot interaction. Its ability to run SLAM algorithms, coupled with its ROS2 support, forms the foundation for the robot's autonomous navigation and control system.

2.1.2 Firmware/OS

1. Ubuntu Server 22.04 LTS

Ubuntu Server 22.04 LTS (Jammy Jellyfish) is installed on the Raspberry Pi to act as the primary operating system for the robot. It provides a lightweight, command-line-based Linux environment optimized for ARM processors. This ensures efficient resource usage while still supporting essential software such as ROS2 Humble and device drivers. The server edition is suitable for headless operation, enabling remote access and integration with robotics frameworks [16].

2. Ubuntu Desktop 22.04 LTS

Ubuntu Desktop 22.04 LTS is installed on the development computer. It offers a full graphical user interface (GUI) and a stable Linux environment for coding, simulation, and debugging. It also supports development tools such as ROS2, Python, and web frameworks. Ubuntu Desktop is widely used in robotics research and provides seamless compatibility with ROS2 Humble, making it an ideal choice for this project [17].

2.1.3 Programming Language

1. Python

The primary programming language used in this project is Python. Python is a high-level, interpreted language that emphasizes readability and simplicity. It uses indentation-based syntax, which makes it easier for developers to write clear and structured code compared to other languages such as C++ or Java [18].

Python is particularly suitable for robotics projects because of its extensive library support for various client libraries such as NumPy, OpenCV, Flask, and ROS2 and its ability to rapidly prototype algorithms for tasks such as navigation, image processing, and data communication. Compared with C++, which offers higher performance and direct hardware control, Python provides faster development cycles and easier debugging, which is crucial in research and prototyping environments [19]. While Java is widely used for enterprise systems and mobile applications, it lacks the same level of robotics-specific libraries and community support that Python provides [20].

In this project, Python is used for developing ROS2 nodes, implementing algorithms, and integrating the web interface using Flask. The combination of ease of use, cross-platform compatibility, and large open-source community makes Python the most practical choice for this robotics project.

2. JavaScript

JavaScript plays a crucial role in the development of the Web UI for the robot. The web interface is built using JavaScript, HTML, and CSS, providing an intuitive platform for the user to interact with the robot. JavaScript is used for creating dynamic and responsive elements on the webpage, such as buttons for controlling the robot's movements, displaying real-time sensor data, and receiving alerts from the fall detection system. This integration allows the user to monitor the robot's status remotely, set navigation goals, and view camera feeds.

JavaScript's widespread usage in web development and its ability to handle asynchronous communication make it the ideal language for the real-time features required in the web interface, such as the display of live video feeds and sensor data updates.

3. HTML

HTML is used for the structure of the Web UI, defining the layout and ensuring a responsive design that adapts to different screen sizes. It works alongside JavaScript to display data and provide interactive elements. The Web UI enables users to control the robot and receive live updates on its status, creating a seamless interaction between the user and the robot.

2.1.4 Algorithm

1. Robotics Algorithms

The project employs Simultaneous Localization and Mapping (SLAM) as the core robotics algorithm to allow the robot to build a map of an unknown environment while simultaneously estimating its own position within that map. In ROS2 Humble, SLAM is implemented through packages such as `slam_toolbox`, which provides lifelong mapping and localization functionalities [21]. SLAM enables the robot to navigate autonomously without relying on pre-existing maps, making it suitable for dynamic and unstructured environments. The algorithm works by combining data from sensors (LiDAR, IMU, odometry) to construct a two-dimensional occupancy grid map. This map is continuously updated as the robot explores, allowing real-time decision-making for navigation and obstacle avoidance.

Once a reliable map is generated, the project utilizes the Navigation2 Simple Commander API to control robot movement across the mapped environment. The Simple Commander API provides a Python interface to send navigation goals by specifying x, y coordinates and orientation within the SLAM-generated map. This allows the robot to autonomously move to precise target locations on command, supporting tasks such as point-to-point navigation and return-to-home functionality. By combining SLAM with the Simple Commander API, the robot achieves both exploration and goal-driven navigation in a seamless manner.

2. AI Models for Fall Detection

To enhance the system's safety monitoring capability, the project integrates AI-based human fall detection models. Two open-source deep learning projects are adopted:

- i. Ambianic.ai Fall Detection [22]

This model uses PoseNet for single-frame pose estimation, followed by heuristics over 2-3 consecutive frames to detect falls. It applies rules based on torso tilt, height drop, and aspect ratio changes between frames to identify falls. It is optimized for edge deployment on low-resource devices like Raspberry Pi, ensuring privacy-preserving, real-time fall detection. It uses pose estimation and object detection frameworks which based on TensorFlow Lite and EdgeML to identify abnormal body postures and sudden movements that indicate a fall. While the model is simple and effective for single-person, low-clutter environments, it may not handle more complex scenarios with multiple people or occlusions as effectively. While this model is lightweight, it is optimized to be run on edge devices such as Raspberry Pi.

ii. Human Fall Detection (taufeeque9) [23]

This model uses a pose estimation pipeline to detect falls based on human body keypoints. It employs OpenPifPaf for pose extraction and an LSTM classifier to learn temporal features from keypoints. The model extracts five key features from body pose: torso tilt, height drop, aspect ratio of the body bounding box, vertical velocity, and center-of-mass displacement. These features are fed into an LSTM that classifies whether a fall has occurred. The model is highly suitable for multi-camera and multi-person scenarios, offering good performance on UP-Fall dataset (F1 score $\approx 92.5\%$). However, it requires significant computational resources, including a GPU, due to the heavy pose inference.

Together, these AI models add an intelligent perception layer to the project, providing health and safety monitoring in addition to navigation tasks.

3. Speech Processing Models (STT/TTS)

Voice interaction is enabled through offline speech recognition (STT) and cloud-based speech synthesis (TTS):

i. Speech-to-Text (STT):

The project uses both Vosk and Whisper models for offline speech recognition.

- Vosk is a lightweight, Kaldi-based toolkit designed for real-time speech recognition on edge devices. It supports multiple languages and requires minimal computational resources, making it highly suitable for Raspberry Pi [24].
 - Whisper, developed by OpenAI, is a transformer-based model that offers robust transcription even in noisy environments. While Whisper requires more computational power than Vosk, it provides higher accuracy in recognizing natural language commands [25].
- ii. Text-to-Speech (TTS):
- The project employs Google Text-to-Speech (gTTS), a free and open-source Python library that interfaces with Google's TTS service. It converts text into natural-sounding speech in real time, providing auditory feedback to users [26].

By combining Vosk, Whisper, and gTTS, the system achieves a balance between offline robustness and natural voice interaction, without additional cost.

4. Communication and Integration Algorithm

For communication between the robot backend (ROS2) and the web frontend (Flask), the project uses a ROS2 publish–subscribe mechanism integrated into Flask routes. Flask acts as the middleware for bridging ROS2 topics with HTTP endpoints. Through this design:

- Flask routes subscribe to ROS2 topics (e.g., battery status, sensor readings) and deliver real-time updates to the web interface.
- HTTP POST requests from the web interface are translated into ROS2 publish actions (e.g., movement commands, start/stop navigation).

This approach enables seamless communication between users and the robot via a web browser, avoiding the complexity of MQTT or external brokers. Flask is lightweight, Python-based, and highly compatible with ROS2 libraries, making it the optimal choice for integration [27].

2.1.5 Summary of the Technologies Review

Table 2.1.1: Summary of the Technologies Review

Category	Technology/Tool	Purpose in Project	Reason for Selection
Operating System	Ubuntu Server 22.04 (Raspberry Pi) Ubuntu Desktop 22.04 (Remote PC)	Provides a stable Linux environment for robot operation and software development.	Widely supported in robotics, compatible with ROS2, and lightweight for embedded devices like Raspberry Pi.
Programming Language	Python JavaScript HTML	Python is used for ROS2 nodes, Flask backend, AI model integration, and control scripts. JavaScript and HTML are used for the web interface development.	Python: High readability, extensive library ecosystem (NumPy, OpenCV, Flask, ROS2 libraries), and rapid prototyping. JavaScript: Enables dynamic and interactive web interfaces. HTML: Provides structure for the web interface, ensuring responsive design.
Robotics Algorithm	SLAM (via slam_toolbox) + Navigation2 Simple Commander API	Builds a map and localizes the robot; enables sending navigation goals via (x,y) coordinates.	Allows exploration and autonomous point-to-point navigation with precise location targeting.
AI Models	Ambianic.ai Fall Detection Human Fall Detection (CNN-based)	Detects human falls in real time for safety monitoring.	Lightweight deep learning models optimized for edge devices, free and open-source implementations.
Speech Processing	Vosk (STT) Whisper (STT) Google TTS	Recognizes spoken commands offline and provides natural-sounding voice feedback.	Vosk is lightweight and suitable for Raspberry Pi; Whisper provides higher accuracy; Google TTS is free.
Communication Layer	Flask + ROS2 Topics (Publish/Subscribe)	Bridges robot backend (ROS2) with web interface for monitoring and control.	Lightweight, Python-based, seamless integration with ROS2; enables real-time updates and HTTP interaction.

2.2 Previous Works on Companion Robots for Elderly

2.2.1 Follow Me: A Personal Robotic Companion System for the Elderly [28]

The article "Follow Me: A Personal Robotic Companion System for the Elderly" by Rong Wang, Zhiqi Shen, Huiguo Zhang, and Cyril Leung addresses the development of an intelligent robotic companion to assist elderly individuals in their daily lives. The article identifies the increasing number of elderly individuals who live alone and the resulting dangers of social isolation and impaired physical mobility. To surmount these challenges, the authors propose a robot that integrates user-following ability, speech recognition, and speech synthesis to provide companionship and assistance. The system aims to enhance the quality of life for elderly people by ensuring that they are not lonely and can easily communicate with the robot for assistance.

One of the most significant features of the proposed system is its user-following capability, in which the robot can follow an elderly individual automatically. This is achieved through the Kinect sensor, which utilizes skeleton tracking, depth sensing, and motion detection to detect and track a particular user. The robot has two modes of operation: Auto-follow Mode, in which it follows the user continuously using movement detection, and Voice Control Mode, in which users can call or command the robot using voice commands. The dual-mode operation provides flexibility in interacting with the user and makes the robot suitable for varying levels of mobility and environments.

Furthermore, it employs voice recognition and speech generation for interaction between the robotic system and the elderly user. With Microsoft's Text-to-Speech API, the robot can provide responses, reminders, and alerts through speech to assist elderly individuals in organizing everyday tasks. Even though the feature reduces loneliness by adding a minimal amount of interaction, its conversation feature is still limited to pre-defined responses, hence not engaging elderly individuals into a more tangible conversation.



Figure 2.2.1: Follow Me: Robot Car Platform [28]

A notable aspect of the study is the emphasis on affordability. The robot is designed using cost-effective hardware components, making it accessible to a wider range of elderly users. By

utilizing a simple toy car platform as Figure 2.2.1 and equipped with a Kinect sensor, the study demonstrates that functional companion robots can be developed without requiring expensive robotic hardware. However, while affordability is an advantage, the system's hardware constraints limit its ability to integrate advanced functionalities such as autonomous room navigation and real-time health monitoring.

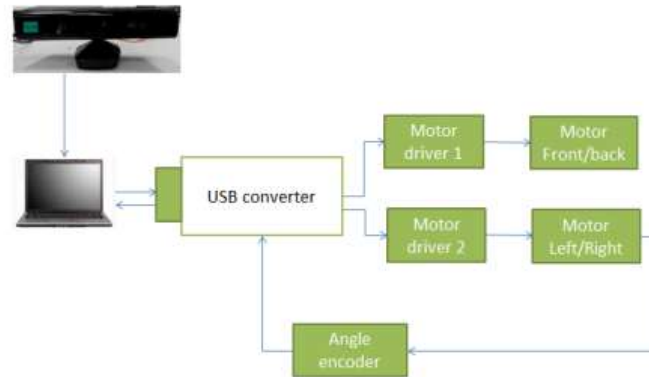


Figure 2.2.2: Flow Diagram of System Connection of Follow Me [28]

The study briefly describes the hardware connection and system flow but does not provide a detailed schematic or hardware figure. Based on Figure 2.2.2, the flow of the system suggests that the Kinect sensor is responsible for tracking movement and depth perception, while the voice recognition module enables user interaction. Although the research outlines the overall structure and operational process, a clearer depiction of hardware integration would have provided a more comprehensive understanding of the system's architecture and potential limitations.

Overall, the study provides a valuable foundation for understanding the potential of companion robots in elderly care, demonstrating how movement tracking and voice interaction can enhance user experience. However, its limitations in autonomous navigation, emergency detection, and remote monitoring highlight the need for a more comprehensive solution. Our project aims to build upon these concepts by integrating AI-driven safety monitoring, intelligent companionship, and remote accessibility, offering a more robust and effective approach to elderly care through

2.2.2 PARO Therapeutic Robot

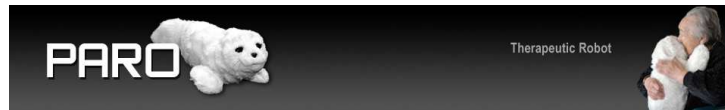


Figure 2.2.3: PARO Therapeutic Robot [29]

PARO is a socially assistive robot designed specifically for providing therapy by interaction to, for example, elderly with cognitive impairments such as dementia [30]. It was developed by Takanori Shibata of the National Institute of Advanced Industrial Science and Technology (AIST) in Japan as a replica baby harp seal that has tactile sensors, microphones, and actuators so that it can react to touch, sound, and light and show behaviour like a real pet.

There have been a variety of investigations regarding the usefulness of PARO in practice. A systematic review concluded that PARO can be an effective non-pharmacological approach in improving behavioural and psychological symptoms of dementia with potential reduction in medication and improvement in sociability among older people [31]. In another review, the benefits of PARO were cited as reduction of negative emotions and behavioural symptoms, promotion of social interaction, and improvement in positive mood and quality of care experience [32].

Further research confirms the efficacy of PARO in facilitating activity engagement, relaxation, and mood improvement in dementia patients, suggesting its efficacy at any stage of the disease [33]. A pilot study also demonstrated that interaction with PARO can decrease pain perception, fatigue, and anxiety in recurrent ovarian carcinoma patients [34].

Despite these positive outcomes, some studies advise that results must be interpreted with caution due to the limited number of studies and methodological variations across research [31]. Furthermore, barriers to the widespread application of PARO are expense, infection control, and ethical considerations regarding the use of robotic companions in care delivery [32].

However, there's some limitations on PARO Therapeutic Robot is it is not integrated with a camera device, and hence it could not monitor the condition of elderly through real-time video streaming instead only could just monitor by the sensors data. Other than that, PARO Therapeutic Robots are not equipped with any wheels, this will result in the PARO Therapeutic Robot unable to navigate freely in the house for monitoring and companion to elderly in any rooms.

CHAPTER 2

In summary, the literature suggests that PARO is a promising intervention to improve older adults', particularly those with dementia, well-being by acting as a therapeutic robot and providing happiness towards elderly. Further research is needed to address difficulties and ease its integration into care practices.



Figure 2.2.4: Testing on PARO [34]

2.2.3 ElliQ



Figure 2.2.5: ElliQ [35]

Intuition Robotics' ElliQ is a social robot companion that leverages AI in helping elderly people to live independently, enhance wellness, and reduce loneliness. It's hardware comprises a robotic head with a swiveling motion to provide expressive gestures, a touchscreen tablet for visual interaction, in-built microphones and speakers for verbal interaction, and sensors for detecting user presence and interaction. The latest version, ElliQ 3.0, boasts impressive

hardware upgrades, including an octa-core processor and MediaTek dual-core AI chip, making it more powerful and enabling more sophisticated AI features [36].

ElliQ's software leverages state-of-the-art artificial intelligence, such as generative AI and large language models (LLMs) to facilitate natural, context-aware dialogue [37]. This allows the robot to interact with users across a wide range of topics, adapt its interactions based on individual preferences, and provide personalized suggestions for activities and wellness routines. The AI system also includes a Relationship Orchestration Engine, which guides ElliQ's real-time decision-making to facilitate rich and empathetic interactions.

In actual use, ElliQ has been shown to enhance the well-being of older adults. A pilot program by New York State Office for the Aging cited a 95% reduction in loneliness among ElliQ users, with the users experiencing over 30 interactions daily [38]. The interactions range from conversation, reminders for health, cognitive games, and multimedia activities, which translate to improved mental and physical well-being.

One of ElliQ's most important features is its proactive conversation system, encouraging elderly to engage in constant interaction [39]. Through AI-powered speech recognition and response, ElliQ supports the conversation, gives assistance, and provides companionship to eliminate loneliness and isolation. The feature stands out from passive monitoring systems because it makes the interaction more natural and engaging, helping to preserve the mental health of older individuals.

ElliQ also comes with customized reminders and health check-ins, reminding the elderly individuals of their daily routines. It has reminders for medication, hydration, and wellness check-ins to allow the elderly individuals to be healthy and independent. The system can also synchronize with medical information and send out reports to caregivers, giving a more structured and efficient health management system. This option is especially ideal for patients with memory loss or ongoing health conditions who need to be checked constantly.

Another significant feature is entertainment and mental stimulation, as ElliQ provides music, news, trivia, and games that are designed to keep elderly people mentally active. With interactive content that is user-interest-based, it assists in cognitive health and mental stimulation, avoiding diseases like dementia and depression. ElliQ can also suggest activities based on the user's interests, improving the experience and making it more enjoyable.

Elliq is also capable of real-time communication with caregivers and loved ones. Its voice and video calling feature enables users to make connection with loved ones with easy, uncomplex technology. This feature is critical to preventing social connection loss, particularly for the elderly who may find smartphones or computers challenging to use. With an uncomplex communication process, Elliq closes the distance gap between the elderly and loved ones, which typically results in isolation when aging.

Despite its innovative design, Elliq is still a stationary device, i.e., it cannot move or provide full-room monitoring. While it offers substantial social interaction and health monitoring, it does not offer autonomous mobility, real-time monitoring, or fall detection that is critical for emergency support. Compared to our TurtleBot3-based carebot that integrates mobility, AI-powered fall detection, and remote monitoring, Elliq would better be categorized as an engagement-focused assistant rather than an umbrella elderly care solution.

Overall, Elliq is an excellent solution for social interaction, health management, and real-time communication and thus an excellent AI-based assistant for independent older adults. For those requiring more safety and continuous monitoring, a mobile robot system with AI-based monitoring would be a more integrated solution.

2.2.4 Comparison between previous works and proposed works

Table 2.2.1 shows the comparison between the existing work (Follow Me Robot, Paro PARO Therapeutic Robot, Elliq AI Companion) and the proposed solution.

Table 2.2.1: Comparison between previous works and proposed works

Feature	Follow Me Robot	Paro Therapeutic Robot	Elliq AI Companion	Proposed Mobile Companion Robot
Mobility	High	None	None	High
Autonomous Navigation	No (Only basic navigation)	No	No	Yes (Moves to monitor elderly in every room)
Real-Time Monitoring	No	No	No	Yes (Dynamic monitoring with movement)

CHAPTER 2

Fall Detection	No	No	No	Yes (AI-powered camera detection)
Voice Interaction	Yes (Basic commands)	No (Only responds to auditory stimuli)	Yes (Conversational AI)	Yes (AI-driven speech interaction)
Companionship & Engagement	No	Yes (Through Sensors)	Yes (AI-driven social interactions)	Yes (Conversational AI with interactive responses)
Remote Monitoring	No	No	No	Yes (Web-based caregiver interface)
Emergency Alerts	No	No	No	Yes (Real-time caregiver notifications)
Health & Medication Reminders	No	No	Yes	Yes (Customizable alerts and reminders)
User Interface (UI)	No	No	Yes (Touchscreen Tablet)	Yes (Web Interface)
Telegram Bot	No	No	No	Yes

2.3 Chapter Summary

This chapter reviewed the key technologies selected for the proposed mobile companion robot. These include Ubuntu Server and Desktop as the operating systems, Python as the primary programming language, SLAM with the Navigation2 Simple Commander API for mapping and navigation, AI-based fall detection models for health monitoring, Vosk and Whisper for speech recognition, Google TTS for voice synthesis, and the Flask framework for seamless web-based communication. The critical evaluation confirmed that these technologies collectively provide a robust, lightweight, and cost-effective foundation for the system.

In addition, this chapter also critically analyzed three robotic solutions in the elderly care sector including Follow Me Robot, PARO Therapeutic Robot, and ElliQ Robot to evaluate their strengths and limitations. The Follow Me Robot provides basic navigation support but lacks interaction, health monitoring, and emergency alert features. The PARO Therapeutic Robot is effective for therapy and emotional support, yet it does not support voice interaction, health monitoring, or emergency functions. ElliQ offers advanced conversational AI, personalized interaction, and health reminders, but it is stationary and lacks real-time remote monitoring capabilities.

A direct comparison was then drawn to highlight the originality and strengths of the proposed design. Unlike the reviewed systems, the proposed robot integrates mobility, in-home real-time health monitoring, emergency alert capabilities, AI-based voice interaction, and a web-based caregiver interface into a single platform. This comprehensive integration addresses the shortcomings of existing solutions and demonstrates the novelty and value of the proposed system in supporting independent living for the elderly.

Chapter 3

SYSTEM METHODOLOGY/APPROACH (FOR DEVELOPMENT-BASED PROJECT)

3.1 Methodology

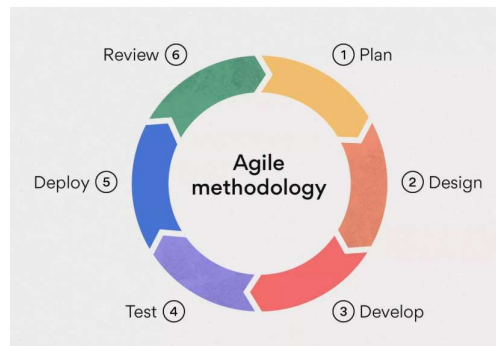


Figure 3.1.1: Agile Methodology [40]

For this project, a modified Agile methodology was adopted, combining the principles of Agile development with formal time management techniques to ensure flexibility, adaptability, and the timely delivery of objectives. The Agile approach is well-suited for projects that require iterative development and continuous feedback, making it an ideal fit for this individual robotics project. Given the nature of working independently, the methodology places a strong emphasis on self-discipline, adaptability, and ongoing collaboration with the project supervisor to ensure progress remains aligned with project goals and deadlines [41]. This also allowed for continuous assessment of the work, helping to identify issues early and adjust the project plan as needed.

The project was divided into four distinct phases: Planning, Development, Testing, and Documentation, each of which aligns with Agile principles of iterative cycles, feedback, and continuous improvement.

In the Planning phase, the project objectives, deliverables, and timelines were established. The planning included consultations with the supervisor to ensure the approach was feasible and aligned with the overall project scope. The Planning phase served as the foundation for the project, allowing for the creation of a systematic roadmap that guided the execution of tasks. This was done by breaking the project into manageable goals and setting deadlines for each key milestone, with flexibility built in for adjustments based on feedback.

During the Development phase, a timeboxing approach was used, which is a core Agile technique. In this phase, specific time slots were allocated for each key task, such as SLAM implementation for navigation, web interface development, and AI model integration. This approach ensured that each task was given sufficient time, while also maintaining focus and productivity throughout development. The Agile principle of sprints was applied, where the focus shifted to completing manageable tasks within specific time periods. Regular self-assessments were conducted at the end of each sprint to evaluate progress, identify difficulties, and make necessary adjustments to the plan. Supervisory feedback played a key role in refining the approach, ensuring the project remained on track.

The Testing phase was integrated throughout the project, in line with Agile's emphasis on continuous testing and validation. Each system module such as SLAM navigation, fall detection, and voice interaction was rigorously tested to ensure performance, reliability, and stability. When errors or issues were discovered, they were documented, addressed, and re-tested to confirm the fixes were effective. This process of iterative testing and refinement ensured that each system component was functioning as expected before moving on to the next phase. Frequent feedback loops allowed for quick identification and resolution of issues.

Finally, the Documentation phase involved keeping detailed records of the development process, including the methods used, issues encountered, and implemented solutions. Agile methodology encourages reflective documentation, so this phase was focused on creating comprehensive notes outlining the challenges faced during development and the decisions made to overcome them. This also ensured that all work was thoroughly documented for future reference and potential replication. The documentation process remained flexible, with updates made regularly to reflect changes in project direction or scope based on feedback received during development and testing.

In short, by incorporating Agile-inspired techniques such as timeboxing, sprints, self-assessments, and continuous testing and feedback, this methodology provided a balanced, flexible framework that facilitated smooth progression throughout the project. The project's development was guided by structured direction while allowing for continuous adjustments and improvements, ultimately leading to the successful and timely delivery of the robot's autonomous capabilities.

3.2 System Design Diagram

The overall system architecture, as shown in Figure 3.2.1, is centered around the integration of a TurtleBot3 Burger mobile robot and a Remote PC, which communicate over a wireless network. This distributed computing model was chosen to facilitate the project's Agile methodology, allowing for iterative development and testing of key modules while offloading computationally intensive tasks, such as AI model processing and data stream management, to the more powerful Remote PC. By splitting tasks between the TurtleBot3 Burger and the Remote PC, the project maximized the use of available resources, with the TurtleBot3 focusing on real-time motor control and sensor data acquisition, while the Remote PC handles AI-driven processing and complex navigation algorithms.

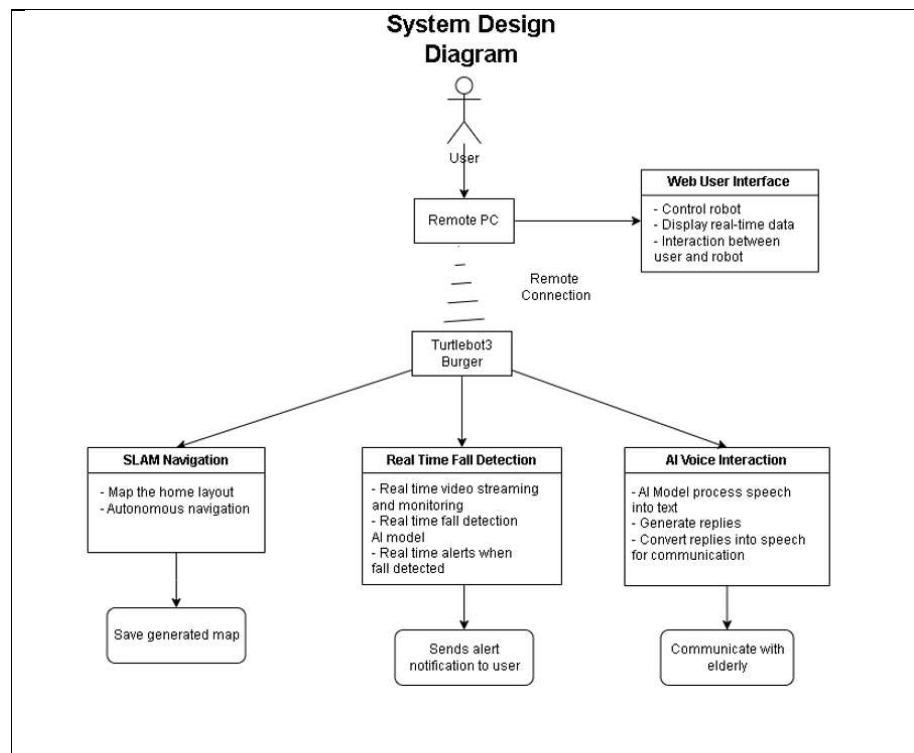


Figure 3.2.1: Overall System Design Diagram

The user interacts with the system primarily through a Web UI hosted on the Remote PC. This interface serves as the command center of the robot, allowing the user to remotely control the robot, view real-time data such as camera feeds, and receive critical notifications, including alerts for fall detection. By leveraging Flask and ROS2 topics, the Web UI provides a seamless bridge between the robot's backend which is running on ROS2 and the user's

interface, ensuring that data flows in real-time and that users can monitor and interact with the robot effectively.

The core functionalities of the robot are divided into three key modules, each of which supports the overall goals of the project and aligns with the methodology for iterative testing and development:

First of all, SLAM Navigation. This module enables the robot to build and navigate through maps of its environment using SLAM (Simultaneous Localization and Mapping). It forms the foundation of the robot's autonomy, ensuring it can map and navigate without human intervention, and will be tested iteratively to ensure accuracy in real-world environments.

Next, this project had implemented Real-Time Fall Detection. The fall detection module uses AI models to analyse data from the robot's camera and sensor suite, detecting any potential falls in real-time. This is a critical component for ensuring user safety, and its performance will be continuously evaluated to optimize detection accuracy.

Besides, it also had the capabilities of AI-Driven Voice Interaction. This module allows the user to interact with the robot through natural language commands, powered by Speech-to-Text (STT) and Text-to-Speech (TTS) models. By providing voice-based control and feedback, this subsystem enhances user experience and accessibility. Like other modules, this system will be improved iteratively, with feedback loops from testing phases incorporated into its development.

Each of these modules is developed and tested incrementally, following an Agile-inspired approach where tasks are broken into smaller components and continuously refined based on feedback and performance. This iterative process ensures that each subsystem is optimized before integration into the full system, aligning with the methodology's emphasis on flexibility, regular evaluation, and adaptability.

3.3 Development Tools And Communication Interfaces

This section will describe the tools and software used throughout the development process, detailing how each tool was employed to support system setup, development, and communication.

1. Oracle VM VirtualBox



Figure 3.3.1: Oracle VM VirtualBox [42]

Oracle VM VirtualBox is a software that enables user to host multiple Virtual Machine on a PC. Oracle VM VirtualBox is installed to create a Virtual Machine (VM) to act as the remote PC to TurtleBot3. The VM is installed with Ubuntu Desktop. The VM created is downloaded with ROS2 and RVIZ to perform SLAM navigation and AI model processing in the remote PC instead of in the Turtlebot3.

2. MobaXterm

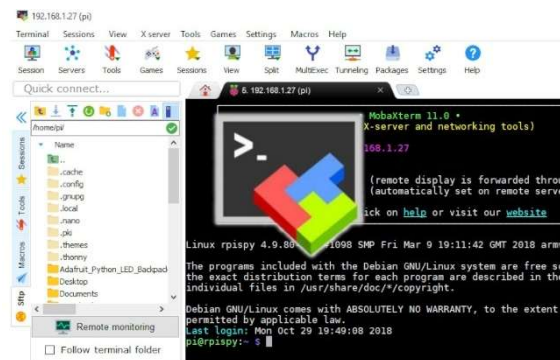


Figure 3.3.2: MobaXterm [43]

MobaXTerm was used for remote access to the Raspberry Pi via SSH, enabling file transfers and terminal commands during development and testing. It allowed easy management of the Raspberry Pi's terminal and remote system, ensuring smooth communication between the local development environment and the robot.

3. Telegram (TelegramBot)

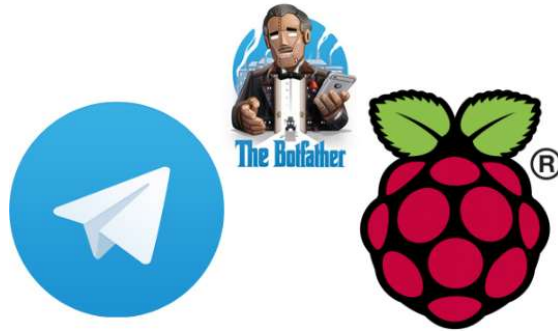


Figure 3.3.3: Telegram Bot [44]

The Telegram Bot enabled communication between the robot and the user via Telegram, sending real-time alerts and updates such as fall detection notifications. The bot was integrated with the robot to provide an easy-to-use notification system, ensuring that the user received critical alerts remotely.

4. Visual Studio Code

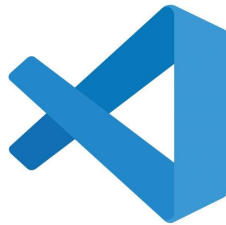


Figure 3.3.4: Visual Studio Code

Visual Studio Code is a lightweight, versatile IDE with excellent support for Python, ROS2 development, and integration with Git for version control. In this project, it is used as the primary code editor for writing Python scripts and developing ROS2 nodes.

5. Simple Commander API



Figure 3.3.5: Simple Commander API

Simple Commander API was used to interface with the robot’s hardware and execute specific commands, such as controlling the robot’s movements or initiating tasks. It simplified the process of commanding the robot remotely, integrating it seamlessly into the system’s control flow.

3.4 Timeline

3.4.1 FYP 1 Timeline

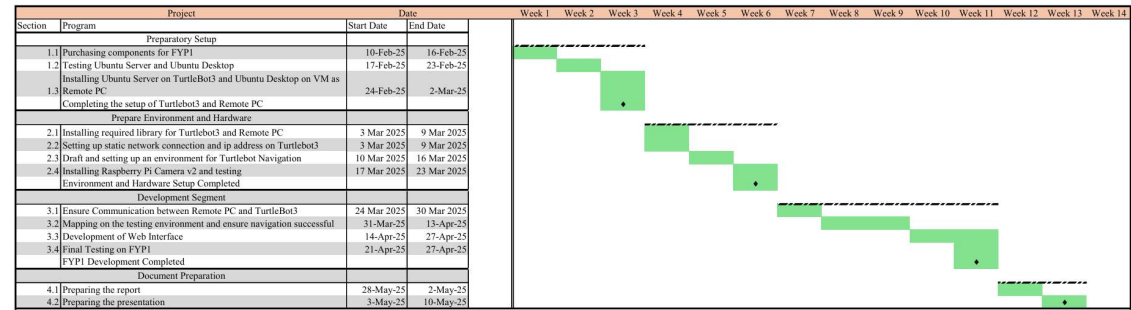


Figure 3.4.1: FYP 1 Timeline and Milestone

This project is divided into several stages including Planning Section, Preparatory Section, Design Section, Development Section, Testing Section and Closure Section. In FYP1, this project is targeted to be completed 30% including the planning section, preparatory section and minor part of the development section. For the planning section, it is completed within the first two weeks of this semester, which includes selecting and purchasing the hardware and software components required in this project. In the following weeks, it comes to the preparatory section which requires the setup of hardware including Turtlebot3 and a remote PC, installation of operating system, integrating camera to the robot and preparing the environment for navigation and mapping simulation. Once the preparatory setup of this project is completed, it could

CHAPTER 3

proceed to development stage to achieve autonomous navigation and mapping including the fourth objectives which is developing WebUI for the robot for presentation display.

3.4.2 FYP 2 Timeline

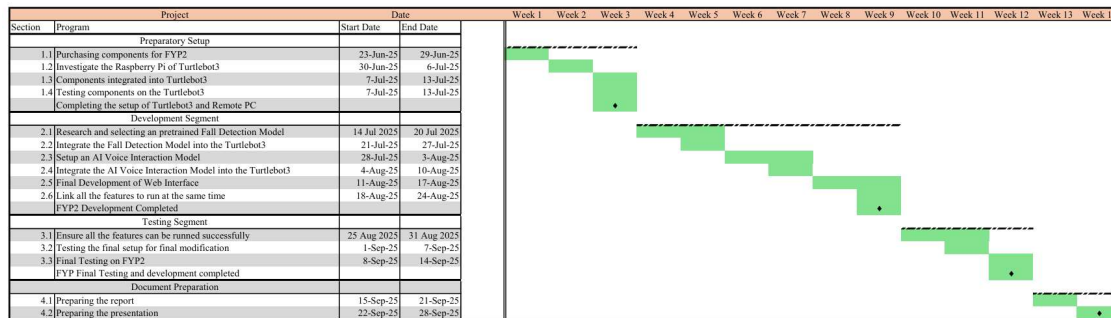


Figure 3.4.2: FYP 2 Timeline and Milestone

Building upon the foundational 30% of the work completed in FYP1, Final Year Project 2 (FYP2) is dedicated to completing the remaining 70% of the project. This semester will focus on integrating advanced AI functionalities into the existing Turtlebot3 platform. The workflow is structured around the core stages of Development, comprehensive Testing, and final project Closure, transforming the initial prototype into a fully featured, intelligent robotic system.

The primary focus of this semester is the advanced Development stage. This begins with researching and integrating a pre-trained Fall Detection Model, a critical safety feature of this project. Following that, an AI Voice Interaction Model will be implemented to allow for intuitive, hands-free control and communication with the robot. The web user interface, initially created in FYP1, will be further developed and refined to support these new features. The culmination of this stage will be the final integration of all systems including autonomous navigation, fall detection, voice control, and the web UI into a single, cohesively functioning application.

Once development is feature-complete, the project will transition into a rigorous Testing phase. The objective here is to ensure the entire system is stable, reliable, and performs accurately under various conditions. This involves end-to-end testing of all integrated functionalities to identify and resolve any bugs or performance issues. The outcome of this stage will be a polished and robust final prototype that is ready for demonstration. The project concludes with the Closure stage, which involves preparing all final deliverables, including the comprehensive final report detailing the entire project lifecycle and the polished presentation for the final review.

3.5 Estimated Cost

Table 3.5.1: Estimated Cost

<i>No</i>	Item	Unit Cost (RM)	Quantity	Total Price (RM)
	Environment Setup			
<i>1</i>	Polystyrene Board	4	4	16
<i>2</i>	Velcro Tape (3M)	11.6	1	11.6
	Total Cost			27.6
	Hardware Setup			
<i>3</i>	Raspberry Pi 8MP Camera Module V2	79	1	79
<i>4</i>	Pan Tilt Servo Kit for Camera	29.9	1	29.9
<i>5</i>	ReSpeaker 2-Microphone Raspberry Pi HAT	69	1	69
<i>6</i>	Raspberry Pi 15-pin Camera FFC Cable - 50cm	6.9	1	6.9
<i>7</i>	Male to Female Jumper Wire	2.5	1	2.5
<i>8</i>	Raspberry Pi Mini USB Microphone	20	1	20
<i>9</i>	MicroSD Card	Sponsored by Lab	1	-
<i>10</i>	Turtlebot3 Burger	Sponsored by Lab	1	-
	Total Cost			207.3
	Overall Cost			234.9

Chapter 4

SYSTEM DESIGN

4.1 System Block Diagram

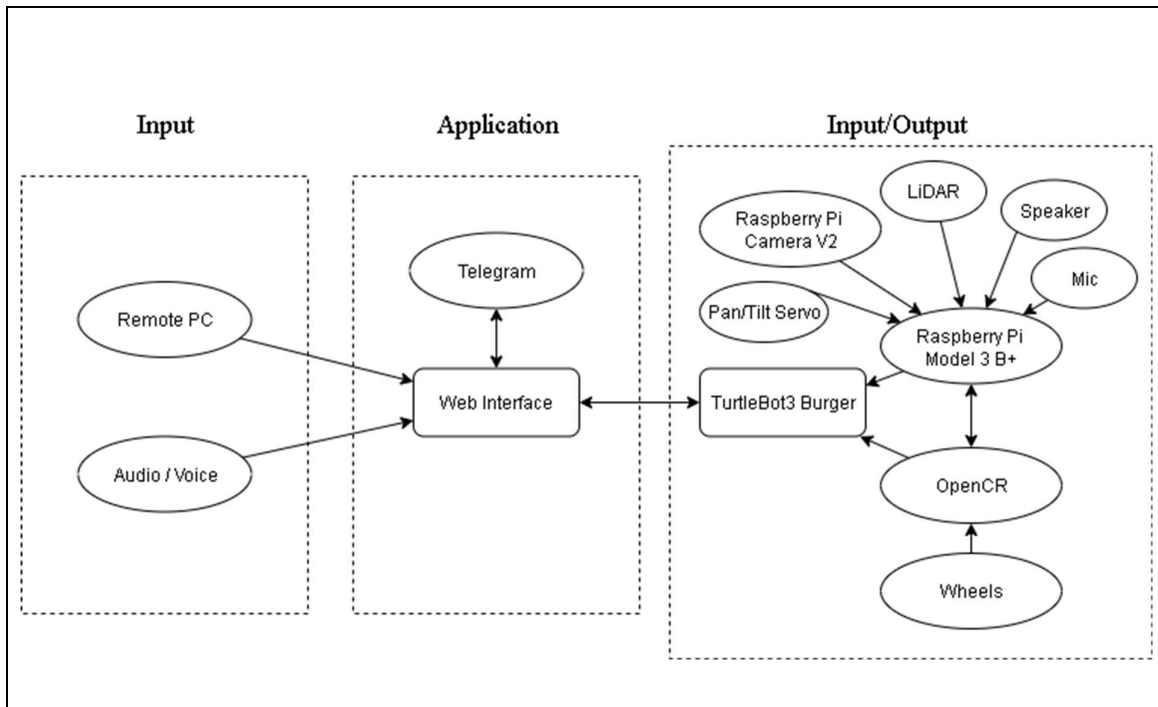


Figure 4.1.1: System Block Diagram

The system's architecture is organized into three distinct functional layers: Input, Application, and Input/Output, as illustrated in the figure below. This design separates user interaction from the core software logic and the robot's physical hardware, creating a clear and modular structure.

1. Input Layer

This layer represents the primary ways a user interacts with the system.

- **Remote PC:** The user's primary machine for accessing the web interface, monitoring the robot, and performing high-level tasks like initiating SLAM or viewing logs.
- **Audio / Voice:** This represents the user's voice commands, which are captured by the robot's microphone as a direct form of input for the AI Voice Interaction module.

2. Application Layer

This layer is the software core of the system, acting as the bridge between the user and the robot.

- **Web Interface:** This is the central hub of the entire system. Developed using Flask, it receives commands from both the Remote PC (e.g., button clicks, navigation goals) and the Audio/Voice input. It processes these commands and relays them to the TurtleBot3. It also displays real-time data from the robot.
- **Telegram:** Integrated with the web interface, Telegram serves as an external communication channel to push critical alerts, such as fall detection notifications, directly to the user's devices.

3. Input/Output Layer

This layer encompasses all the physical hardware of the robot platform, responsible for sensing the environment and performing actions.

Turtlebot3 Burger is the main mobile robot platform that communicates directly with the Web Interface to receive commands and send sensor data. It houses the following key components:

- **Raspberry Pi Model 3 B+:** The onboard main computer or "brain" of the robot. It runs the core ROS2 nodes and manages all connected sensors and actuators.
- **Sensors:** The Raspberry Pi collects data from various sensors, including the LiDAR for mapping and obstacle avoidance, the Raspberry Pi Camera V2 for video streaming and fall detection, the Pan/Tilt Servo for camera positioning, and the Microphone for capturing voice commands.
- **Actuators:** The Pi controls output devices like the Speaker for providing audio feedback and alerts.
- **OpenCR:** This board functions as a dedicated motor driver and sensor interface. It receives high-level commands (like "move forward") from the Raspberry Pi and translates them into precise electrical signals to control the Wheels. It also processes data from the robot's internal IMU and wheel encoders.

4.2 System Components Specifications

4.2.1 Hardware

The development of the companion robot requires a combination of robotic components, sensors, and computing devices to ensure seamless integration of navigation, monitoring, and interaction capabilities. The following hardware components are essential for the project:

1. TurtleBot3 Burger

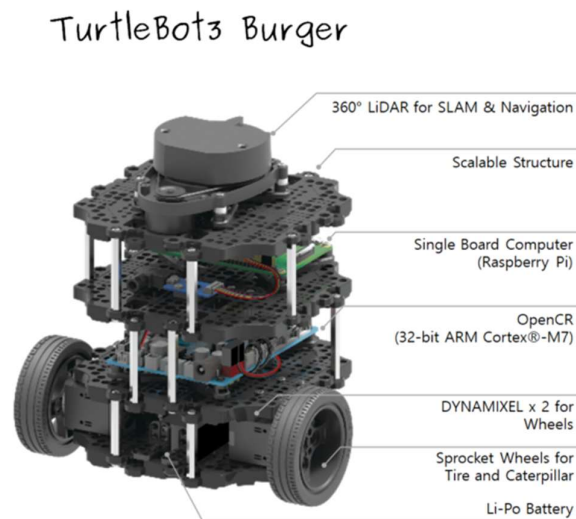


Figure 4.2.1: TurtleBot3 Burger [15]

Table 4.2.1 Specifications of Turtlebot3 [45]

Description	Specification
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Maximum payload	15kg
Size (L x W x H)	138mm x 178mm x 192mm
Weight (+ SBC + Battery + Sensors)	1kg
Climbing Threshold	10 mm or lower
Expected operating time	2h 30m
Expected charging time	2h 30m
SBC (Single Board Computer)	Raspberry Pi 3B+

MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	-
Actuator	XL430-W250
LDS (Laser Distance Sensor)	360 Laser Distance Sensor <u>LDS-02</u>
Camera	-
IMU	Gyroscope 3 AxisAccelerometer 3 Axis
Power connectors	3.3V / 800mA, 5V / 4A, 12V / 1A
Expansion pins	GPIO 18 pins, Arduino 32 pin
Peripheral Connections	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
DYNAMIXEL ports	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences
Programmable LEDs	User LED x 4
Status LEDs	Board status LED x 1, Arduino LED x 1, Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC Connection	USB
Firmware Upgrade	via USB / via JTAG
Power Adapter (SMPS)	Input: 100-240V, AC 50/60Hz, 1.5A@max Output: 12V DC, 5A

2. OpenCR 1.0 (Built in TurtleBot3)

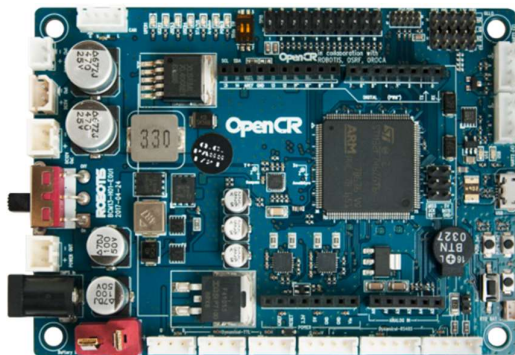


Figure 4.2.2: OpenCR 1.0 [46]

Table 4.2.2 Specifications of OpenCR 1.0 [46]

Description	Specification
-------------	---------------

Microcontroller	STM32F746ZGT6 / 32-bit ARM Cortex®-M7 with FPU (216MHz, 462DMIPS)
Sensors	(New) 3-axis Gyroscope, 3-Axis Accelerometer, A Digital Motion Processor™ (ICM-20648)
Programmer	ARM Cortex 10pin JTAG/SWD connector, USB Device Firmware Upgrade (DFU), Serial
Digital I/O	32 pins (L 14, R 18) *Arduino connectivity, 5Pin OLLO x 4, GPIO x 18 pins, PWM x 6, I2C x 1, SPI x 1
Analog INPUT	ADC Channels (Max 12bit) x 6
Communication Ports	USB x 1 (Micro-B USB connector/USB, 2.0/Host/Peripheral/OTG), TTL x 3 (<u>B3B-EH-A</u> / DYNAMIXEL), RS485 x 3 (<u>B4B-EH-A</u> / DYNAMIXEL), UART x 2 (<u>20010WS-04</u>), CAN x 1 (<u>20010WS-04</u>)
LEDs and buttons	LD2 (red/green): USB communication, User LED x 4: LD3 (red), LD4 (green), LD5 (blue), User button x 2, Power LED: LD1 (red, 3.3 V power on), Reset button x 1 (for power reset of board), Power on/off switch x 1
Input Power Sources	5 V (USB VBUS), 5-24 V (Battery or SMPS), Default battery : LI-PO 11.1V 1,800mAh 19.98Wh, Default SMPS : 12V 4.5A, External battery Port for RTC (Real Time Clock) (<u>Molex 53047-0210</u>)
Input Power Fuse	125V 10A <u>LittleFuse 0453010</u>
Output Power Sources	*12V max 4.5A(<u>SMW250-02</u>) *5V max 4A(<u>5267-02A</u>), 3.3V@800mA(<u>20010WS-02</u>)
Dimensions	105(W) X 75(D) mm
Weight	60g

Table 4.2.3 Role of OpenCR 1.0 in TurtleBot3 [46]

Role	Description
Motor Control	Drives the Dynamixel XL430-W250-T motors (wheels) via PWM signals.
Sensor Integration	Reads: - Wheel encoders (for odometry) - IMU (MPU9250 gyro/accelerometer).
ROS Communication	Translates ROS messages (Example: /cmd_vel and /odom) ↔ motor commands via roserial.

Power Management	Distributes power from the battery (12V LiPo) to motors, sensors, and SBC (Example: Raspberry Pi).
------------------	--

3. Raspberry Pi 3B+ (Built-in TurtleBot3)



Figure 4.2.3: Raspberry Pi 3B+ [47]

Table 4.2.4 Specifications of Raspberry Pi 3B+ [48]

Description	Specification
Processor	Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz
Memory	1GB
Connectivity	<ul style="list-style-type: none">• 2.4 GHz and 5 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 4.2, BLE• Gigabit Ethernet over USB 2.0 (maximum throughput 300Mbps)• 4 × USB 2.0 interface
Video and sound	<ul style="list-style-type: none">• 1 x full size HDMI• MIPI DSI display port• MIPI CSI camera port• 4 pole stereo output and composite video port
Multimedia:	H.264, MPEG-4 decode (1080p30); H.264 encode (1080p30); OpenGL ES 1.1, 2.0 graphics
SD card support:	Micro SD format for loading operating system and data storage
Input Power:	<ul style="list-style-type: none">• 5V/2.5A DC via micro USB connector• 5V DC via GPIO header• Power over Ethernet (PoE)-enabled (requires separate PoE HAT)

Table 4.2.5 Role of Raspberry Pi 3B in TurtleBot3 [48]

Role	Description
ROS Master	Hosts the ROS core (roscore) and manages all ROS communication.
Sensor Processing	Processes data from: - LIDAR (/scan topic) - Camera (if attached) - IMU (via OpenCR).
Navigation/SLAM	Runs algorithms like gmapping (SLAM) or move_base (navigation).
Network Gateway	Enables Wi-Fi/SSH for remote control and monitoring.
Vision Tasks	Executes OpenCV/ML workflows (object detection, etc.).

3. LiDAR Sensor LDS-01 (Built-in TurtleBot3)



Figure 4.2.4: LiDAR Sensor LDS-01 [49]

Table 4.2.6 Role of LiDAR Sensor LDS-01 in TurtleBot3

Role	Description
Environment Mapping	Creates 2D occupancy grids (/scan topic) for SLAM mapping and navigation purpose.
Obstacle Avoidance	Provides real-time distance data (0.12–3.5m range) for navigation stacks (move_base).
Localization	Helps the robot determine its position via AMCL (Adaptive Monte Carlo Localization).

4. Raspberry Pi Camera Module v2

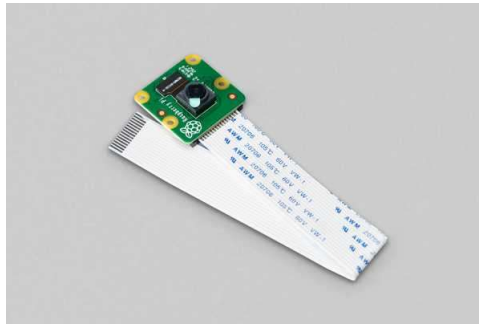


Figure 4.2.5: Raspberry Pi Camera Module v2 [50]

Table 4.2.7 Role of Raspberry Pi Camera Module v2 in TurtleBot3

Role	Description
Vision Processing	Real-Time video streaming and fall detection model.
ROS nodes	Publishes to /camera/image_raw & /camera/image_raw/compressed
Documentation	Captures images/videos for logging or telepresence.

5. USB Microphone For Raspberry Pi



Figure 4.2.6: USB Microphone For Raspberry Pi [51]

Table 4.2.8 Role of USB Microphone For Raspberry Pi

Role	Description
Voice Commands	Captures user's spoken words as an audio signal
Audio Feedback	Streams processed replies or messages to speakers for voice interaction.

6. USB Speaker on Raspberry Pi



Figure 4.2.7: USB Speaker on Raspberry Pi

Table 4.2.9: Role of USB Speaker on Raspberry Pi in TurtleBot3

Role	Description
Voice / Speech Synthesis	Serves as the robot's "voice" by converting digital audio signals.
System Alerts & Notifications	Provides non-verbal auditory cues to the user such as task completion, low battery.
Human Interaction	Verbal responses to voice commands.

7. Pan Tilt Servo Kit for Camera

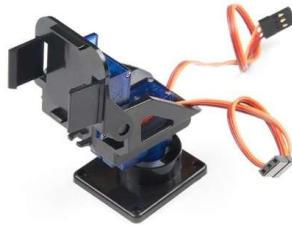


Figure 4.2.8: Pan Tilt Servo Kit for Camera [52]

Table 4.2.10: Pan Tilt Servo Kit for Camera in TurtleBot3

Role	Description
Dynamic Movement for Camera	Ensures the camera can have dynamic movement to visualize in wider or various angle.
Human Interaction	Control the display of the angle of camera, instead of moving the whole robot.

8. Laptop

Table 4.2.11: Specifications of laptop

Description	Specifications
Model	Asus ZenBook UX425EA
Processor	11th Gen Intel(R) Core (TM) i5-1135G7
Operating System	Windows 11 Home
Graphic	Intel® Iris Xe Graphics
Memory	8GB LPDDR4X RAM
Storage	512GB M.2 NVMe™ PCIe® 3.0 SSD

Table 4.2.12: Role of laptop

Role	Description
Communication	Act as remote PC for communication to TurtleBot3 via VirtualBox.
AI Model	Processes real time video with AI fall detection model. Process speech to text and generate output.
Web Interface	Launch Web Interface for TurtleBot3 locally.

4.2.2 Firmware

1. ROS2 Humble



Figure 4.2.9: ROS2 Humble [53]

ROS2 Humble acts as the core robotics framework that enables communication between nodes. The reason for selecting ROS2 Humble in this project is to support the following features such as teleoperation, SLAM, navigation, manipulation and autonomous driving. ROS2 provides a flexible and scalable architecture, which is crucial for complex robotic systems. It supports

Bachelor of Information Technology (Honours) Communications and Networking
Faculty of Information and Communication Technology (Kampar Campus), UTAR

real-time communication and multi-threading for distributed systems, enabling efficient use of resources. Additionally, it is optimized for embedded systems like the Raspberry Pi, which is central to your robot's operation.

2. Ubuntu



Figure 4.2.10: Ubuntu [53]

Ubuntu is the operating system used across the project's components. It plays a crucial role in providing the foundation for running the robot's software stack, supporting communication between nodes, and ensuring system stability. There are two versions of Ubuntu used in the project:

- **Ubuntu Server 22.04 LTS (Raspberry Pi)**

Ubuntu Server 22.04 LTS (Jammy Jellyfish) is installed on the Raspberry Pi to act as the primary operating system for the robot. It provides a lightweight, command-line-based Linux environment optimized for ARM processors. This ensures efficient resource usage while still supporting essential software such as ROS2 Humble and device drivers. The server edition is suitable for headless operation, enabling remote access and integration with robotics frameworks [16].

- **Ubuntu Desktop 22.04 LTS (Remote PC)**

Ubuntu Desktop 22.04 LTS is installed on the development computer. It offers a full graphical user interface (GUI) and a stable Linux environment for coding, simulation, and debugging. It also supports development tools such as ROS2, Python, and web frameworks. Ubuntu Desktop is widely used in robotics research and provides seamless compatibility with ROS2 Humble, making it an ideal choice for this project [17].

4.2.3 Software/Framework

1. RViz2 (SLAM Navigation and Mapping)

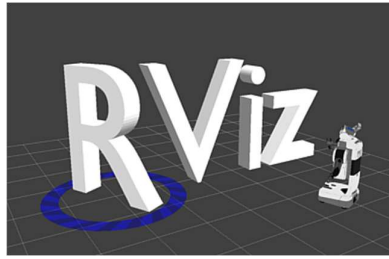


Figure 4.2.11: Rviz

Rviz2 is software that runs on top of ROS or ROS2. It is a visualization tool that is used to visualize sensor data such as camera feeds and LiDAR. Rviz2 is useful in this project as it is used to integrate with ROS and verify LiDAR scans in real-time. It also serves the purpose to monitor robot navigation and ensure the correct environment mapping.

2. Flask



Figure 4.2.12: Flask

Flask is used to develop the front-end interface that communicates with the robot's backend. The web interface provides the user with real-time updates, such as sensor data, battery status, and robot position. It also allows the user to send commands to control the robot, like setting navigation goals or activating specific functions.

3. Vosk



Figure 4.2.13: Vosk

Vosk is used to enable voice command functionality, allowing the robot to receive and interpret user commands spoken in natural language. The speech data captured by the microphone is processed and converted into text by Vosk, which is then used for command parsing and action

Bachelor of Information Technology (Honours) Communications and Networking
Faculty of Information and Communication Technology (Kampar Campus), UTAR

execution. Vosk is lightweight and operates offline, making it suitable for embedded systems like the Raspberry Pi. It's also designed for real-time speech recognition, which is crucial for interactive systems like the robot in this project.

4. Whisper



Figure 4.2.14: Whisper

Whisper is used in conjunction with Vosk to enhance the robot's ability to understand voice commands. While Vosk is used for basic speech recognition, Whisper provides higher accuracy and supports multiple languages, which is useful for processing more complex commands and multilingual support. Whisper provides high-quality transcription with greater accuracy than some other systems, making it ideal for handling natural language commands and improving the overall performance of the voice interface.

5. GTTS



Figure 4.2.15: Google Text-to-Speech

GTTS is used to generate spoken responses from the robot after processing voice commands or when providing feedback to the user. For example, after interpreting a user's command, GTTS generates an audio response, which is then played through the robot's speaker. GTTS is a simple, lightweight tool that can convert text into natural-sounding speech. It integrates well with Python and can be easily adapted to the robot's system for providing voice feedback to the user.

4.3 Circuits and Components Design

This section details the electronic architecture and physical connections of the robot's onboard systems. The design centers on the Raspberry Pi 3B+ as the main processing unit, interfacing with all sensors and actuators. The following diagrams and descriptions provide a comprehensive guide for reproducing the system's hardware configuration.

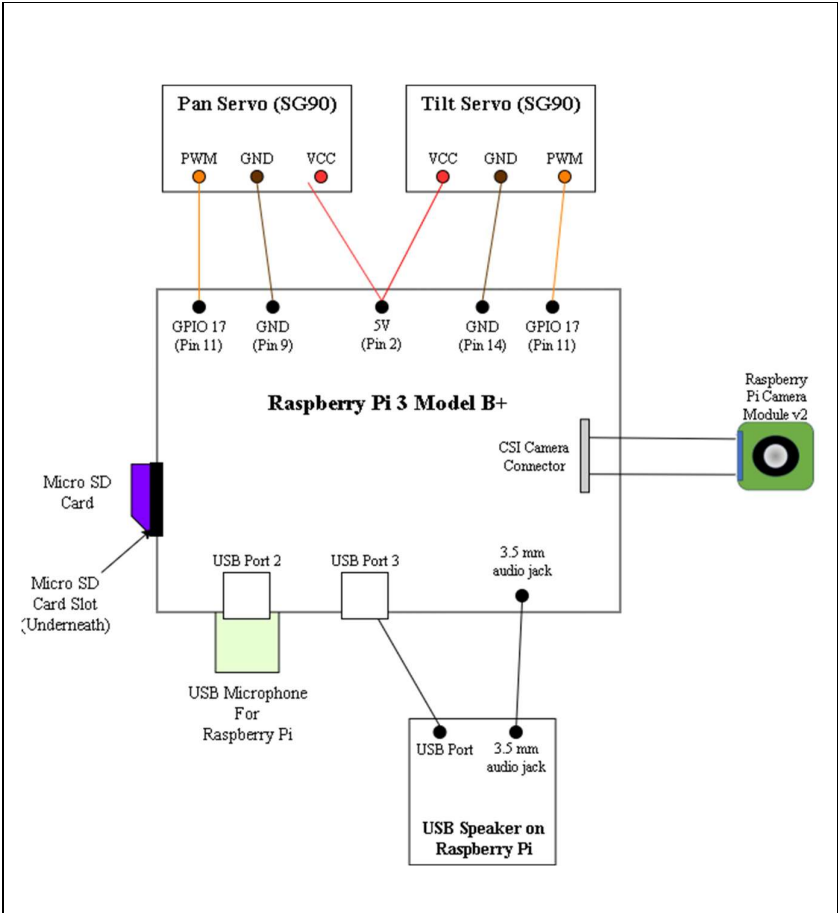


Figure 4.3.1: Raspberry Pi Connections

Table 4.3.1: Components Description and Connection

Component	Connection Type	Connected To	Purpose
Pan Servo (SG90)	Signal	RPi Pin 11 (GPIO17)	PWM Control Signal
	Power (+5V)	RPi Pin 2 (5V)	5V Power Supply
	Ground (GND)	RPi Pin 9 (GND)	Common Ground

Tilt Servo (SG90)	Signal	RPi Pin 12 (GPIO18)	PWM Control Signal
	Power (+5V)	RPi Pin 2 (5V)	5V Power Supply
	Ground (GND)	RPi Pin 14 (GND)	Common Ground
LDS-01 LiDAR	Data	RPi USB Port 1	SLAM & Navigation Data
USB Microphone	Data/Power	RPi USB Port 3	Audio Input
Speaker	Power	RPi USB Port 4	5V Power Supply
	Signal	RPi 3.5mm Audio Jack	Audio Signal Output
Raspberry Pi Camera	Data	RPi CSI Port	Video Stream Input
OpenCR Board (SBC)	Power Out	RPi Micro USB Port	5V Power Supply to Pi

The entire robot is powered by a LI-PO 11.1V 1800mAh battery. This battery connects directly to the OpenCR board (SBC), which serves as the central power distribution hub and contains the necessary voltage regulators.

The OpenCR board steps down the voltage to power the robot's wheel motors directly. It also provides a stable 5V output to the Raspberry Pi via a USB cable connected to one of the Pi's USB ports. The Raspberry Pi, in turn, distributes 5V power from its GPIO header to the pan-tilt servos and through its other USB ports to the LiDAR, microphone, and speaker. This hierarchical power design ensures each component receives its required stable voltage from a single battery source.

4.4 System Components Interaction Operations

4.4.1 Overall System Design

As shown in Figure 4.4.1 below, the system's architecture is centered around a Turtlebot3 Burger mobile robot and a Remote PC, which communicate via a wireless remote connection. This distributed computing model is crucial, allowing the computationally intensive tasks, such as running AI models and processing data streams, to be offloaded to the more powerful Remote PC. This leaves the Turtlebot3's onboard controller to focus on real-time motor control and sensor data acquisition.

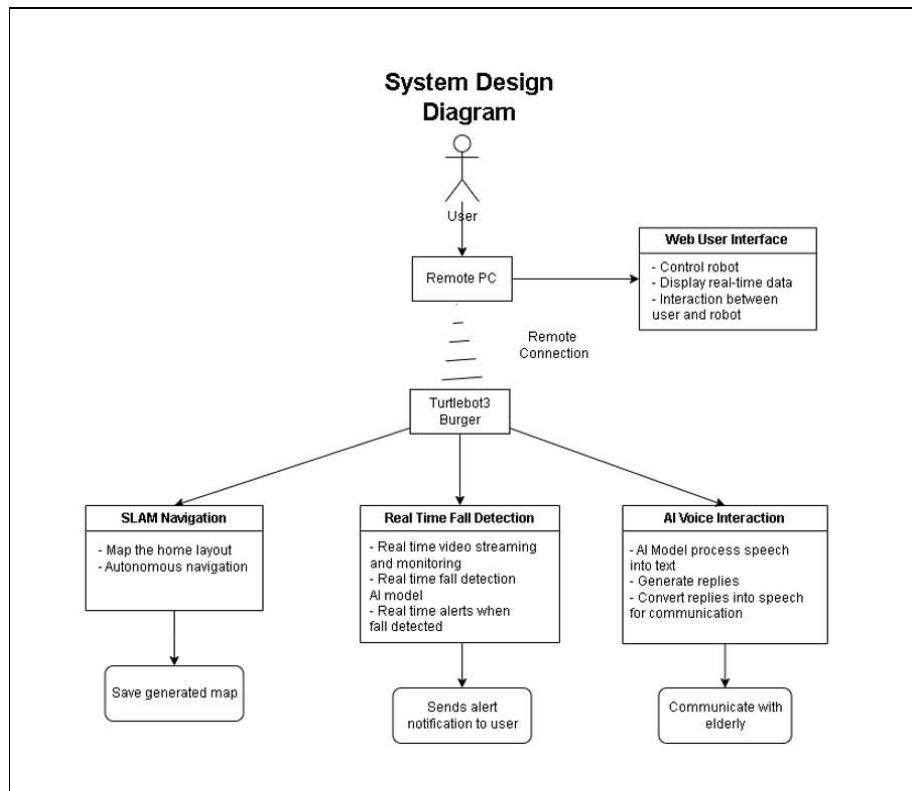


Figure 4.4.1: Overall System Design Diagram

The user interacts with the system primarily through a Web User Interface hosted on the Remote PC. This interface serves as the command center, enabling the user to control the robot, view real-time data (such as camera feeds), and receive critical notifications, like fall detection alerts.

The core functionalities of the robot are modularized into three distinct subsystems: SLAM Navigation, Real-Time Fall Detection, and AI-Driven Voice Interaction. Each of these modules leverages the combined capabilities of the Turtlebot3 and the Remote PC to achieve its specific goals.

4.4.2 Autonomous Navigation

First of all, the foundation of the robot's autonomy is its ability to navigate its environment. This is achieved through Simultaneous Localization and Mapping (SLAM), as detailed in Figure 4.4.2.

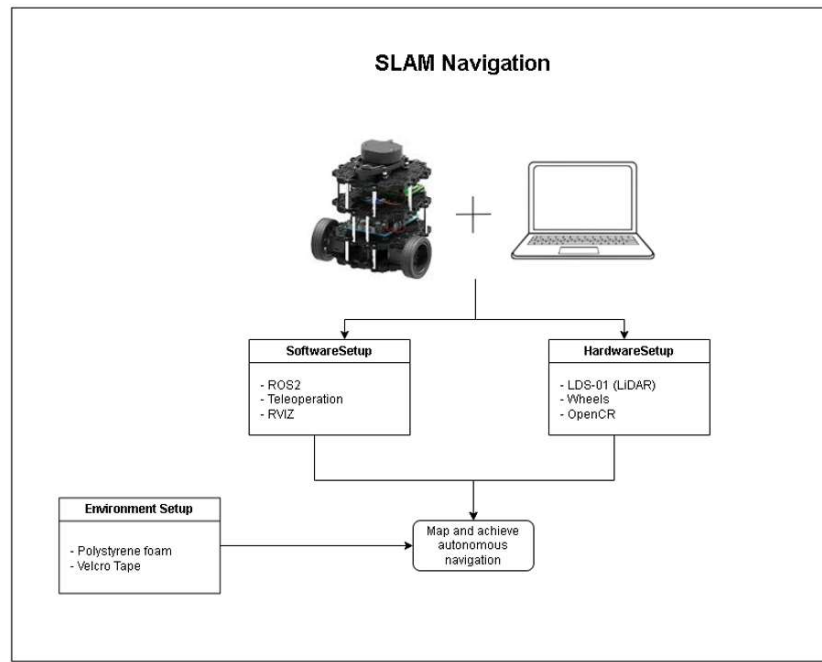


Figure 4.4.2: SLAM Navigation

The main objective is to create a 2D map of the environment (a home layout) and enable the robot to autonomously navigate within that map.

The primary sensor for this module is the LDS-01 (LIDAR), which scans the surroundings to measure distances to obstacles. The robot's movement is managed by its Wheels and the OpenCR main controller board.

The entire process is managed within the Robot Operating System 2 (ROS2) framework. Initially, the robot is manually controlled using Teleoperation to explore the area. The LIDAR data is published to a ROS2 topic, and the SLAM algorithm on the Remote PC processes this data to incrementally build a map. This map, along with the robot's real-time position, is visualized using RVIZ. Once the mapping is complete, the generated map is saved and can be used by the navigation stack for autonomous path planning and movement. The Environment Setup, using materials like polystyrene foam, is employed during the development phase to create a controlled and simplified area for testing and calibrating the SLAM algorithm.

4.4.3 Real Time Fall Detection

Next, to serve as a monitoring and safety assistant, the robot is equipped with a real-time fall detection system, outlined in Figure 4.4.3 below.

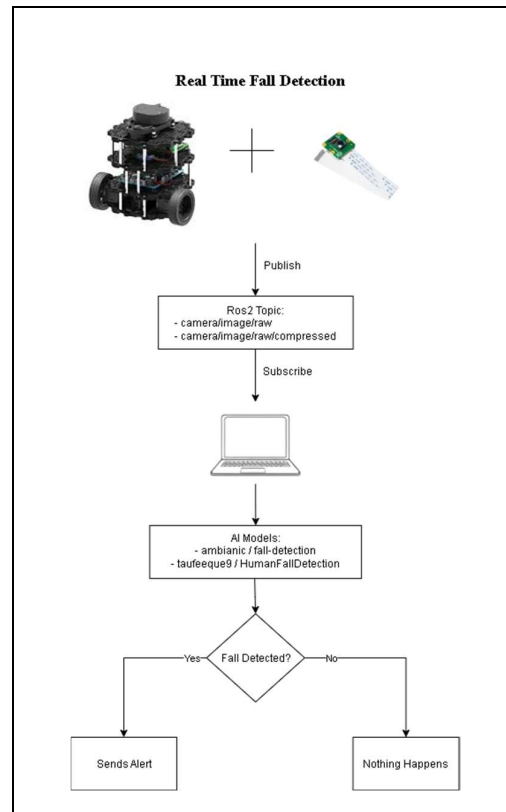


Figure 4.4.3: Real Time Fall Detection

To achieve the main objectives which is continuously monitor the user via a video stream and automatically send an alert if a fall is detected, a camera is mounted on the Turtlebot3 captures the video feed.

The camera node on the robot publishes the video stream to a ROS2 topic (e.g., /camera/image/raw). The Remote PC subscribes to this topic, receiving the video feed in real-time. This feed is then processed by pre-trained AI models (such as ambianic/fall-detection or taufeeqe9/HumanFallDetection) that are specialized in human pose estimation and activity recognition. A decision-making logic analyzes the output of the AI model. If the model's confidence in detecting a fall exceeds a predefined threshold, the system triggers and sends an alert to the user via the web interface. If no fall is detected, the system continues monitoring without interruption.

4.4.4 AI Driven Voice Interaction

Besides, the system also facilitates natural and intuitive communication with the user through an AI-powered voice interaction module, as shown in Figure 4.4.4 below. This module creates a complete loop for capturing, understanding, and responding to human speech.

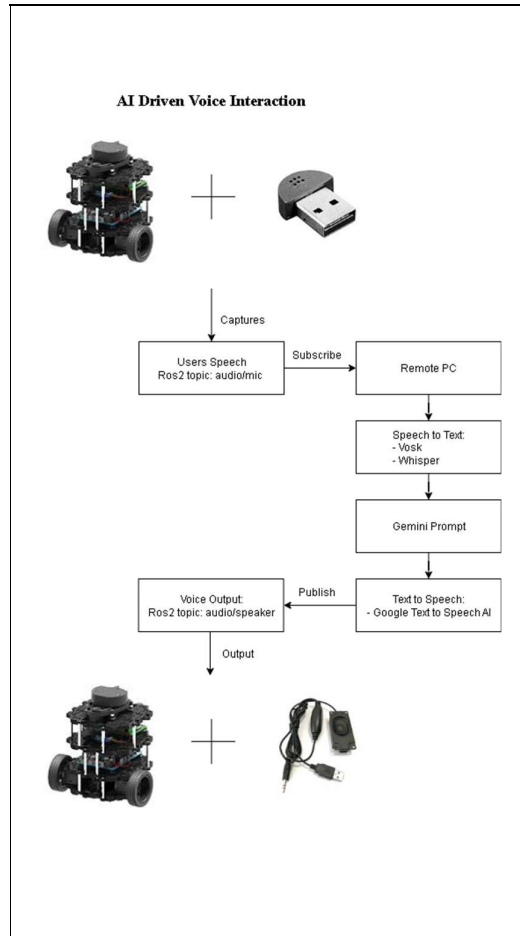


Figure 4.4.4: AI Driven Voice Interaction

The objective of this feature is to allow the user to communicate with the robot using natural language voice commands and receive intelligent, spoken replies while a USB microphone is used to capture the user's speech, and a USB speaker is used to output the robot's synthesized voice.

For the Speech Input Process, the microphone captures the user's voice and publishes the audio data to the /audio/mic ROS2 topic. Then, the Remote PC subscribes to this topic to receive the audio stream and a Speech-to-Text (STT) model, Whisper AI, transcribes the

spoken words into a text string. Lastly, this text is then passed as a prompt to a large language model, Gemini, for natural language understanding and response generation.

For the Voice Output Process, this project utilizes Gemini API to process the prompt and generates a relevant and coherent text-based reply. Then This text reply is then fed into a Text-to-Speech (TTS) engine, Google Text to Speech AI, which converts it into audible speech and the resulting audio data is published to the /audio/speaker ROS2 topic. Lastly, the robot's speaker, subscribed to this topic, receives and plays the audio, completing the communication cycle with the user.

4.5 System Architecture Diagram

In this project, it involves the integration of Turtlebot3 and Remote PC, hence it is important that their connection is established successfully. This section will present the network architecture diagram of this project.

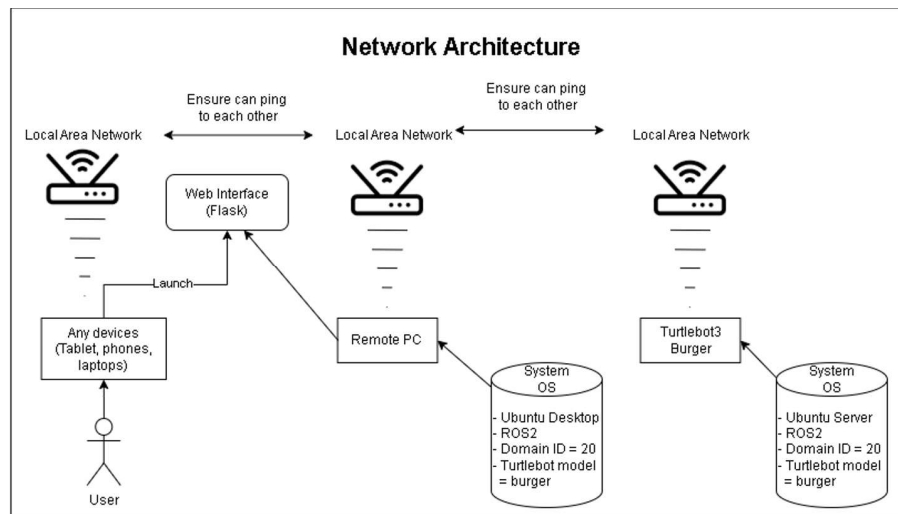


Figure 4.5.1: Network Architecture Diagram

To ensure that the TurtleBot3 and the Remote PC can establish a seamless connection, they are required to first installed the correct System Operating System which is Ubuntu Server and Ubuntu Desktop. Then, the framework ROS2 Humble must be consistent with a synchronized domain identifier and TurtleBot model. It is not compulsory to place them in the same network, if their ping test is successful, the connection could be established.

4.6 System Workflow

4.6.1 Overall System Workflow

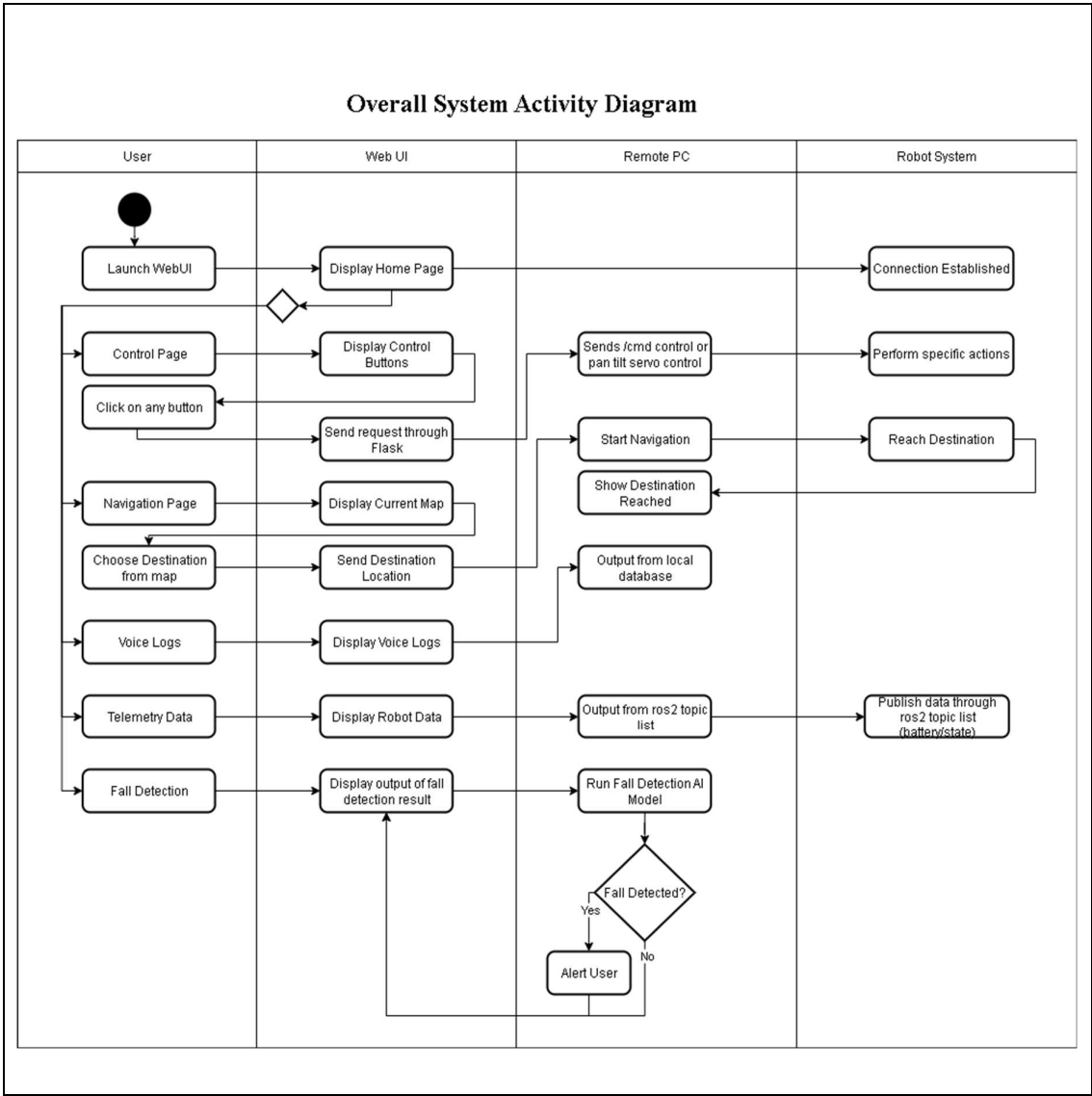


Figure 4.6.1: Overall System Activity Diagram

This system provides a complete framework for a User to remotely control and monitor a Robot System. The interaction is managed through a Web UI, which communicates with a central Remote PC that serves as the brain of the operation. This setup allows the user not only to give direct commands but also to receive critical feedback and automated alerts, ensuring both effective control and safety.

The system offers two primary methods for directing the robot: manual control and autonomous navigation. For direct manual control, the User can access a control page on the Web UI with buttons for real-time actions like movement or camera adjustments. When a button is pressed, a request is sent to the Remote PC, which translates it into a specific command for the Robot to execute instantly. For longer-range movement, the user can select a destination on a map within the UI. This triggers the autonomous navigation function, where the Remote PC commands the robot to travel to the specified coordinates on its own and reports back through the UI upon arrival.

A crucial aspect of the system is its ability to provide the user with real-time data and historical logs from the robot. The Robot System continuously publishes its operational data, such as battery status, using ROS2 topics, which the Remote PC actively monitors and collects. This telemetry data, along with other information like stored voice logs from a database, can be requested by the User at any time through the Web UI. The Remote PC retrieves the requested information and presents it to the user, creating a vital feedback loop that keeps the user informed of the robot's status.

Beyond user-initiated actions, the system includes a proactive safety feature powered by artificial intelligence. The Remote PC continuously runs a fall detection AI model, which analyses sensor or camera data from the robot to check for potential incidents. This process runs in the background without requiring user input. If the model detects a fall, it automatically triggers an alert that is immediately sent to the Telegram to notify the User. This automated monitoring ensures that critical safety events are identified and reported promptly, adding a layer of intelligent oversight to the robot's operation.

4.6.2 SLAM Navigation Workflow

The SLAM navigation process begins when the user activates the mapping mode. Once initiated, the user manually drives the robot to explore the environment. As the robot moves, its LIDAR sensor scans the surroundings and collects real-time sensor data. This data is then processed by the system to create and continuously update a map of the environment. The mapping process continues until the user is satisfied with the coverage, at which point the user halts the process. The system then saves the generated map. Later, the user can set a navigation goal on the saved map. The robot uses the map to plan an optimal path and autonomously navigates to the specified destination, ensuring accurate and efficient movement within the mapped environment.

4.6.3 Real Time Fall Detection Workflow

The real-time fall detection workflow operates in a continuous loop, starting with the robot capturing a video stream via its camera. This video feed is transmitted to the Remote PC, where an AI model processes the data. The system then checks for signs of a fall using a decision node. If no fall is detected, the loop repeats, and the system continues capturing and analyzing the video feed. However, if a fall is detected, the system immediately triggers an alert, notifying the user through the interface. Afterward, the monitoring loop resumes, ensuring ongoing surveillance for potential falls. This process enables continuous safety monitoring for the user, with real-time detection and alerts.

4.6.4 AI-Driven Voice Interaction Workflow

The AI-driven voice interaction workflow begins when the user speaks a command or query to the robot. The robot's microphone captures the spoken audio and streams it to the Remote PC. The audio is then processed by a Speech-to-Text (STT) model, which converts it into text. The text is forwarded to the Gemini AI, which generates an appropriate response. This text-based response is then passed to a Text-to-Speech (TTS) engine, which converts the text into synthesized speech. The generated audio is sent back to the robot, where its speaker plays the response for the user to hear, completing the voice interaction cycle. This workflow enables natural communication between the user and the robot, allowing for hands-free control and feedback.

4.7 Chapter Summary

This chapter provides a comprehensive technical blueprint of the mobile companion robot's system design, covering both hardware implementation and operational logic. It begins with the System Block Diagram, which illustrates the overall architecture of the robot, highlighting the separation between the onboard systems and the remote control station. The System Components Specifications section follows, detailing the individual hardware components, their roles, and technical specifications. Next, the Circuits and Components Design is presented, including wiring schematics, component pinout tables, and descriptions of the power distribution system. The chapter continues with the System Components Interaction Operations, which outlines the step-by-step data flow and interplay between hardware and software for the robot's key functionalities, such as autonomous navigation, voice command processing, and real-time fall detection. The System Architecture Diagram provides a high-level overview of the distributed computing model, where key tasks are offloaded to the remote PC while the robot focuses on real-time control. Finally, the System Workflow section describes the operational processes in detail, explaining how the robot navigates, detects falls, and interacts with the user via voice commands. Collectively, this chapter serves as a complete technical guide, offering all the necessary details to reproduce the robot's hardware setup and understand its operational logic.

Chapter 5

SYSTEM IMPLEMENTATION

5.1 Hardware Setup

5.1.1 Robot Platform Setup

The core of the mobile robot platform is the TurtleBot3 Burger. This model was chosen for its compact size, comprehensive open-source support within the Robot Operating System 2 (ROS2) framework, and modular design. The standard Burger configuration includes a Raspberry Pi 3B+ as the main single-board computer, an OpenCR 2.0 board for motor control and sensor integration, and an LDS-01 360° LiDAR sensor for mapping and navigation.

To expand its functionality for interactive tasks in this project, this base was enhanced with an integrated vision and audio system.

A Raspberry Pi Camera V2 was mounted on a Pan-Tilt Servo Kit to create an dynamic vision system. The two servos of this kit were connected directly to the Raspberry Pi's GPIO header for both power and control signals. The connections use the physical (BOARD) numbering scheme, which corresponds to the BCM pin numbers used in the “pan_tilt_ros_onefile.py” control script.

The specific connections are detailed in the table below:

Component	Connection Type	Physical (BOARD) Pin	BCM Equivalent (for Code)
Pan Servo	Signal	Pin 11	GPIO 17
	Power (+VCC)	Pin 2 (5V)	N/A
	Ground (GND)	Pin 9	N/A
Tilt Servo	Signal	Pin 12	GPIO 18
	Power (+VCC)	Pin 2 (5V)	N/A
	Ground (GND)	Pin 14	N/A

Table 5.1.1: Pan-Tilt Servo Kit Connection

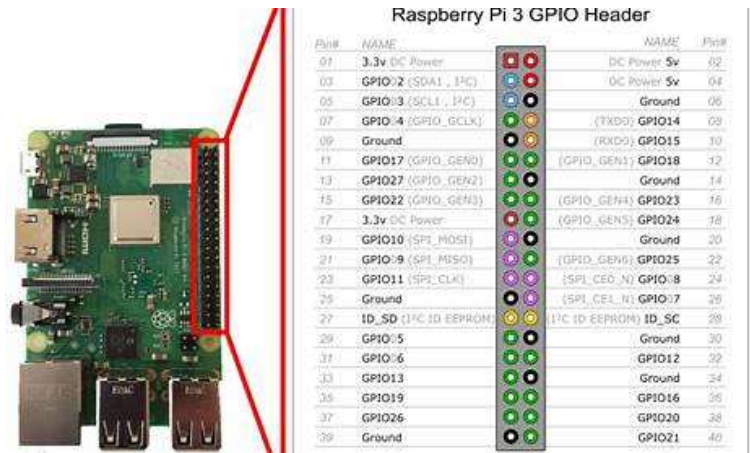


Figure 5.1.1: Raspberry Pi 3 GPIO Header [54]

For audio capabilities, a USB Microphone was connected to a USB port for input, and a Raspberry Pi Speaker was connected via USB for power and the 3.5mm jack for audio output. These modifications transform the TurtleBot3 into an interactive mobile robot capable of seeing, hearing, and responding to its environment.

These modifications transform the TurtleBot3 from a purely navigational platform into a more sophisticated interactive agent. A summary of the key hardware components is provided in the table below.

Component	Function
TurtleBot3 Burger	Base mobile platform, motors, and chassis
Raspberry Pi 3B+	Main onboard computer (ROS Master)
LDS-01 LiDAR	360° distance scanning for SLAM & navigation
Raspberry Pi Camera V2	Visual data acquisition (object detection, etc.)
Pan-Tilt Servo Kit	Controls the camera's orientation (pitch/yaw)
USB Microphone	Captures audio input (voice commands)
Raspberry Pi Speaker	Provides audio output (speech, alerts)

Table 5.1.2: Summary of Hardware Components

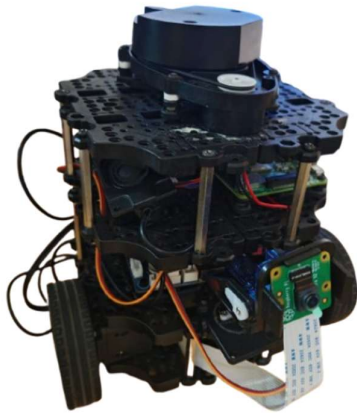


Figure 5.1.2: Ellie Front View

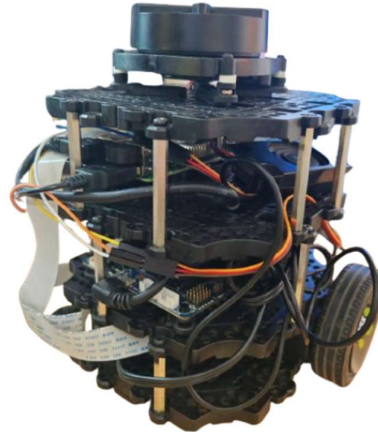


Figure 5.1.3: Ellie Back View

5.1.2 Network Infrastructure

A dedicated local wireless network was established to ensure stable and consistent communication between all project components. A TP-Link 4G LTE router was used to create a Wi-Fi network with the SSID “FYP”.

To guarantee that each device could be reliably addressed, DHCP Address Reservation was configured on the router. This approach assigns a permanent, static IP address to each device based on its unique MAC address, while still allowing the devices themselves to be configured to obtain an IP address automatically. This method prevents IP conflicts and simplifies network discovery, which is crucial for the ROS2 (Robot Operating System) communication protocols used in this project.

The following IP addresses were reserved for the key devices:

- Turtlebot3 (ubuntu): 192.168.1.100
- Host Laptop (LAPTOP-TVNB0IPV): 192.168.1.101
- Remote PC Virtual Machine (FYP): 192.168.1.102

The configuration on the router is shown in the screenshot below, linking each device's MAC address to its reserved IP.

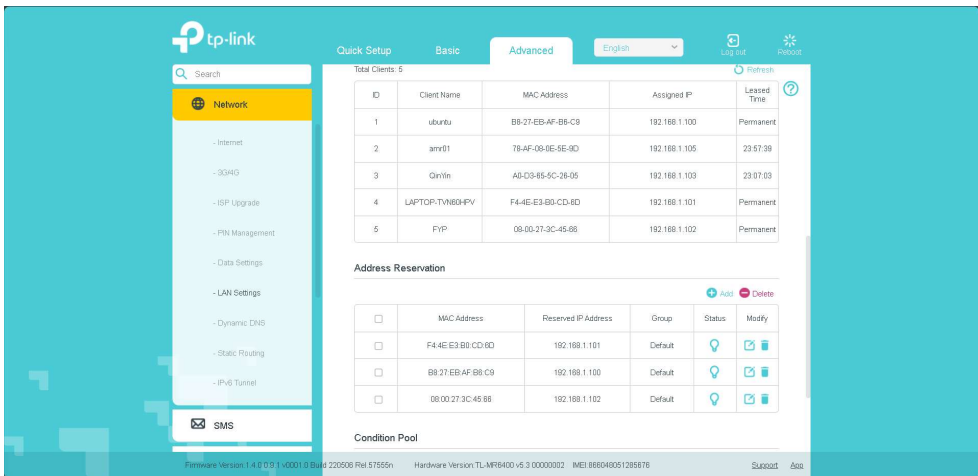


Figure 5.1.4: Network configuration setup

On the client devices running Ubuntu (both the Turtlebot3 and the Virtual Machine), the network was configured using Netplan. The configuration file located at `/etc/netplan/50-cloud-init.yaml` was modified to automatically connect to the “FYP” Wi-Fi network, as shown in the figure below.

```
# This file is generated from information provided by the datasource. Changes
# to it will not persist across an instance reboot. To disable cloud-init's
# network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
# /etc/netplan/01-netcfg.yaml
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: true
      optional: true
  wifis:
    wlan0:
      dhcp4: true
      optional: true
      access-points:
        "FYP":
          password:
```

Figure 5.1.5: 50-cloud-init.yaml file configuration

After saving the changes to the file, the new network configuration was applied by running the following command in the terminal: “`sudo netplan apply`”.

CHAPTER 5

5.2 Software Setup

5.2.1 PC Setup

There is some essential software that are required to be installed on the PC including Oracle VM VirtualBox, Visual Studio Code and MobaXterm.

Ubuntu 22.04 LTS Desktop (64-bit) is the Operating System used for the remote PC. Download the image file from [Download Ubuntu Desktop | Ubuntu](#), then launch a new Virtual Machine using Oracle VM VirtualBox with the image .iso file. It is used to act as the remote PC for TurtleBot3.

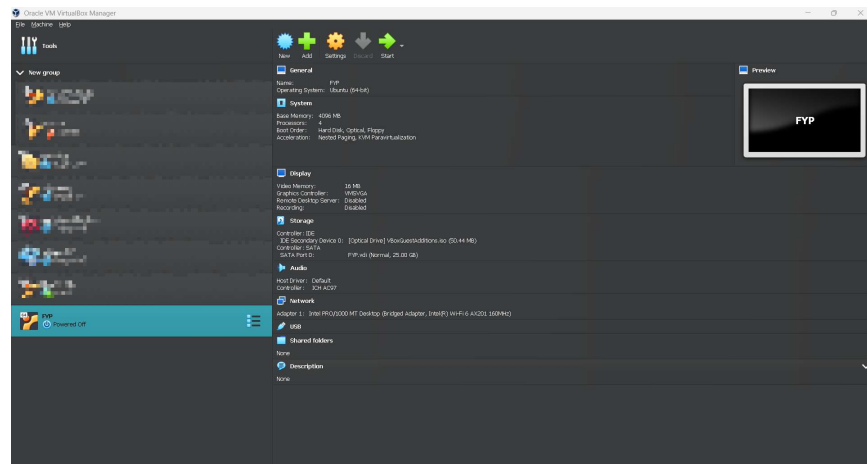


Figure 5.2.1: Setting Up of Virtual Machine

In the Virtual Machine, ROS2 packages and TurtleBot3 packages are required to be installed. ROS 2 Humble is selected over Jazzy and Noetic in this project. The installation of the ROS2 Humble, ROS2 packages and TurtleBot3 packages could be referred to below:

```
$ locale # check for UTF-8

$ sudo apt update && sudo apt install locales
$ sudo locale-gen en_US en_US.UTF-8
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
$ export LANG=en_US.UTF-8

$ locale # verify settings
```

Figure 5.2.2: Ensure locale supporting UTF-8 exists

```
$ sudo apt install software-properties-common
$ sudo add-apt-repository universe
```

Figure 5.2.3: Ensure Ubuntu Universe Repository enabled

```
$ sudo apt update && sudo apt install curl -y
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
```

Figure 5.2.4: Add ROS 2 GPG key with apt


```
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

Figure 5.2.5: Add repository to sources list

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install ros-humble-desktop
```

Figure 5.2.6: Install ROS 2 packages

```
$ sudo apt install ros-humble-gazebo-*
```

Figure 5.2.7: Install Gazebo

```
$ sudo apt install ros-humble-cartographer
$ sudo apt install ros-humble-cartographer-ros
```

Figure 5.2.8: Install Cartographer

```
$ sudo apt install ros-humble-navigation2
$ sudo apt install ros-humble-nav2-bringup
```

Figure 5.2.9: Install Navigation2

```
$ source /opt/ros/humble/setup.bash
$ mkdir -p ~/turtlebot3_ws/src
$ cd ~/turtlebot3_ws/src/
$ git clone -b humble https://github.com/ROBOTIS-GIT/DynamixelSDK.git
$ git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3.git
$ sudo apt install python3-colcon-common-extensions
$ cd ~/turtlebot3_ws
$ colcon build --symlink-install
$ echo 'source ~/turtlebot3_ws/install/setup.bash' >> ~/.bashrc
$ source ~/.bashrc
```

Figure 5.2.10: Install TurtleBot3 Packages

```
$ echo 'export ROS_DOMAIN_ID=20 #TURTLEBOT3' >> ~/.bashrc
$ echo 'source /usr/share/gazebo/setup.sh' >> ~/.bashrc
$ echo 'source /opt/ros/humble/setup.bash' >> ~/.bashrc
$ source ~/.bashrc
```

Figure 5.2.11: Setup ROS environment

5.2.2 SBC Setup

Ubuntu Server 22.04.5 LTS (64-bit) is the Operating System used for TurtleBot3. To install the Ubuntu Server, it is required to plug an SD card to the computer, and launch the software Raspberry Pi Imager, then select the Raspberry Pi Device version, Raspberry Pi 3B is used in this project, Operating System and the storage. Once it is done, plug the SD card to the SBC connected on Turtlebot3.

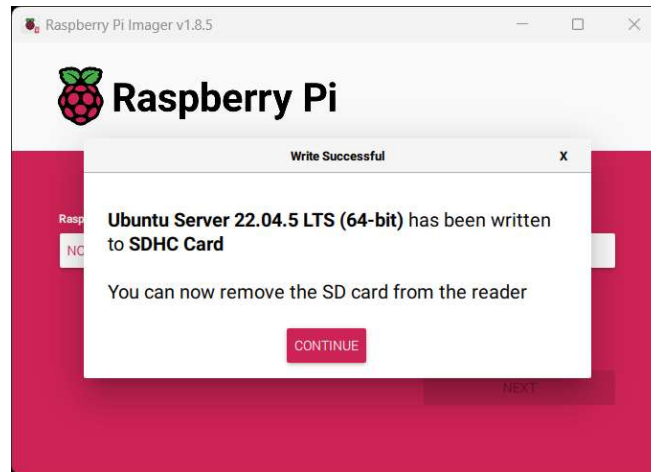


Figure 5.2.12: Installing Ubuntu Server

Next, power up the raspberry pi and configure the network configuration to allow internet access. The steps are listed below:

```
ubuntu@ubuntu:~$ sudo vim /etc/netplan/50-cloud-init.yaml
```

Figure 5.2.13: Open network configuration file

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: true
      optional: true

  wifis:
    wlan0:
      dhcp4: true
      optional: true
      access-points:
        "FYP":
          password: "abc12345"
```

Figure 5.2.14: Example of network configuration file settings

Meanwhile, install the required packages on Raspberry pi, including ROS2 Humble, LDS Configuration.

```
$ sudo apt install python3-argcomplete python3-colcon-common-extensions libboost-system-dev build-essential
$ sudo apt install ros-humble-hls-lfcd-lds-driver
$ sudo apt install ros-humble-turtlebot3-msgs
$ sudo apt install ros-humble-dynamixel-sdk
$ sudo apt install libudev-dev
$ mkdir -p ~/turtlebot3_ws/src && cd ~/turtlebot3_ws/src
$ git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone -b humble https://github.com/ROBOTIS-GIT/ld08_driver.git
$ cd ~/turtlebot3_ws/src/turtlebot3
$ rm -r turtlebot3_cartographer turtlebot3_navigation2
$ cd ~/turtlebot3_ws/
$ echo 'source /opt/ros/humble/setup.bash' >> ~/.bashrc
$ source ~/.bashrc
$ colcon build --symlink-install --parallel-workers 1
$ echo 'source ~/turtlebot3_ws/install/setup.bash' >> ~/.bashrc
$ source ~/.bashrc
```

Figure 5.2.15: Install and Build ROS packages

```
$ sudo cp 'ros2 pkg prefix turtlebot3_bringup /share/turtlebot3_bringup/script/99-turtlebot3-cdc.rules' /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

Figure 5.2.16: USB Port Settings for OpenCR

```
$ echo 'export ROS_DOMAIN_ID=20 #TURTLEBOT3*' >> ~/.bashrc
$ source ~/.bashrc
```

Figure 5.2.17: Setting ROS Domain ID

```
$ echo 'export LDS_MODEL=LDS-01' >> ~/.bashrc
$ source ~/.bashrc
```

Figure 5.2.18: Setting LDS Model

5.2.3 OpenCR Setup

OpenCR is connected to Raspberry Pi using micro-USB cable, while it is already connected on the Turtlebot3 Burger. In this project, relevant packages are required to be installed in order to upload the OpenCR firmware. The following command are required to configured in Raspberry Pi.

```
$ sudo dpkg --add-architecture armhf
$ sudo apt-get update
$ sudo apt-get install libc6:armhf
```

Figure 5.2.19: Install packages for OpenCR

```
$ export OPENCN_PORT=/dev/ttyACM0
$ export OPENCN_MODEL=burger
$ rm -rf ./opencn_update.tar.bz2
```

Figure 5.2.20: Specifying the model for OPENCN

```
$ wget https://github.com/ROBOTIS-GIT/OpenCR-Binaries/raw/master/turtlebot3/ROS2/latest/opencn_update.tar.bz2
$ tar -xvf opencn_update.tar.bz2
```

Figure 5.2.21: Download firmware and required loader for OpenCR

```
$ cd ./opencn_update
$ ./update.sh $OPENCN_PORT $OPENCN_MODEL.opencn
```

Figure 5.2.22: Upload firmware to OpenCR

```
turtlebot@turtlebot:~$ export OPENCNCR_PORT=/dev/ttyACM0
turtlebot@turtlebot:~$ export OPENCNCR_MODEL=burger
turtlebot@turtlebot:~$ rm -rf ./opencnrcr_update.tar.bz2
turtlebot@turtlebot:~$ wget https://github.com/ROBOTIS-GIT/OpenCR/raw/develop/arduino/opencnrcr_release/shell_update/opencnrcr_update.tar.bz2 && tar -xvf opencnrcr_update.tar.bz2 && cd ./opencnrcr_update && ./update.sh 50PENCNCR_PORT 50PENCNCR_MODEL opencnrcr && cd ..
--2018-03-05 11:42:25-- https://github.com/ROBOTIS-GIT/OpenCR/raw/develop/arduino/opencnrcr_release/shell_update/opencnrcr_update.tar.bz2
Resolving github.com (github.com)... 192.30.255.112, 192.30.255.113
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCR/develop/arduino/opencnrcr_release/shell_update/opencnrcr_update.tar.bz2 [following]
--2018-03-05 11:42:25-- https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCR/develop/arduino/opencnrcr_release/shell_update/opencnrcr_update.tar.bz2
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.228.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.228.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 293362 (286K) [application/octet-stream]
Saving to: 'opencnrcr_update.tar.bz2'

opencnrcr_update.tar.bz2  100%[=====] 286.49K  468KB/s  in 0.6s

2018-03-05 11:42:27 (468 KB/s) - 'opencnrcr_update.tar.bz2' saved [293362/293362]

opencnrcr_update/burger turtlebot3_core.ino.bin
opencnrcr_update/waffle turtlebot3_core.ino.bin
opencnrcr_update/opencnrcr_ld_shell_arm
opencnrcr_update/update.sh
opencnrcr_update/
opencnrcr_update/opencnrcr_ld_shell_x86
opencnrcr_update/waffle.opencnrcr
opencnrcr_update/burger.opencnrcr
opencnrcr_update/released_1.0.17.txt
arm7L
arm
OpenCR Update Start..
opencnrcr_ld_shell ver 1.0.0
opencnrcr_ld_main
[ ] file name      : burger.opencnrcr
[ ] file size      : 172 KB
[ ] fw name        : burger
[ ] fw_ver         : 1.0.17
[OK] Open port    : /dev/ttyACM0
[ ]
[ ] Board Name     : OpenCR R1.0
[ ] Board Ver      : 0x17020800
[ ] Board Rev      : 0x00000000
[OK] flash erase   : 0.92s
[OK] flash write   : 1.84s
[OK] CRC Check     : 11A1E12 11A1E12 , 0.005000 sec
[OK] Download
[OK] jump to fw
turtlebot@turtlebot:~$
```

Figure 5.2.23: Example of successful firmware upload [55]

5.2.4 Raspberry Pi Camera Setup

To configure the Raspberry Pi Camera V2, the following steps were performed:

1. Update System and Install Dependencies:

The system's package list was updated, and essential dependencies were installed. These packages included build tools (Meson, CMake), multimedia frameworks (GStreamer, Qt), and Python libraries required to build the libcamera framework.

Commands:

```
sudo apt update
sudo apt install -y python3-pip git python3-jinja2 \
libboost-dev libgnutls28-dev openssl libtiff-dev pybind11-dev \
qtbase5-dev libqt5core5a libqt5widgets5 meson cmake \
python3-yaml python3-ply \
libglb2.0-dev libgstreamer-plugins-base1.0-dev
```

2. Install ROS2 Camera Driver:

The `ros-humble-camera-ros` package was installed to provide the necessary interface between `libcamera` and the ROS2 ecosystem. This driver allows the camera to publish image data as ROS2 topics.

Commands:

```
sudo apt install ros-humble-camera-ros
```

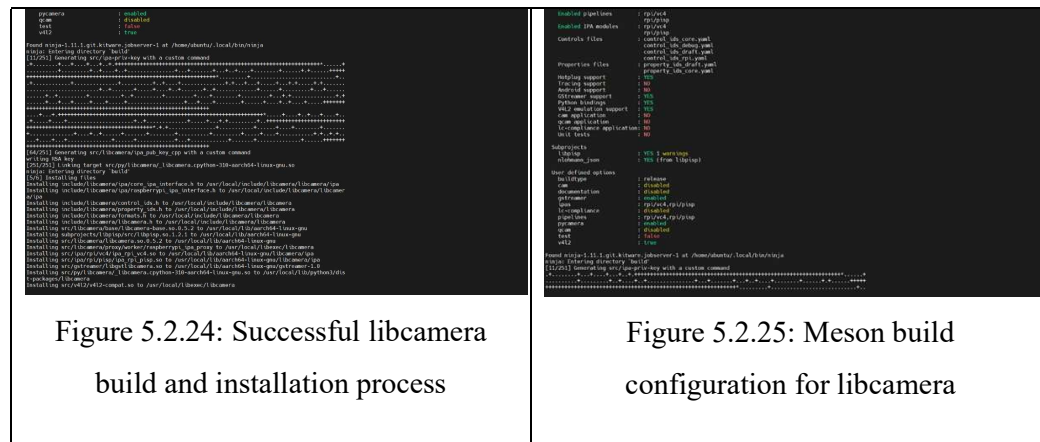
3. Clone and Build libcamera:

The latest `libcamera` source code was cloned from the official repository. It was then configured using `Meson` to enable Raspberry Pi-specific pipelines (VC4, PISP) and `GStreamer` support. Finally, the library was compiled and installed using `Ninja`.

Commands:

```
git clone https://github.com/raspberrypi/libcamera.git
cd libcamera
meson setup build --buildtype=release \
  -Dpipelines=rpi/vc4,rpi/pisp -Dipas=rpi/pisp \
  -Dv4l2=true -Dgstreamer=enabled -Dpycamera=enabled \
  -Dtest=false -Dlc-compliance=disabled \
  -Dcam=disabled -Dqcam=disabled -Ddocumentation=disabled
ninja -C build
sudo ninja -C build install
sudo ldconfig
```

The successful build and configuration are shown in Figure 5.2.24 and Figure 5.2.25, confirming the correct modules were enabled and installed without errors.



4. Launch the Camera Node in ROS2:

The ROS2 camera node was launched to activate the camera.

Commands:

ros2 launch camera_ros camera.launch.py

The successful launch is shown in Figure 5.2.7. As seen in Figures 5.2.8 and 5.2.9, new ROS2 topics (such as /image_raw) became active, confirming that the camera was successfully publishing its image stream.

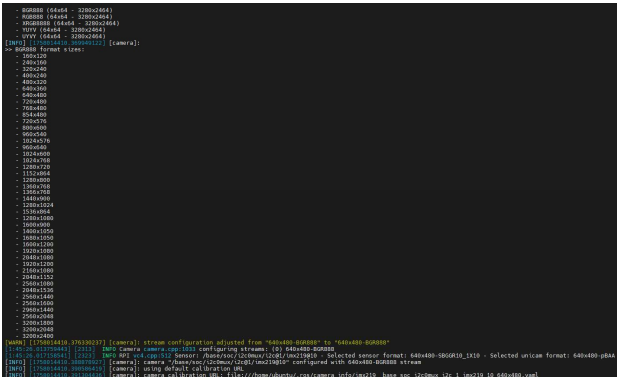
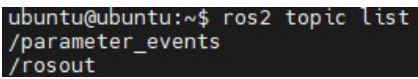
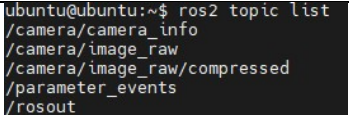


Figure 5.2.26: Successfully run camera node

 <p>ubuntu@ubuntu:~\$ ros2 topic list /parameter_events /rosout</p>	 <p>ubuntu@ubuntu:~\$ ros2 topic list /camera/camera_info /camera/image_raw /camera/image_raw/compressed /parameter_events /rosout</p>
Figure 5.2.27: Ros2 topic before launching camera node	Figure 5.2.28: Ros2 topic after launching camera node

5.2.5 Pan Tilt Servo Setup

The pan-tilt servo mechanism provides the camera with two degrees of freedom, and its control is managed by a custom ROS2 software package. This section details the software prerequisites, system architecture, and communication protocol developed to enable precise control of the servos

The implementation relies on several key libraries to function. For ROS2 integration, the ros-humble-rcldpy and ros-humble-std-msgs packages were installed, providing the necessary Python client library and standard message definitions, respectively. Direct hardware control of the servo motors on the Raspberry Pi was enabled by installing the RPi.GPIO Python library, which facilitates the generation of Pulse Width Modulation (PWM) signals required

Bachelor of Information Technology (Honours) Communications and Networking
Faculty of Information and Communication Technology (Kampar Campus), UTAR

for positioning the servos. Standard Python modules, including argparse and curses, were also utilized for command-line argument parsing and creating the text-based user interface.

The control system is designed with a distributed, two-node architecture to separate the low-level hardware control from the user interface. The script, `pan_tilt_ros_onefile.py`, can be executed in one of two modes as implemented in the main execution block shown in Figure 5.2.29:

```

247 # ===== MAIN =====
248 def main():
249     parser = argparse.ArgumentParser(description="ROS 2 pan/tilt (single-file)")
250     parser.add_argument('--mode', choices=['driver', 'teleop'], required=True,
251                         help="driver = run on Pi (GPIO); teleop = run on PC (keyboard)")
252     args = parser.parse_args()
253
254     rclpy.init()
255
256     if args.mode == 'driver':
257         node = PanTiltDriver()
258         try:
259             rclpy.spin(node)
260         finally:
261             node.destroy_node()
262             rclpy.shutdown()
263     else:
264         node = Teleop()
265         try:
266             node.run_ui()
267         finally:
268             node.destroy_node()
269             rclpy.shutdown()

```

Figure 5.2.29: Main Function of Pan Tilt Servo Setup

1. **Driver Mode:** This node runs on the TurtleBot3's Raspberry Pi. Its primary responsibility is to interface directly with the GPIO pins to control the servo motors. It subscribes to ROS2 command topics and translates incoming messages into the appropriate PWM signals. Concurrently, it publishes the current pan and tilt angles to provide state feedback to the rest of the system. This mode is activated by running the script with the `--mode driver` argument.

```

# ===== DRIVER =====
class PanTiltDriver(Node):
    """Runs on the Pi. Subscribes to step/center topics, publishes angles, drives GPIO."""
    def __init__(self):
        super().__init__('pan_tilt_driver')

        # Lazy-import GPIO so teleop can run on a PC without RPi.GPIO
        global GPIO
        import RPi.GPIO as GPIO

        GPIO.setmode(GPIO.BCM)
        GPIO.setup(TILT_PIN, GPIO.OUT)
        GPIO.setup(PAN_PIN, GPIO.OUT)
        self.GPIO = GPIO

        self.tilt_pwm = GPIO.PWM(TILT_PIN, FREQ_HZ)
        self.pan_pwm = GPIO.PWM(PAN_PIN, FREQ_HZ)
        self.tilt_pwm.start(0) # ldl = no pulses
        self.pan_pwm.start(0)

        # tracked angles (software model)
        self.tilt_deg = float(CENTER_DEG)
        self.pan_deg = float(CENTER_DEG)

        # pubs
        self.pub_tilt = self.create_publisher(Int32, '/tilt/angle', 10)
        self.pub_pan = self.create_publisher(Int32, '/pan/angle', 10)

        # subs
        self.create_subscription(Int32, '/tilt/step', self.on_tilt_step, 10)
        self.create_subscription(Int32, '/pan/step', self.on_pan_step, 10)
        self.create_subscription(Int32, '/tilt/center', self.on_tilt_center, 10)
        self.create_subscription(Int32, '/pan/center', self.on_pan_center, 10)

        # periodic publisher (keeps teleop UI fresh)
        self.create_timer(0.25, self.publish_angles)

        self.get_logger().info("Driver ready (GPIO18: Tilt, GPIO17: Pan).")

```

Figure 5.2.30: Driver Node Initialization

2. Teleoperation Mode: This node functions as the user interface and can be run on a remote PC or another machine. The teleoperation interface captures user key presses and publishes corresponding commands to the driver node, as illustrated in the control loop logic in Figure 5.2.10. It also subscribes to the angle topics published by the driver to display real-time positional feedback to the operator. This mode is launched using the `--mode teleop` argument.

```
# --- Key UI (curses) ---
def run_ui(self):
    import curses
    def loop(stdscr):
        curses.cbreak(); curses.noecho(); curses.curs_set(0)
        stdscr.nodelay(True)
        status = "Ready"
        while rclpy.ok():
            stdscr.clear()
            stdscr.addstr(0,0,"Teleop: 1/2/3 = Tilt UP/STOP/DOWN | 4/5/6 = Pan UP/STOP/DOWN")
            stdscr.addstr(1,0,"          c=center both t=center tilt p=center pan Esc=quit")
            stdscr.addstr(2,0,f"Tilt angle: {self.tilt_deg:3d}"  Pan angle: {self.pan_deg:3d}")
            stdscr.addstr(3,0,f"Status: {status}")
            stdscr.refresh()

            rclpy.spin_once(self, timeout_sec=0.0)
            ch = stdscr.getch()
            if ch == -1:
                time.sleep(0.02); continue
            if ch == 27: # Esc
                break
            elif ch == ord('1'):
                self.pub_tilt_step.publish(Int32(data=+1)); status="Tilt +1"
            elif ch == ord('2'):
                self.pub_tilt_step.publish(Int32(data=0)); status="Tilt STOP"
            elif ch == ord('3'):
                self.pub_tilt_step.publish(Int32(data=-1)); status="Tilt -1"
            elif ch == ord('4'):
                self.pub_pan_step.publish(Int32(data=+1)); status="Pan +1"
            elif ch == ord('5'):
                self.pub_pan_step.publish(Int32(data=0)); status="Pan STOP"
            elif ch == ord('6'):
                self.pub_pan_step.publish(Int32(data=-1)); status="Pan -1"
            elif ch in (ord('c'), ord('C')):
                self.pub_tilt_ctr.publish(Empty()); self.pub_pan_ctr.publish(Empty()); status="Center BOTH"
            elif ch in (ord('t'), ord('T')):
                self.pub_tilt_ctr.publish(Empty()); status="Center TILT"
            elif ch in (ord('p'), ord('P')):
                self.pub_pan_ctr.publish(Empty()); status="Center PAN"
            else:
                pass
        # graceful exit
    import curses
    curses.wrapper(loop)
```

Figure 5.2.31: Teleop Node Key Handling

Communication between the driver and teleoperation nodes is managed through a set of defined ROS2 topics. The teleop node publishes user commands to the `/pan/step`, `/tilt/step`, `/pan/center`, and `/tilt/center` topics. The driver node subscribes to these topics to execute movements. In the reverse direction, the driver node continuously publishes the servos' current positions to the `/pan/angle` and `/tilt/angle` topics, which the teleop node subscribes to for its status display.

The keyboard inputs for the teleoperation interface are mapped as follows: keys 1 and 3 control upward and downward tilt, while 4 and 6 control left and right pan. The c, t, and p keys are used to center both servos, only the tilt servo, or only the pan servo, respectively.

This distributed architecture provides a robust and flexible solution for controlling the pan-tilt mechanism, effectively decoupling the user interface from the hardware control layer within the ROS2 framework.

5.2.6 Audio and Speaker Setup

To enable real-time audio interaction, both microphone input and speaker output were integrated into the ROS2 system. This setup facilitates the capture of audio from a microphone, its publication as a ROS2 topic, and the simultaneous playback of audio streams received from other topics. This section details the system-level configuration and the ROS2 nodes developed for this purpose.

To identify the connected hardware, the `arecord -l` and `aplay -l` commands were used. The output, shown in Figure 5.2.32, confirmed that the USB microphone was detected as card 2, device 0. The available playback devices, as seen in Figure 5.2.33, included the HDMI output (card 0) and the 3.5mm headphone jack (card 1). Audio levels for both capture (microphone gain) and playback (speaker volume) were then configured using the `alsamixer` terminal utility, as depicted in Figure 5.2.34, to ensure optimal audio quality.

```
ubuntu@ubuntu:~$ arecord -l # capture devices (cards)
**** List of CAPTURE Hardware Devices ****
card 2: Device [USB PnP Sound Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Figure 5.2.2: Output of `arecord -l`

```
ubuntu@ubuntu:~$ aplay -l # playback devices (cards)
**** List of PLAYBACK Hardware Devices ****
card 0: b1 [bcm2835 HDMI 1], device 0: bcm2835 HDMI 1 [bcm2835 HDMI 1]
  Subdevices: 4/4
  Subdevice #0: subdevice #0
  Subdevice #1: subdevice #1
  Subdevice #2: subdevice #2
  Subdevice #3: subdevice #3
card 1: Headphones [bcm2835 Headphones], device 0: bcm2835 Headphones [bcm2835 Headphones]
  Subdevices: 4/4
  Subdevice #0: subdevice #0
  Subdevice #1: subdevice #1
  Subdevice #2: subdevice #2
  Subdevice #3: subdevice #3
```

Figure 5.2.33: Output of `aplay -l`

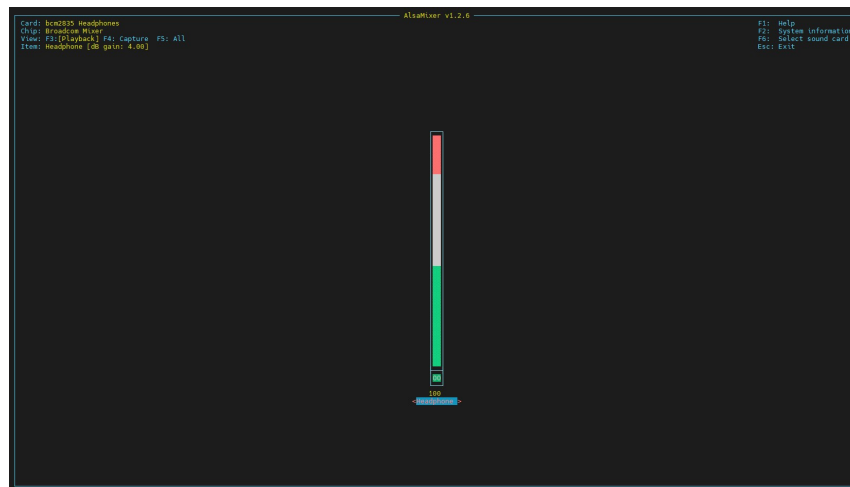


Figure 5.2.34: Volume Control through `alsamixer`

A custom Python script, `audio_speaker_ros.py`, was developed to create two ROS2 nodes that run concurrently to manage the audio stream.

1. **Microphone Publisher Node (MicPub):** The MicPub node captures audio by running `arecord` as a subprocess. To prevent blocking the main ROS2 process, the audio capture loop runs in a dedicated thread (Figure 5.2.35). In this loop, raw audio frames are read from the subprocess and published to the `/audio/mic` topic.

```
# -----
# Microphone publisher node
# -----
class MicPub(Node):
    def __init__(self):
        super().__init__('mic_pub')
        self.declare_parameter('mic_device', 'plughw:1,0')
        self.device = self.get_parameter('mic_device').get_parameter_value().string_value
        self.pub = self.create_publisher(UInt8MultiArray, '/audio/mic', 10)
        self.thread = threading.Thread(target=self.capture_loop, daemon=True)
        self.thread.start()
        self.get_logger().info(f"MicPub using device={self.device} @ {RATE}Hz {CHANNELS}ch {FMT}")

    def capture_loop(self):
        cmd = ['arecord', '-d', self.device, '-f', FMT, '-c', str(CHANNELS),
              '-r', str(RATE), '-t', 'raw', '-q']
        with subprocess.Popen(cmd, stdout=subprocess.PIPE, bufsize=0) as p:
            while rcpy.ok():
                chunk = p.stdout.read(BYTES_PER_FRAME)
                if not chunk:
                    break
                msg = UInt8MultiArray(data=list(chunk))
                self.pub.publish(msg)
```

Figure 5.2.35: MicPub function

2. **Speaker Subscriber Node (SpkSub):** The SpkSub node handles playback by piping audio data to an `aplay` subprocess. As shown in Figure 5.2.36, the `on_audio` callback function receives messages from the `/audio/speaker` topic and writes the data directly to the subprocess's standard input, resulting in immediate playback.

```
# -----
# Speaker subscriber node
# -----
class SpkSub(Node):
    def __init__(self):
        super().__init__('spk_sub')
        self.declare_parameter('spk_device', 'default')
        self.device = self.get_parameter('spk_device').get_parameter_value().string_value
        self.proc = subprocess.Popen(
            ['aplay', '-d', self.device, '-f', FMT, '-c', str(CHANNELS),
            '-r', str(RATE), '-t', 'raw', '-q', '-B', '200000', '-F', '4000'],
            stdin=subprocess.PIPE)
        self.sub = self.create_subscription(UInt8MultiArray, '/audio/speaker', self.on_audio, 10)
        self.get_logger().info(f"SpkSub using device={self.device} @ {RATE}Hz {CHANNELS}ch {FMT}")

    def on_audio(self, msg: UInt8MultiArray):
        try:
            self.proc.stdin.write(bytes(msg.data))
        except BrokenPipeError:
            pass

    def destroy_node(self):
        try:
            if self.proc and self.proc.stdin:
                self.proc.stdin.close()
            if self.proc:
                self.proc.terminate()
        finally:
            super().destroy_node()
```

Figure 5.2.36: SpkSub function

To enable simultaneous audio capture and playback, both the MicPub and SpkSub nodes are added to a `MultiThreadedExecutor`, as shown in Figure 5.2.37. A multithreaded ROS2 executor manages both nodes within a single process, ensuring that audio capture and playback can occur in parallel. The successful activation of the audio system was verified by inspecting the list of active ROS2 topics. As shown in Figures 5.2.38 and 5.2.39, the `/audio/mic` and

/audio/speaker topics appear after the script is run, confirming that the nodes are operational. This setup establishes a complete, bi-directional audio pipeline within the ROS2 environment, enabling functionalities such as speech recognition and remote audio communication.

```
# -----
# Main
# -----
def main():
    rclpy.init()
    mic = MicPub()
    spk = SpkSub()

    executor = rclpy.executors.MultiThreadedExecutor()
    executor.add_node(mic)
    executor.add_node(spk)

    try:
        executor.spin()
    finally:
        mic.destroy_node()
        spk.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Figure 5.2.37: Main Function of audio_bridge.py

```
ubuntu@ubuntu:~$ ros2 topic list
/parameter_events
/rosout
```

Figure 5.2.38: Ros2 topic list before running audio node

```
ubuntu@ubuntu:~$ ros2 topic list
/audio/mic
/audio/speaker
/parameter_events
/rosout
```

Figure 5.2.39: Ros2 topic list after running audio node

5.2.7 Telegram Setup

To establish a remote communication channel between the user and the robot, a Telegram Bot was created. This process was managed using BotFather, Telegram's official tool for bot administration.¹ Find BotFather: In the Telegram app, search for the official BotFather (with a blue checkmark) and start the chat.

The setup began by initiating a conversation with BotFather and using the /newbot command. After providing a unique name ("Ellie") and username for the bot, BotFather generated a unique API token. This token is critical as it authenticates the robot's software, allowing it to control the bot programmatically via Telegram's API. The token was securely stored for integration into the application's backend. The creation process is documented in Figure 5.2.40.

Following the initial setup, the bot's profile was customized using additional BotFather commands, such as /setdescription and /setuserpic, to configure its public-facing details (Figure 5.2.40). The final, user-facing bot page is displayed in Figure 5.2.41. This setup provides the necessary foundation for relaying messages and commands between the user's Telegram client and the robot's ROS2 system.

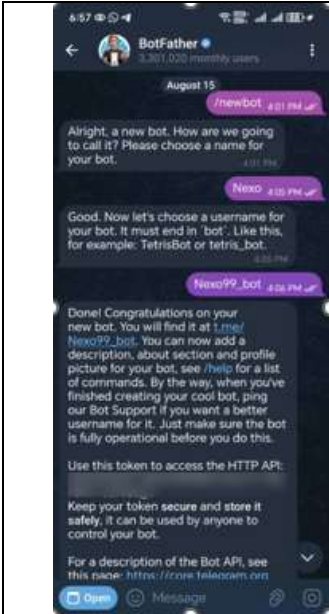


Figure 5.2.40: Create Bot

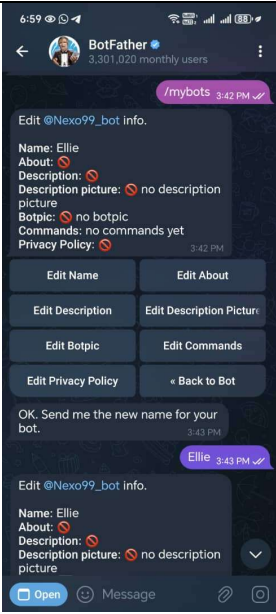


Figure 5.2.41: Edit the Details

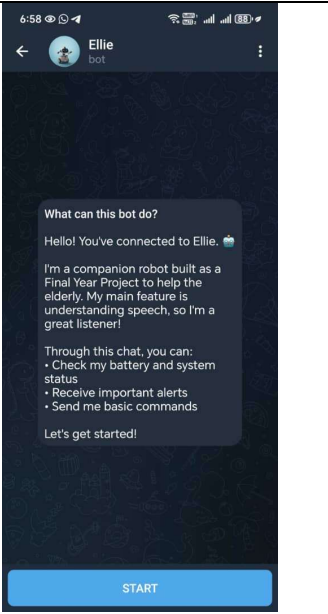


Figure 5.2.42: Example of Bot Page

The script's primary role is to route user commands and messages to the correct backend functions. Figure 5.2.43 shows the implementation of the main command handlers. This code snippet illustrates how text commands sent by the user in Telegram (e.g., /status) are received and mapped to specific functions within the robot's control system. This structure serves as the primary dispatcher for all user-initiated actions.

```

def handle_telegram_command(command: str, chat_id: str, user_id: str = None) -> str:
    """Handle incoming Telegram commands"""
    command = command.lower().strip()

    # Check if user is authorized (if TELEGRAM_ALLOWED_USERS is configured)
    if TELEGRAM_ALLOWED_USERS and user_id and str(user_id) not in TELEGRAM_ALLOWED_USERS:
        return "❌ You are not authorized to use this bot."

    if command == "/start":
        return "👋 Welcome to TurtleBot3 Telegram Bot!"

    I'm your robot's remote interface. Here's what I can do:

    # Available Commands:
    /help - Show this help message
    /status - Get robot status
    /photo - Take a photo
    /chat - Start interactive chat with AI
    /test - Test bot connectivity

    🚀 Ready to help! Send me a command to get started.

    elif command == "/help":
        return "📋 Available Commands:"

    /start - Welcome message
    /help - Show this help
    /status - Get robot status (battery, connection, camera)
    /photo - Capture and send current photo
    /chat - Start interactive chat with AI
    /test - Test bot connectivity

    # If you're a developer, you can use these commands:
    # Chat: Send any text message to chat with the AI assistant.

    elif command == "/status":
        return get_robot_status_for_telegram()

    elif command == "/photo":
        try:
            photo_bytes = get_latest_image()
            if photo_bytes:
                caption = f"📷 Robot Camera Snapshot {datetime.now().strftime('%H:%M:%S')}"
                telegram_bot_send_message(caption, chat_id, photo_bytes)
                return "📸 Photo sent!"
            else:
                return "❌ No camera image available. Camera may be offline or not initialized."
        except Exception as e:
            return f"❌ Error capturing photo: {str(e)}"

    elif command == "/test":
        return "✅ Test Test Successful!"

```

Figure 5.2.43: Create Telegram bot command with functionalities

In addition to explicit commands, the bot integrates conversational AI by prompting the text to Gemini. The code snippet in Figure 5.2.44 details the AI reply function. This function processes any free-text messages from the user, forwards them to a language model for processing, and relays the generated response back to the user, enabling natural and unscripted conversation.

```

def handle_telegram_chat(message_text: str, chat_id: str, user_id: str = None) -> str:
    """Handle interactive chat with Gemini AI"""
    # Check if user is authorized
    if TELEGRAM_ALLOWED_USERS and user_id and str(user_id) not in TELEGRAM_ALLOWED_USERS:
        return "❌ You are not authorized to use this bot."

    try:
        # Import the STT/ITS core for Gemini integration
        import test_speech as stt_its_core

        # Add robot content to the message
        robot_content = f"""You are an AI assistant for a TurtleBot3 robot. The user is chatting with you via Telegram.

Current robot status:
- Battery: {telemetry_data.get('battery_level', 0.0)}%
- Connected: {telemetry_data.get('connected', False)}
- Camera: {telemetry_data.get('camera_status', 'Unknown')}

User message: {message_text}

Please respond in a helpful, friendly way. You can mention the robot's capabilities like taking photos, checking status, or navigation if relevant to the conversation.

# Get AI response using Gemini
ai_response = stt_its_core.gemini_chat(robot_content)

# Add a small indication that this is from the robot
return f"🤖 {ai_response}"

    except ImportError:
        return "❌ AI chat feature is not available. Please check the STT/ITS system configuration."
    except Exception as e:
        log_error(f"Error in Telegram chat: {e}")
        return f"❌ Sorry, I encountered an error: {str(e)}"

```

Figure 5.2.44: Telegram bot with AI reply function

5.3 Setting and Configuration

5.3.1 System Launch and Robot Bringup:

This section outlines the procedure to activate the TurtleBot3 and all its custom software components, making the robot fully operational. The process consists of a one-time configuration of the environment, followed by the execution of an automated launch script that starts all necessary services.

Before the robot can be launched, the environment on both the Raspberry Pi and the remote PC must be correctly configured. This involves two key steps which is Network Setup and Environment Variables. A static IP address (192.168.1.100) is assigned to the robot to ensure a stable and predictable connection point. Connectivity is verified from the remote PC using ping and ssh. Besides, all required setup commands, including sourcing ROS2 workspaces and exporting variables like TURTLEBOT3_MODEL and ROS_DOMAIN_ID, are added to the ~/.bashrc file. This ensures that any new terminal session is automatically configured with the correct environment, as shown in Figure 5.3.1.

```
source /opt/ros/humble/setup.bash
source ~/turtlebot3_ws/install/setup.bash
export ROS_DOMAIN_ID=30 #TURTLEBOT3
export LDS_MODEL=LDS-01
export TURTLEBOT3_MODEL=burger

alias robot-start="~/robot_start.sh"
alias robot-stop="tmux kill-session -t robot"
alias robot-attach="tmux attach -t robot"
```

Figure 5.3.1: Environment setup commands in the .bashrc file (Turtlebot3).

To simplify the complex process of launching multiple ROS2 nodes, an automated shell script named robot_start.sh was created. This script uses tmux, a terminal multiplexer, to launch and manage all required robot-side processes concurrently within a single, organized session. This method is efficient and eliminates the need to open and manage multiple terminal windows manually.

The script, detailed in Figure 5.3.2, performs the following actions: First, it creates a tmux session named "robot." Then, it launches five essential nodes, each in its own labeled window, including: "bringup" (the core TurtleBot3 hardware drivers), "camera" (the Raspberry Pi camera node), "audio" (the custom microphone and speaker bridge), "servo" (the pan-tilt servo driver), and "rosbridge" (the WebSocket server for web interface communication).


```
#!/bin/bash
set -e

# --- Ensure ROS is sourced even if .bashrc didn't run (desktop shortcut, cron, etc.)
if [ -f /opt/ros/humble/setup.bash ]; then
    source /opt/ros/humble/setup.bash
fi

# If you have a workspace overlay, uncomment:
# [ -f ~/turtlebot3_ws/install/setup.bash ] && source ~/turtlebot3_ws/install/setup.bash

# --- Env
export ROS_DOMAIN_ID="${ROS_DOMAIN_ID:-30}"
# If you rely on Envs and they're not already in ~/.bashrc, uncomment:
# export TURTLEBOT3_MODEL=burger
# export LDS_MODEL=LDS-01

# --- Preflight checks (friendlier errors)
for cmd in tmux ros2 python3 dd; do
    command -v "$cmd" >/dev/null 2>&1 || { echo "Missing: $cmd"; exit 1; }
done

SESSION=robot

# If already running, just attach
if tmux has-session -t "$SESSION" 2>/dev/null; then
    exec tmux attach -t "$SESSION"
fi

# Window 1: Audio bridge
tmux new-session -d -s "$SESSION" -n audio \
    "python3 /home/ubuntu/testaudio/audio_bridge.py --ros-args -p mic_device:=plughw:1,0 -p sph_device:=default"

# Window 2: Servo driver
tmux new-window -t "$SESSION" -n servo \
    "python3 /home/ubuntu/pan_tilt_ros_onefile.py --mode driver"

# Window 3: Turtlebot3 bringup
tmux new-window -t "$SESSION" -n bringup \
    "ros2 launch turtlebot3_bringup robot.launch.py"

# Window 4: lib camera (add device params if you need a specific camera)
tmux new-window -t "$SESSION" -n camera \
    "ros2 run camera_ros camera_node --ros-args --params-file camera_params.yaml"

# Window 5: rosbridge server (WebSocket)
tmux new-window -t "$SESSION" -n rosbridge \
    "ros2 launch rosbridge_server rosbridge_websocket_launch.xml"

# Start on the 'audio' window
exec tmux attach -t "$SESSION"
```

Figure 5.3.2: robot_start.sh script

The entire robot system is started by executing a single command in the robot's terminal: `“./robot_start.sh”`. Upon execution, the script initializes the tmux session, and all nodes begin running in the background. Figure 5.3.3 shows the resulting tmux interface, with each window corresponding to a running process. The user can easily switch between windows to monitor the status of each component. With this single command, the robot is made fully operational and is ready to execute high-level tasks.

```
[INFO] [launch]: default logging verbosity is set to INFO
urdf file name : turtlebot3_burger.urdf
[INFO] [robot_state_publisher-1]: process started with pid [1414]
[INFO] [hls_laser_publisher-2]: process started with pid [1416]
[INFO] [turtlebot3_ros-3]: process started with pid [1418]
[INFO] [turtlebot3_ros-3] [INFO] [75844395.3731723] [turtlebot3_node]: Init TurtleBot3 Node Main
[INFO] [turtlebot3_ros-3] [INFO] [75844394.966705263] [turtlebot3_node]: Init DynamixelSDKWrapper
[INFO] [turtlebot3_ros-3] [INFO] [75844394.99994820] [DynamixelSDKWrapper]: Succeeded to open the port(/dev/ttyACM0)
[INFO] [turtlebot3_ros-3] [INFO] [75844395.037564846] [DynamixelSDKWrapper]: Succeeded to change the baudrate!
[INFO] [turtlebot3_ros-3] [INFO] [75844395.091205420] [turtlebot3_node]: Start Calibration of Gyro
[INFO] [hls_laser_publisher-2] [INFO] [75844395.255256594] [hls_laser_publisher]: Init hls_laser_publisher Node Main
[INFO] [hls_laser_publisher-2] [INFO] [75844395.264905123] [hls_laser_publisher]: port : /dev/ttyUSB0 frame_id : base_scan
[INFO] [robot_state_publisher-1] [INFO] [75844395.349542410] [robot_state_publisher]: got segment base_footprint
[INFO] [robot_state_publisher-1] [INFO] [75844395.551545989] [robot_state_publisher]: got segment base_link
[INFO] [robot_state_publisher-1] [INFO] [75844395.551651797] [robot_state_publisher]: got segment base_scan
[INFO] [robot_state_publisher-1] [INFO] [75844395.552830797] [robot_state_publisher]: got segment caster_back_link
[INFO] [robot_state_publisher-1] [INFO] [75844395.552523112] [robot_state_publisher]: got segment imu_link
[INFO] [robot_state_publisher-1] [INFO] [75844395.552678221] [robot_state_publisher]: got segment wheel_left_link
[INFO] [robot_state_publisher-1] [INFO] [75844395.552842067] [robot_state_publisher]: got segment wheel_right_link
[INFO] [turtlebot3_ros-3] [INFO] [75844395.094151513] [turtlebot3_node]: Calibration end
[INFO] [turtlebot3_ros-3] [INFO] [75844395.094750623] [turtlebot3_node]: Add Motors
[INFO] [turtlebot3_ros-3] [INFO] [75844395.097218953] [turtlebot3_node]: Add Wheels
[INFO] [turtlebot3_ros-3] [INFO] [75844395.103213243] [turtlebot3_node]: Add Sensors
[INFO] [turtlebot3_ros-3] [INFO] [75844395.218113997] [turtlebot3_node]: Succeeded to create battery state publisher
[INFO] [turtlebot3_ros-3] [INFO] [75844395.246126520] [turtlebot3_node]: Succeeded to create imu publisher
[INFO] [turtlebot3_ros-3] [INFO] [75844395.303845989] [turtlebot3_node]: Succeeded to create sensor state publisher
[INFO] [turtlebot3_ros-3] [INFO] [75844395.322450187] [turtlebot3_node]: Succeeded to create joint state publisher
[INFO] [turtlebot3_ros-3] [INFO] [75844395.321850481] [turtlebot3_node]: Add Devices
[INFO] [turtlebot3_ros-3] [INFO] [75844395.322335263] [turtlebot3_node]: Succeeded to create motor power server
[INFO] [turtlebot3_ros-3] [INFO] [75844395.355807170] [turtlebot3_node]: Succeeded to create reset server
[INFO] [turtlebot3_ros-3] [INFO] [75844395.375646699] [turtlebot3_node]: Succeeded to create sound server
[INFO] [turtlebot3_ros-3] [INFO] [75844395.412169966] [turtlebot3_node]: Run!
[INFO] [turtlebot3_ros-3] [INFO] [75844395.702659340] [diff_drive_controller]: Init Odometry
[INFO] [turtlebot3_ros-3] [INFO] [75844395.843550640] [diff_drive_controller]: Run!
```

Figure 5.3.3: The tmux session after a successful launch, showing all active nodes.

The remote PC's `~/.bashrc` file must be configured with the same ROS2 environment variables (e.g., `ROS_DOMAIN_ID=30`, `TURTLEBOT3_MODEL=burger`) to ensure seamless communication with the robot for this project.

```
# ROS2 Setup
source /opt/ros/humble/setup.bash
source ~/turtlebot3_ws/install/setup.bash
source /usr/share/gazebo/setup.sh
source ~/audio_ws/install/setup.bash

# ROS2 Environment Variables
export ROS_DOMAIN_ID=30
export TURTLEBOT3_MODEL=burger
export LDS_MODEL=LDS-01
```

Figure 5.3.4: Environment setup commands in the .bashrc file (RemotePC).

5.3.2 SLAM Navigation and Mapping

To enable the "Ellie" robot to operate autonomously, it was necessary to map its primary operational environment the Final Year Project (FYP) Laboratory and implement a system for programmatic, goal-based navigation. This process was divided into two major phases: mapping the lab using SLAM and then developing a Python-based navigation script using the Nav2 SimpleCommanderAPI.

Phase 1: Mapping the Environment

The first phase focused on creating a precise 2D digital map of the FYP Laboratory. This was accomplished through a three-step process: performing a live SLAM scan, saving the raw map data, and finally, post-processing the map image to enhance its quality for navigation.

The mapping process was controlled entirely from the Remote PC, which has the necessary processing power to run the SLAM algorithm. The TurtleBot3, acting as a mobile sensor platform, was physically located in the lab.

Two ROS2 nodes were launched in separate terminals on the Remote PC:

- **Cartographer Node:** This node initiated the SLAM algorithm, ready to receive sensor data from the robot.

Command:
ros2 launch turtlebot3_cartographer cartographer.launch.py

- **Teleoperation Node:** This node allowed for manual keyboard control to drive the robot through the environment.

Command:
ros2 run turtlebot3_teleop teleop_keyboard


```
kokfu@FYP: $ ros2 run turtlebot3_teleop teleop_keyboard

Control Your TurtleBot3!
-----
Moving around:
    w
a   s   d
    x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi :
~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi
: ~ 1.82)

space key, s : force stop

CTRL-C to quit
```

Figure 5.3.5: Launch teleoperation on another terminal of Remote PC

The robot was carefully teleoperated throughout the lab, ensuring its LiDAR sensor scanned every wall, doorway, and static obstacle. During this process, the map's construction was visualized in real-time within RViz to monitor progress and ensure complete coverage.



Figure 5.3.6: Robot Mapping the environment

Once the live scan was complete, the resulting map was saved from the Remote PC using the `nav2_map_server` utility:

Command:

```
ros2 run nav2 map_server map_saver cli -f ~/map
```

This command produced `map.pgm`, an image file representing the unprocessed map, and its corresponding `map.yaml` configuration file. This generated `map.pgm` and `map.yaml` files, which serve as the static map for all subsequent navigation tasks.



Figure 5.3.7: map.pgm and map.yaml

Once the scan was complete, the raw map was saved as map.pgm by default. This initial map contained sensor noise and minor structural gaps. To optimize it for the navigation stack, it was edited in GIMP 2.0. Stray pixels were removed, and walls were reinforced to create a clean, unambiguous map. This post-processing step resulted in a much cleaner and more robust final map, map.pgm (Figure 5.3.8). This refined map ensures that the Nav2 path planner operates more efficiently and reliably, preventing the robot from attempting to plan routes through non-existent wall gaps.

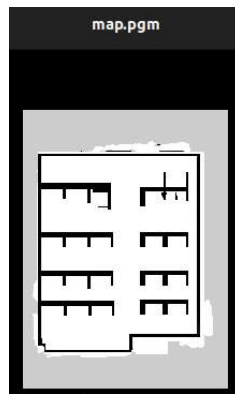


Figure 5.3.8: Final map.pgm after processing

Phase 2: Programmatic Navigation via SimpleCommanderAPI

Instead of relying on manual goal-setting through RViz, a more robust and application-oriented approach was implemented using the Nav2 SimpleCommanderAPI. This Python API allows the robot to navigate to specific, pre-defined locations through code, which is essential for an autonomous companion robot. First of all, the navigation stack was launched with the newly created map.

```
Command:
ros2 launch turtlebot3_navigation2 navigation2.launch.py \
  use_sim_time:=False \
  map:=/home/kokfu/turtlebot_webui/v3/config/map.yaml
```

CHAPTER 5

Several key locations within the lab, such as "Home", "Point A" and "Point B" were identified for the robot's tasks. The robot was manually driven to each of these spots, and its precise map coordinates (x, y) and orientation (z, w) were recorded. These named poses were stored for later use in the control script, as shown in Table 5.3.1.

Table 5.3.1: Navigation Points

Point	x	y
Home	0.2600000381469727	0.7956252288818364
Point A	2.7100000381469727	0.6956252288818363
Point B	0.6100000381469728	0.45437477111816404

A Python script was developed using the BasicNavigator class from the nav2_simple_commander library. This script manages the lifecycle of a navigation task: setting the robot's initial pose, clearing the costmap, and sending it to a goal.

With this script, commanding the robot is as simple as calling a function. For example, a command received from the User to "go to Point A" would trigger the script to send the "Point A" coordinates to Nav2. The system then handles the path planning and execution autonomously, as visualized in Figure 5.3.9. This programmatic approach makes the robot's navigation reliable, repeatable, and easily integrated with other systems like the Telegram interface or a voice command module.

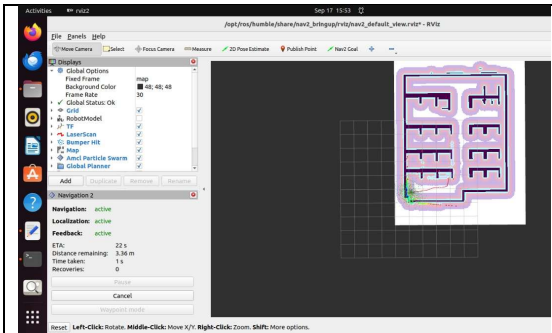


Figure 5.3.9: Provide navigation goal

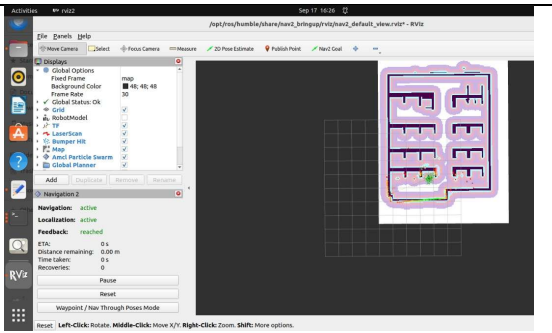


Figure 5.3.10: Goal Reached

To visualize the robot's navigation on a digital map, it was necessary to convert between map coordinates (in meters) and pixel coordinates. This conversion process allows the robot's position to be displayed on the map within the user interface.

```
# Convert map coordinates to pixel coordinates
def map_to_pixel(mx: float, my: float):
    px = (mx - ORIGIN_X) / MAP_RESOLUTION
    py = (ORIGIN_Y + (MAP_H - 1) * MAP_RESOLUTION - my) / MAP_RESOLUTION
    return px, py
```

Figure 5.3.11: Map to Pixel Conversion

The BasicNavigator from the nav2_simple_commander library is utilized to send the robot to a goal pose. It abstracts the complexities of interacting with ROS 2's navigation stack and makes it easier to command the robot.

```
# Nav2 Simple Commander setup
self.navigators = BasicNavigator()
```

Figure 5.3.12: Nav2 Simple Commander setup

The initial pose is sent to ROS 2 using the /initialpose topic, which is critical for setting up AMCL (Adaptive Monte Carlo Localization) and initiating navigation.

```
# Publish initial pose for AMCL
def publish_initialpose(self, x: float, y: float, yaw_rad: float):
    msg = PoseWithCovarianceStamped()
    msg.header.frame_id = 'map'
    msg.pose.pose.position.x = float(x)
    msg.pose.pose.position.y = float(y)
    msg.pose.pose.orientation.z = math.sin(yaw_rad / 2.0)
    msg.pose.pose.orientation.w = math.cos(yaw_rad / 2.0)
    self.initpose_pub.publish(msg)
```

Figure 5.3.13: Publish Initial Pose

The navigate_to_sync function sends the robot to a target location, waits for the task to complete, and returns the result. This function is essential for performing autonomous navigation tasks.

```

# Navigate to a target goal
def navigate_to_sync(self, x: float, y: float, yaw_rad: float = 0.0) -> dict:
    goal = _make_pose(x, y, yaw_rad)
    try:
        self.navigator.goToPose(goal)
    except Exception as e:
        return {'accepted': False, 'status': -1, 'error': f'goToPose failed: {e}'}

    t0 = time.time()
    while not self.navigator.isTaskComplete():
        if (time.time() - t0) > 300.0: # Timeout after 5 minutes
            self.navigator.cancelTask()
            return {'accepted': True, 'status': -4, 'error': 'result timeout'}
        time.sleep(0.1)

    result = self.navigator.getResult()
    return {'accepted': True, 'status': _name(result), 'status_code': _code(result)}

```

Figure 5.3.14: Navigate to goal function

The API endpoints allow users to interact with the robot's navigation system. These endpoints provide the ability to initialize the robot's pose, send movement commands, and check the robot's current status.

```

# Flask routes for robot control
@nav_bp.post('/init_pose')
def api_init_pose():
    data = request.get_json(force=True) or {}
    x = float(data.get('x', 0.0))
    y = float(data.get('y', 0.0))
    yaw = float(data.get('yaw', 0.0))
    _node.publish_initialpose(x, y, yaw)
    return jsonify({'ok': True})

@nav_bp.post('/goal')
def api_goal():
    data = request.get_json(force=True) or {}
    x = float(data['x'])
    y = float(data['y'])
    yaw = float(data.get('yaw', 0.0))
    result = _node.navigate_to_sync(x, y, yaw)
    return jsonify(result)

@nav_bp.get('/pose')
def api_pose():
    p = _node.get_last_pose()
    if p is None:
        return jsonify({'has_pose': False})
    x, y, yaw = p
    return jsonify({'has_pose': True, 'x': x, 'y': y, 'yaw': yaw})

```

Figure 5.3.15: Flask API for Navigation Control

The API also allows users to save and navigate to named points, which are defined by specific coordinates and yaw angles. This feature enables easy navigation to frequently used locations within the lab.

```

# Save a named point
def set_point(self, name: str, x: float, y: float, yaw: float):
    self.named_points[name] = {'x': float(x), 'y': float(y), 'yaw': float(yaw)}
    _save_points(self.named_points)

# Get a saved named point
def get_point(self, name: str):
    return self.named_points.get(name)

```

Figure 5.3.16: Save and get saved points

Later, once we run the Web UI, the navigation node startup together will ensure AMCL and initial pose ready as shown in Figure 5.3.17.

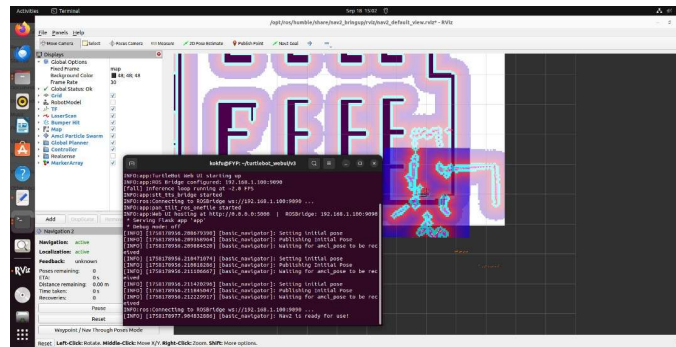


Figure 5.3.17: WebUI startup, subscribed to Nav2

5.3.3 Fall Detection AI Model

This subchapter provides a detailed description of the implementation and setup of the Fall Detection AI Models used in the TurtleBot3 WebUI. The system integrates two distinct models for robust fall detection: the PoseNet-based Fall Detection Model (Primary) and the OpenPifPaf-based Fall Detection Model (Secondary). The section covers the necessary files, technical setup, model logic, code examples, and the real-time operation of both models. Additionally, it discusses the integration of these models into the system, ensuring effective fall detection performance.

1. PoseNet-based Fall Detection Model

The PoseNet-based model relies on TensorFlow Lite for human pose estimation. To implement this model, users should first download the required files from the GitHub repository. These files include `fall_prediction.py`, which serves as the main API for making fall predictions, and `fall_detect.py`, which contains the core logic for fall detection. Additionally, users need to download the `posenet_mobilenet_v1_100_257x257_multi_kpt_stripped.tflite` model, which is the TensorFlow Lite model used by PoseNet.

Once the repository is downloaded, users will have access to all the necessary files and models to set up the system. The next step is configuring the model, which can be done by updating the `_fall_detect_config()` function to include the path to the downloaded TensorFlow Lite model. The following code snippet illustrates how to set up the configuration for the PoseNet model:

```

7 def _fall_detect_config():
8
9     _dir = os.path.dirname(os.path.abspath(__file__))
10     _good_tflite_model = os.path.join(
11         _dir,
12         'ai_models/posenet_mobilenet_v1_100_257x257_multi_kpt_stripped.tflite'
13     )
14     _good_edgetpu_model = os.path.join(
15         _dir,
16         'ai_models/posenet_mobilenet_v1_075_721_1281_quant_decoder_edgetpu.tflite'
17     )
18     _good_labels = 'ai_models/pose_labels.txt'
19     config = {
20         'model': {
21             'tflite': _good_tflite_model,
22             'edgetpu': _good_edgetpu_model,
23         },
24         'labels': _good_labels,
25         'top_k': 3,
26         'confidence_threshold': 0.6,
27         'model_name': 'mobilenet'
28     }
29     return config

```

Figure 5.3.18: Configuration for PoseNet model

The system processes three consecutive frames and compares pose keypoints in each frame for fall detection:

```

32 def Fall_prediction(img_1, img_2, img_3=None):
33
34     config = _fall_detect_config()
35     result = None
36
37     fall_detector = FallDetector(**config)
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82

```

```

65     if img_3:
66
67         time.sleep(fall_detector.min_time_between_frames)
68         process_response(fall_detector.process_sample(image=img_3))
69
70         if len(result) == 1:
71
72             category = result[0]['label']
73             confidence = result[0]['confidence']
74             angle = result[0]['leaning_angle']
75             keypoint_corr = result[0]['keypoint_corr']
76
77             dict_res = {}
78             dict_res["category"] = category
79             dict_res["confidence"] = confidence
80             dict_res["angle"] = angle
81             dict_res["keypoint_corr"] = keypoint_corr
82             return dict_res

```

Figure 5.3.19: Three-Frame Temporal Analysis

The PoseNet-based model is integrated into the TurtleBot3 Web UI, where it continually analyzes frames from a live video feed. When a fall is detected, it updates the system state and triggers an alert.

```

234 # Process with fall-detection-main (internal model)
235 if FALL_MODEL_MODE in ('internal', 'hybrid'):
236     # Snapshot 3 frames (oldest->newest)
237     img1, img2, img3 = infer_buf[0], infer_buf[1], infer_buf[2]
238     # Model expects PIL image inputs
239     result = Fall_prediction(img1, img2, img3)
240     is_fall, label, conf, angle = _classify(result, CONF_THRESH)
241     with image_lock:
242         # update internal state only; final combination happens in generate_frames()
243         fall_state['label'] = 'FALL' if is_fall else 'no fall'
244         fall_state['confidence'] = conf
245         fall_state['angle'] = angle
246         fall_state['ts'] = datetime.now().strftime('%H:%M:%S')
247     # Log fall detection result
248     prev = fall_state.get('last_status', 'no fall')
249     curr = 'FALL' if is_fall else 'no fall'
250     # Log only on transitions, or periodic high-confidence falls
251     if curr != prev or (is_fall and conf >= max(0.9, CONF_THRESH + 0.15)):
252         log_fall_detection(f"[internal] {curr}", conf, angle)
253
254     # Send Telegram alert when fall is detected on live camera (with cooldown)
255     current_time = time.time()
256     if is_fall and conf >= CONF_THRESH and (current_time - fall_state.get('last_alert_ts', 0)) > TELEGRAM_ALERT_COOLDOWN:
257         try:
258             photo_bytes = get_latest_jpeg()
259             alert_message = f"🚨 FALL DETECTED! 🚨\n\nConfidence: {conf:.2f}\nAngle: {angle:.1f}\nTime: {datetime.now().strftime('%H:%M:%S')}"
```

Figure 5.3.20: Real-time Fall Detection with PoseNet

The following code integrates the PoseNet model into the real-time processing loop:

```

87 def fall_worker():
88     """Run the fall detection models at ~INFER_FPS using frames queued by update_image_data()."""
89     period = 1.0 / max(0.1, INFER_FPS)
90
91     while True:
92         time.sleep(period)
93         try:
94             if len(infer_buf) < 3:
95                 continue
96
97             # Process with fall-detection-main (internal model)
98             if FALL_MODEL_MODE in ('internal', 'hybrid'):
99                 # Snapshot 3 frames (oldest->newest)
100                 img1, img2, img3 = infer_buf[0], infer_buf[1], infer_buf[2]
101                 # Model expects PIL image inputs
102                 result = Fall_prediction(img1, img2, img3)
103                 is_fall, label, conf, angle = _classify(result, CONF_THRESH)
104
105                 with image_lock:
106                     fall_state['label'] = 'FALL' if is_fall else 'no fall'
107                     fall_state['confidence'] = conf
108                     fall_state['angle'] = angle
109                     fall_state['ts'] = datetime.now().strftime('%H:%M:%S')
```

Figure 5.3.21: fall_worker function in app.py

This loop continuously captures frames, processes them for fall detection, and sends alerts if necessary.

2. OpenPifPaf-based Fall Detection Model

The OpenPifPaf-based model utilizes the OpenPifPaf framework for pose estimation and an LSTM classifier for fall detection. To set up this model, users need to download the essential files from the GitHub repository. The key files required for integration include `fall_detector.py`, which contains the main detection class for integrating the OpenPifPaf model, and `lstm_weights.sav`, which holds the trained weights for the LSTM fall classifier. Additionally, `model.py` defines the LSTM model used for classifying whether a fall has occurred based on the pose data provided by OpenPifPaf. These files work together to provide accurate fall detection by analyzing the geometry of human poses, focusing on aspects like torso tilt and body aspect ratio.

Once these files are downloaded, users can begin integrating the OpenPifPaf model into the system and configure it to analyze real-time frames for fall detection.

```

140 class HumanFallDetectionWrapper:
141     def __init__(self, disable_cuda: bool = True, confidence_threshold: float = 0.7):
142         self.available = HUMAN_FALL_DETECTION_AVAILABLE
143         self.disable_cuda = disable_cuda
144         self.confidence_threshold = confidence_threshold
145         self.fall_detector = None
146         self.predictor = None
147         self.frame_buffer = deque(maxlen=3)
148
149         # Fall detection state
150         self.consec_fall_like = 0
151         self.confirm_frames = 3 # Consecutive frames for confirmation

```

Figure 5.3.22: Initialization for HumanFallDetection Model

The system calculates the aspect ratio and torso tilt to determine whether a person has fallen. The logic involves analyzing keypoints and deriving geometric features to assess whether the pose corresponds to a fall.

```

210 def _analyze_pose(self, keypoints: np.ndarray) -> tuple:
211     """
212     Analyze pose keypoints to detect fall.
213
214     Args:
215         keypoints: Array of keypoints with shape (17, 3) [x, y, confidence]
216
217     Returns:
218         Tuple of (is_fall, confidence, angle)
219     """
220     try:
221         # Check if we have enough valid keypoints
222         valid = keypoints[:, 2] > 0.1
223         if valid.sum() < 5:
224             return False, 0.0, 0.0
225
226         # Calculate bounding box aspect ratio
227         xs = keypoints[valid, 0]
228         ys = keypoints[valid, 1]
229         w = (xs.max() - xs.min()) + 1e-6
230         h = (ys.max() - ys.min()) + 1e-6
231         aspect = h / w # small aspect ratio = lying down
232
233         # Check for required keypoints (shoulders and hips)
234         needed = [5, 6, 11, 12] # 5: left shoulder, right shoulder, left hip, right hip
235         if not all(i < len(keypoints) for i in needed):
236             return False, 0.0, 0.0
237
238         if min(keypoints[5, 2], keypoints[6, 2], keypoints[11, 2], keypoints[12, 2]) <= 0.1:
239             return False, 0.0, 0.0
240
241         # Calculate torso tilt angle
242         shoulder_mid = np.array([
243             (keypoints[5, 0] + keypoints[6, 0]) / 2.0,
244             (keypoints[5, 1] + keypoints[6, 1]) / 2.0
245         ])
246
247         hip_mid = np.array([
248             (keypoints[11, 0] + keypoints[12, 0]) / 2.0,
249             (keypoints[11, 1] + keypoints[12, 1]) / 2.0
250         ])
251
252         # Vector from shoulders to hips
253         torso_vector = hip_mid - shoulder_mid
254         up_vector = np.array([0.0, -1.0]) # Upward direction
255
256         # Calculate angle between torso and vertical
257         torso_norm = torso_vector / (np.linalg.norm(torso_vector) + 1e-6)
258         cos_angle = float(np.clip(np.dot(torso_norm, up_vector), -1.0, 1.0))
259         angle_deg = np.degrees(np.arccos(cos_angle))
260
261         # Fall detection criteria
262         aspect_threshold = 0.75 # Lower aspect ratio = more horizontal
263         angle_threshold = 40.0 # Higher angle = more tilted
264
265         is_fall = (aspect < aspect_threshold) and (angle_deg > angle_threshold)
266
267         # Calculate confidence based on how extreme the values are
268         aspect_confidence = min(1.0, (aspect_threshold - aspect) / aspect_threshold)
269         angle_confidence = min(1.0, (angle_deg - angle_threshold) / (90.0 - angle_threshold))
270         confidence = (aspect_confidence + angle_confidence) / 2.0
271
272         return is_fall, confidence, angle_deg
273
274     except Exception as e:
275         logging.error(f"Error analyzing pose: {e}")
276         return False, 0.0, 0.0

```

Figure 5.3.23: Pose Analysis in the OpenPifPaf-based Model

The OpenPifPaf model processes individual frames and requires 3 consecutive frames to confirm a fall. This ensures that only sustained falls trigger detection, reducing the risk of false positives from transient pose changes.

```

367 # Process with HumanFallDetection (direct integration)
368 if FALL_MODEL_MODE in ('humanfall', 'hybrid') and HumanFallDetector is not None:
369     # Get the latest frame from the buffer
370     latest_frame = infer_buf[-1]
371     # Convert BGR to RGB (BGR to RGB for OpenCV)
372     frame_np = np.array(latest_frame)
373     frame_bgr = cv2.cvtColor(frame_np, cv2.COLOR_RGB2BGR)
374
375     # Process frame
376     result = human_fall_detector.process_frame(frame_bgr)
377
378     if 'error' not in result:
379         is_fall = result['is_fall']
380         conf = result['confidence']
381         angle = result['angle']
382
383         with image_lock:
384             fall_state_human['label'] = 'FALL' if is_fall else 'no fall'
385             fall_state_human['confidence'] = conf
386             fall_state_human['angle'] = angle
387             fall_state_human['ts'] = datetime.now().strftime("%H:%M:%S")
388
389         # Log fall detection result
390         prev = fall_state_human.get('last_status', 'no fall')
391         curr = 'FALL' if is_fall else 'no fall'
392         # Log only on transitions, or periodic high-confidence falls
393         if curr != prev or (is_fall and conf >= max(0.9, HUMAN_FALL_CONF_THRESH + 0.15)):
394             log_fall_detection(f"humanfall {curr}", conf, angle)

```

Figure 5.3.24: Real-time Fall Detection with OpenPifPaf

```

348 def extract_video_frames(video_path):
349     """Extract frames from video for fall detection processing"""
350     try:
351         import cv2
352         frames = []
353         cap = cv2.VideoCapture(video_path)
354
355         frame_count = 0
356         while cap.isOpened() and frame_count < 200: # Limit to 200 frames
357             ret, frame = cap.read()
358             if not ret:
359                 break
360
361             # Convert BGR to RGB
362             rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
363
364             # Resize frame to standard size
365             rgb_frame = cv2.resize(rgb_frame, (640, 480))
366
367             # Convert to base64 for transmission
368             _, buffer = cv2.imencode('.jpg', cv2.cvtColor(rgb_frame, cv2.COLOR_RGB2BGR))
369             frame_b64 = base64.b64encode(buffer).decode('utf-8')
370
371             frames.append({
372                 'frame_number': frame_count,
373                 'timestamp': frame_count / 30.0, # Assuming 30 FPS
374                 'image_data': frame_b64
375             })
376
377             frame_count += 1
378
379         cap.release()
380         return frames
381
382     except Exception as e:
383         log_error(f"Error extracting video frames: {e}")
384         return []
385

```

Figure 5.3.25: Video Processing Pipeline

For optimal real-time performance, the system uses a frame buffer to store the last three frames, and frames are processed at a regular interval to maintain efficiency without excessive delay.

```

298 # Optimized frame processing based on test results
299 window_size = 3
300 step_size = 1 # Process every frame for best accuracy (tested optimal)
301 max_frames = min(len(frames_data) - window_size + 1, 30) # Limit to 30 frame groups
302
303 for i in range(0, max_frames, step_size):
304     if i + 2 >= len(frames_data):
305         break
306
307     # Convert base64 frames to PIL Images
308     frame1_b64 = frames_data[i]['image_data']
309     frame2_b64 = frames_data[i + 1]['image_data']
310     frame3_b64 = frames_data[i + 2]['image_data']
311
312     # Decode and process with AI model
313     frame1_pil = Image.fromarray(frame1_rgb)
314     frame2_pil = Image.fromarray(frame2_rgb)
315     frame3_pil = Image.fromarray(frame3_rgb)
316
317     # Process with fall-detection-main
318     result = Fall_prediction(frame1_pil, frame2_pil, frame3_pil)
319     is_fall, label, conf, angle = _classify(result, CONF_THRESH)
320
321
322 # Ring buffer for 3-frame inference (PIL Images)
323 infer_buf = deque(maxlen=3)
324
325 # Desired inference FPS
326 INFER_FPS = float(os.getenv('FALL_INFER_FPS', '2.0'))
327 CONF_THRESH = float(os.getenv('FALL_CONF_THRESH', '0.7'))
328
329 # Shared fall state for overlay + API
330 fall_state = {
331     'label': 'no fall',
332     'confidence': 0.0,
333     'angle': 0.0,
334     'ts': None,
335     '_last_alert_ts': 0.0,
336     '_last_status': 'no fall'
337 }

```

Figure 5.3.26: Optimized Frame Processing

3. Hybrid Model Integration

The hybrid model integrates both the PoseNet and OpenPifPaf models, allowing the system to combine the strengths of both approaches. By merging the outputs from both models based on weighted confidence scores, the hybrid model provides a more reliable and accurate fall detection result. This integration enhances the system's robustness by compensating for the individual limitations of each model, leading to improved overall performance in detecting falls.

In the hybrid model, the results from both the PoseNet and OpenPifPaf models are fused. The system combines the outputs based on weighted confidence scores, ensuring that the final decision on whether a fall has occurred is more accurate. This fusion process helps the system leverage the strengths of each model, increasing detection accuracy and reducing false positives. The system allows for multiple modes including:

- Internal Mode: Uses the PoseNet-based model only.

- HumanFall Mode: Uses the OpenPifPaf-based model.
- Hybrid Mode: Uses both models.
- ROS2 Mode: External communication for fall detection.

Environment variables are used to configure the mode of operation and fine-tune the system's performance:

```

384 # Fall Detection Configuration
385 FALL_MODEL_MODE=hybrid      # internal, humanfall, hybrid, ros2
386 FALL_INFER_FPS=2.0          # Inference frequency
387 FALL_CONF_THRESH=0.7        # Confidence threshold
388 FALL_DECODE_STRIDE=4        # Frame sampling rate
389
390 # HumanFallDetection Settings
391 HUMAN_FALL_DISABLE_CUDA=true # Disable CUDA for compatibility
392 HUMAN_FALL_CONF_THRESH=0.7   # OpenPifPaf confidence threshold

```

Figure 5.3.27: Environment Variables

When a fall is detected, the system triggers a Telegram alert. This alert includes a snapshot of the fall event, along with the fall status, confidence score, and timestamp, allowing the system to notify relevant personnel or take further action. The Telegram alert ensures immediate notification, enabling timely responses to fall events.

```

390 # Send Telegram alert when fall is detected on live camera (with cooldown)
391 current_time = time.time()
392 if is_fall and conf >= CONF_THRESH and (current_time - fall_state.get('_last_alert_ts', 0)) > TELEGRAM_ALERT_COOLDOWN:
393     try:
394         photo_bytes = get_latest_jpeg()
395         alert_message = f"🚨 FALL DETECTED!\n\nConfidence: {conf:.2f}\nAngle: {angle:.1f}°\nTime: {datetime.now().strftime('%H:%M:%S')}"
396         telegram_send(alert_message, photo_bytes=photo_bytes)
397         fall_state['_last_alert_ts'] = current_time
398         log_app(f"Telegram fall alert sent - Confidence: {conf:.2f}")
399     except Exception as e:
400         log_error(f"Failed to send Telegram fall alert: {e}")

```

Figure 5.3.28: Telegram Alert Function

5.3.4 STT/TTS Model

This chapter describes the Speech-to-Text (STT) and Text-to-Speech (TTS) system developed for the project. The system enables real-time voice interaction, allowing the robot to listen to commands (STT) and respond with generated speech (TTS). It integrates VOSK (offline STT) and Whisper (online STT) engines, providing fallback capabilities for network issues. The system also features wake word detection, multi-language support, and AI-powered responses, seamlessly integrated with the robot's ROS2 framework.

Key Components:

Bachelor of Information Technology (Honours) Communications and Networking
Faculty of Information and Communication Technology (Kampar Campus), UTAR

CHAPTER 5

1. `stt_tts_api.py`: Provides REST API for STT/TTS functionality, including API key management, voice selection (e.g., Kore, Aoede, Charon), and language support (English, Chinese). It also handles audio conversion and logs interactions in an SQLite database.
2. `stt_tts_bridge_standalone.py`: A ROS2 node that manages wake word detection, audio monitoring, and STT engine selection (VOSK/Whisper). It integrates with Gemini AI for generating responses and publishes audio inputs and outputs to ROS2 topics.
3. `start_voice_bridge.py`: A script that sets up the environment and manages the initialization of the voice bridge, including model path configuration and process management.

To implement the STT models into the project, use the following installation commands:

Vosk:

```
Commands:
pip install vosk
pip install vosk-model-small-en-us-0.15
```

Whisper:

```
Commands:
pip install git+https://github.com/openai/whisper.git
pip install whisper[ffmpeg]
```

The system integrates with ROS2 to publish and subscribe to topics for real-time audio interaction.

```
### 5. ROS2 Integration

### **Topics**
- `/audio/mic` - Audio input from robot microphone
- `/audio/speaker` - Audio output to robot speakers

### **Services**
- `/voice/do_interaction` - Manual trigger for voice interaction
```

Figure 5.3.29: Ros2 Topic Integration

```
kokfu@FYP:~/turtlebot_webui/v3$ cd /home/kokfu/turtlebot_webui/v3 && ros2 topic
info /audio/speaker
Type: std_msgs/msg/UInt8MultiArray
Publisher count: 0
Subscription count: 1
kokfu@FYP:~/turtlebot_webui/v3$ cd /home/kokfu/turtlebot_webui/v3 && ros2 topic
info /audio/mic
Type: std_msgs/msg/UInt8MultiArray
Publisher count: 1
Subscription count: 0
```

Figure 5.3.30: Precheck on RoS2 related topic

To ensure the robustness of this system, hybrid approach is applied in case one of the model is unavailable.

```
def transcribe(wav16k_path: str, logger=None) -> str:
    global _vosk_available
    # Try VOSK first, fallback to Whisper
    if STT_ENGINE == "vosk" and _vosk_available:
        try:
            if logger:
                logger.info(f"[debug] STT engine: vosk (model_dir={VOSK_MODEL_DIR})")
            return vosk_transcribe(wav16k_path)
        except Exception as e:
            if logger:
                logger.warning(f"[debug] VOSK failed: (e), falling back to Whisper")
            _vosk_available = False

    # Fallback to Whisper
    if logger:
        logger.info(f"[debug] STT engine: whisper ((WHISPER_MODEL), (WHISPER_COMPUTE), lang={LANGUAGE_HINT})")
    return whisper_transcribe(wav16k_path)

def vosk_transcribe(wav16k_path: str) -> str:
    _init_vosk()
    if _vosk_model is None:
        raise RuntimeError("VOSK model not available")

    from vosk import KaldiRecognizer
    wf = wave.open(wav16k_path, "rb")
    rec = KaldiRecognizer(_vosk_model, wf.getframerate())
    text = ""
    while True:
        data = wf.readframes(4000)
        if not data:
            break
        if rec.AcceptWaveform(data):
            partial = json.loads(rec.Result()).get("text", "")
            if partial:
                text += (" " if text else "") + partial
    final = json.loads(rec.FinalResult()).get("text", "")
    if final:
        text += (" " if text else "") + final
    return text.strip()

def whisper_transcribe(wav16k_path: str) -> str:
    _init_whisper()
    segments, info = _whisper_model.transcribe(
        wav16k_path,
        language=LANGUAGE_HINT if LANGUAGE_HINT else None,
        beam_size=1,
        best_of=1,
        vad_filter=True,
        vad_parameters=dict(min_silence_duration_ms=700,
                           condition_on_previous_text=False)
    )
    out = []
    for seg in segments:
        out.append(seg.text.strip())
    return " ".join(out).strip()
```

Figure 5.3.31: Hybrid STT Model Implementation

To manage API quotas, the system uses rotating API keys for services like Gemini, allowing up to 75 requests/day with 5 keys.

```
#### **API Key Management**
'''python
# Rotating API keys for quota management
GENAI_API_KEYS = "key1,key2,key3,key4,key5" # 5 keys = 75 requests/day
'''

#### **API Key Rotation Algorithm**
'''python
def get_next_api_key():
    current_key = _api_keys[_current_key_index]
    _current_key_index = (_current_key_index + 1) % len(_api_keys)
    return current_key
'''
```

Figure 5.3.32: API Key Management & Rotation Algorithm

To track voice interactions, the system logs every interaction in an SQLite database. The database records details such as timestamp, language, STT engine used, input speech, AI response, and the TTS voice used.

```

def init_voice_db():
    """Initialize the voice logs database"""
    with sqlite3.connect(VOICE_LOG_DB) as con:
        con.execute("""
        CREATE TABLE IF NOT EXISTS voice_logs (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            ts REAL,
            lang TEXT,
            stt_engine TEXT,
            text_in TEXT,
            text_out TEXT,
            voice TEXT,
            source TEXT
        );
        """)
        # Prevent exact duplicates
        con.execute("""
        CREATE UNIQUE INDEX IF NOT EXISTS idx_voice_unique
        ON voice_logs(ts, text_in, text_out);
        """)

def insert_voice_log(entry):
    """Insert a voice log entry into the database"""
    try:
        with sqlite3.connect(VOICE_LOG_DB) as con:
            con.execute("""
            INSERT INTO voice_logs (ts, lang, stt_engine, text_in, text_out, voice, source)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """, (
                entry.get("ts"), entry.get("lang"), entry.get("stt_engine"),
                entry.get("text_in"), entry.get("text_out"),
                entry.get("voice"), entry.get("source")
            ))
    except sqlite3.IntegrityError:
        # Duplicate entry; ignore
        pass
    except Exception as e:
        print(f"Failed to store voice log: {e}")

```

Figure 5.3.33: Voice Logging Database

Figure 5.3.34 shows the main voice interaction pipeline for this project.

```

def _run_interaction(self, capture_sec: float):
    try:
        with tempfile.TemporaryDirectory() as td:
            in_wav = os.path.join(td, "in.wav")
            out24 = os.path.join(td, "reply_24k.wav")
            out16 = os.path.join(td, "reply_16k.wav")

            # capture
            self.get_logger().info(f"[run] Capturing last {capture_sec:.1f}s ...")
            self.dump_last_seconds_to_wav(capture_sec, in_wav)

            # STT
            self.get_logger().info(f"[run] Transcribing ...")
            wav16k = ensure_wav_16k_mono_16bit(in_wav)
            said = transcribe(wav16k, logger=self.get_logger())
            self.get_logger().info(f"[STT] {said}")

            if not said:
                return False, "No speech recognized."

            # Gemini
            self.get_logger().info(f"[run] Querying Gemini ...")
            start_time = time.time()
            reply_text = gemini_chat(said, logger=self.get_logger())
            response_time = time.time() - start_time
            self.get_logger().info(f"[AI] {reply_text}")

            # Play
            self.get_logger().info(f"[run] Playing to /audio/speaker ...")
            self._speaking_now = True
            try:
                self.publish_pcm16(frames, realtime=True)
                # Calculate audio duration
                audio_duration = len(frames) / (SAMPLE_RATE * SAMPLE_BYTES * CHANNELS)
                self.get_logger().info(f"[run] Audio output completed ({audio_duration:.2f}s)")
            finally:
                self._speaking_now = False

            return True, said
    except Exception as e:
        self.get_logger().error(f"[run] Interaction failed: {e}")
        return False, str(e)

```

Figure 5.3.34: Main Voice Interaction Pipeline

The system continuously monitors audio for specific wake words, such as "Hey", "Ellie", and "Hey Ellie". Once detected, the system activates for interaction.


```

def _wake_loop(self):
    while rcipy.ok() and not self._stop:
        time.sleep(WAKE_POLL_SEC)

        # Don't trigger while speaking
        if self._speaking_now:
            continue

        # Cool down
        if (time.time() - self._last_trigger_ts) < COOLDOWN_SEC:
            continue

        # Need enough audio to check
        if len(self.ring) < int(SAMPLE_RATE * WAKE_LISTEN_SEC * SAMPLE_BYTES * CHANNELS):
            continue

        try:
            with tempfile.TemporaryDirectory() as td:
                probe_wav = os.path.join(td, "wake_probe.wav")
                self.dump_last_seconds_to_wav(WAKE_LISTEN_SEC, probe_wav)
                wav16k = ensure_wav_16k_mono_16bit(probe_wav)

                # Quick transcription just for wake detection
                txt = transcribe(wav16k, logger=self.get_logger()).lower()
                if not txt:
                    continue

                self.get_logger().info(f"[wake] heard: {txt}")

                if not REQUIRE_WAKE_WORD:
                    # Treat any speech as trigger
                    self._trigger_async()
                    continue

                # Wake-word check
                if any(w in txt for w in WAKE_WORDS):
                    self.get_logger().info(f"[wake] Wake word detected + triggering interaction.")
                    self._trigger_async()
        except Exception as e:
            self.get_logger().error(f"[wake] check failed: {e}")

```

Figure 5.3.35: Wake Words Detection Algorithm

5.3.5 Web Interface

A web-based User Interface (UI) was developed to provide an accessible and intuitive platform for real-time robot monitoring and control. The Flask framework was chosen for the backend due to its lightweight nature and seamless integration with Python-based ROS2 libraries. The primary communication bridge between the web browser and the ROS2 ecosystem is the ROSBridge WebSocket server.

Command:

```
Sudo apt install ros-humble-rosbridge-server
```

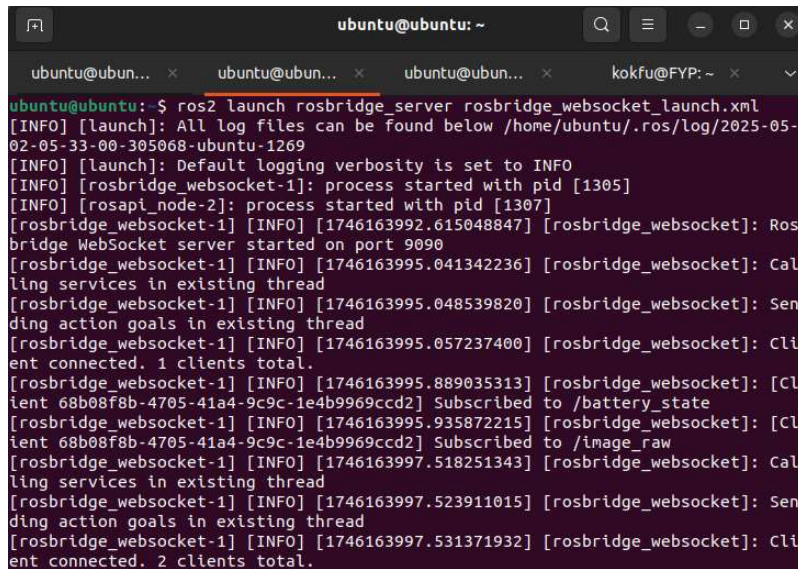
The UI's architecture consists of three core components that work in tandem:

1. **Flask Web Application (Remote PC):** This Python application serves as the web server. Its main responsibilities are to host the HTML, CSS, and JavaScript files that constitute the user interface and to manage the application's routing logic (e.g., directing users to the home, control, or telemetry pages).
2. **ROSBridge Server (TurtleBot3):** This is the crucial middleware that translates communication protocols. It runs on the TurtleBot3 and exposes ROS2 topics, services, and actions over a WebSocket connection. This allows standard web technologies to interact directly with the robot's ROS2 nodes. The server is launched on the robot with the command:

Command

```
ros2 launch rosbridge_server rosbridge_websocket_launch.xml
```

3. Web Browser (Client): The frontend of the application runs in any modern web browser. It uses HTML for structure, CSS for styling, and JavaScript—specifically the `roslib.js` library—to establish a WebSocket connection with the ROSBridge server. This direct connection enables the browser to publish control commands (e.g., to the `/cmd_vel` topic) and subscribe to data streams (e.g., from `/battery_state`) in real-time.



```
ubuntu@ubuntu: ~
ubuntu@ubun... x ubuntu@ubun... x ubuntu@ubun... x kokfu@FYP: ~ x v
ubuntu@ubuntu: $ ros2 launch rosbridge_server rosbridge_websocket_launch.xml
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2025-05-02-05-33-00-305068-ubuntu-1269
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [rosbridge_websocket-1]: process started with pid [1305]
[INFO] [rosapi_node-2]: process started with pid [1307]
[rosbridge_websocket-1] [INFO] [1746163992.615048847] [rosbridge_websocket]: Ros
bridge WebSocket server started on port 9090
[rosbridge_websocket-1] [INFO] [1746163995.041342236] [rosbridge_websocket]: Cal
ling services in existing thread
[rosbridge_websocket-1] [INFO] [1746163995.048539820] [rosbridge_websocket]: Sen
ding action goals in existing thread
[rosbridge_websocket-1] [INFO] [1746163995.057237400] [rosbridge_websocket]: Cli
ent connected. 1 clients total.
[rosbridge_websocket-1] [INFO] [1746163995.889035313] [rosbridge_websocket]: [Cl
ient 68b08f8b-4705-41a4-9c9c-1e4b9969ccd2] Subscribed to /battery_state
[rosbridge_websocket-1] [INFO] [1746163995.935872215] [rosbridge_websocket]: [Cl
ient 68b08f8b-4705-41a4-9c9c-1e4b9969ccd2] Subscribed to /image_raw
[rosbridge_websocket-1] [INFO] [1746163997.518251343] [rosbridge_websocket]: Cal
ling services in existing thread
[rosbridge_websocket-1] [INFO] [1746163997.523911015] [rosbridge_websocket]: Sen
ding action goals in existing thread
[rosbridge_websocket-1] [INFO] [1746163997.531371932] [rosbridge_websocket]: Cli
ent connected. 2 clients total.
```

Figure 5.3.36: Example of bringup WebSocket of Turtlebot

To begin, create a new project directory using Visual Studio Code. Within this directory, create subdirectories as needed (e.g., templates to store HTML files and static for JavaScript and CSS files). Instantiate the Flask application in a Python file (e.g., `app.py`) and set it up to host the necessary routes and handle incoming requests. On the frontend (JavaScript/HTML), establish a connection to the ROSBridge server through `roslibjs` or `roslibpy`.

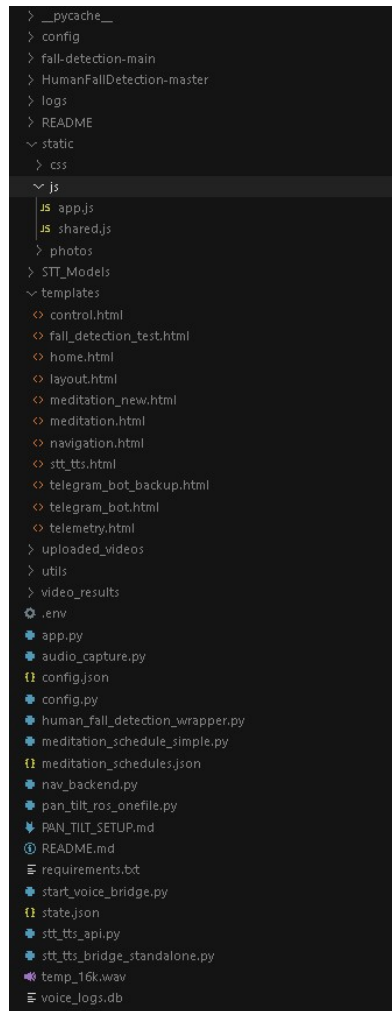


Figure 5.3.37: Sample File Hierarchy

Example Web UI integration of ROS2 to publish and subscribe to topics for real-time audio interaction.

```

### 6. Web UI Integration

#### **API Endpoints**
- `POST /api/stt-tts/start-stt` - Manual STT trigger
- `POST /api/stt-tts/tts` - Text-to-speech conversion
- `POST /api/stt-tts/test-voice` - Voice testing
- `POST /api/stt-tts/direct-tts` - Direct ITS (for replay)
- `GET /api/stt-tts/logs` - Voice interaction history
- `GET /api/stt-tts/voices` - Available voices

```

Figure 5.3.38: WebUI Integration

```

597 @app.route('/')
598 def index():
599     return render_template('home.html')
600
601 @app.route('/telegram-bot')
602 def telegram_bot_page():
603     """Telegram bot management page"""
604     return render_template('telegram_bot.html',
605                           TELEGRAM_BOT_TOKEN=TELEGRAM_BOT_TOKEN,
606                           TELEGRAM_CHAT_ID=TELEGRAM_CHAT_ID)
607
608 @app.route('/control')
609 def control():
610     return render_template('control.html')
611
612 @app.route('/telemetry')
613 def telemetry():
614     return render_template('telemetry.html')
615
616
617 @app.route('/nav')
618 def nav():
619     import os
620     return render_template(
621         'navigation.html',
622         model=os.environ.get('TURTLEBOT3_MODEL', 'unset'),
623         map_yaml=os.environ.get('NAV_MAP_YAML', os.path.expanduser('~/.map.yaml')),
624     )
625
626
627 @app.route('/stt-tts')
628 def stt_tts():
629     return render_template('stt_tts.html')
630
631 @app.route('/fall-detection-test')
632 def fall_detection_test():
633     return render_template('Fall_detection_test.html')
634
635
636 @app.route('/api/telemetry')
637 def get_telemetry():
638     return jsonify(telemetry_data)

```

Figure 5.4.39: Sample Route to each pages

This code also employs the `roslibpy` library within the Flask backend to subscribe to ROS2 topics, enabling the application to receive real-time information from the robot. Similarly, to publish control commands to the TurtleBot3, define publishers in the Flask backend. The following figures illustrate examples of each page created in the Web UI through Flask implementation, which establishes the connection between the robot and the UI.

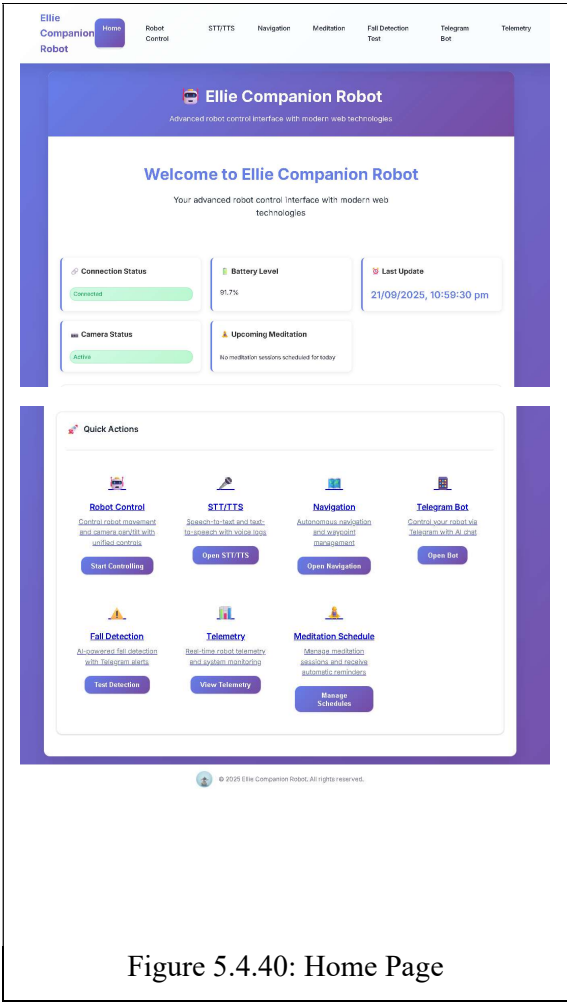


Figure 5.4.40: Home Page

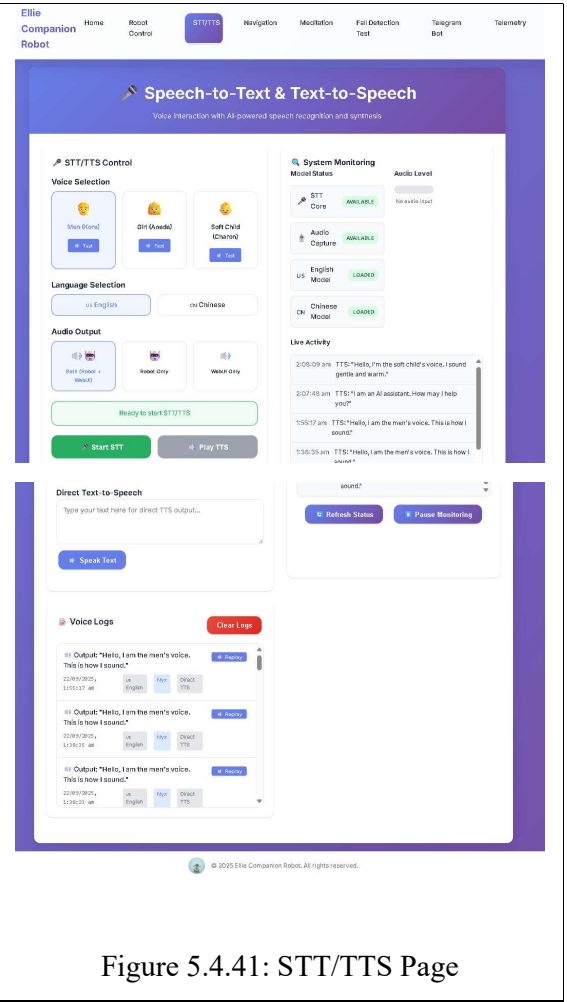


Figure 5.4.41: STT/TTS Page

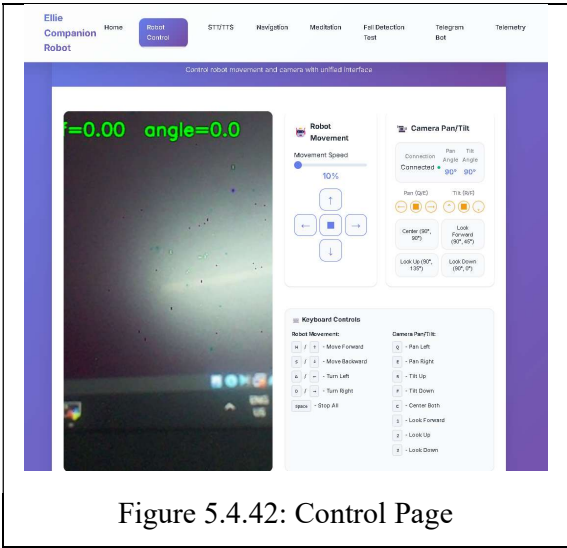


Figure 5.4.42: Control Page

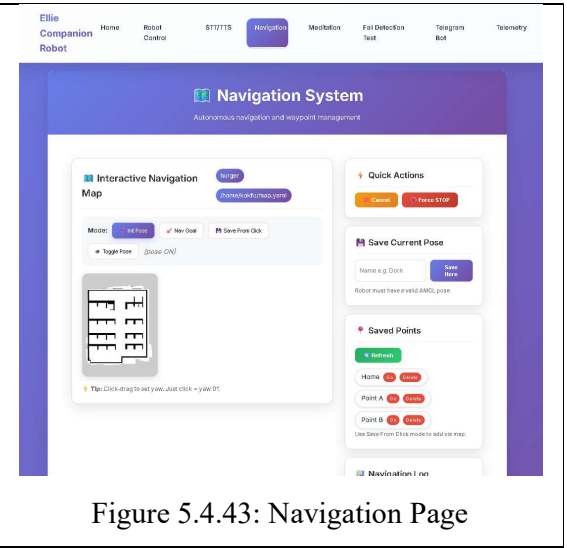


Figure 5.4.43: Navigation Page

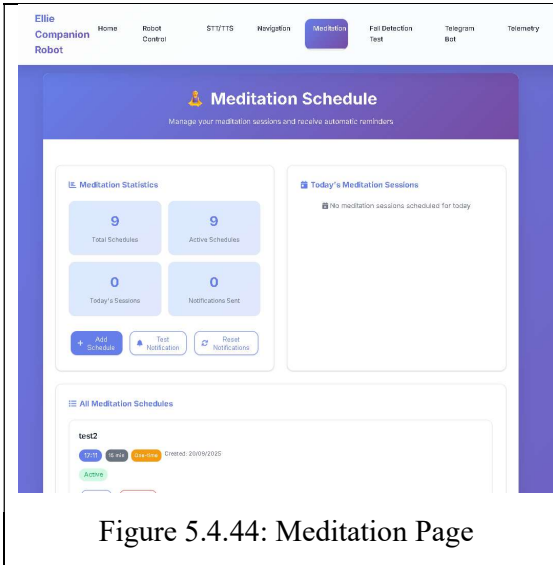


Figure 5.4.44: Meditation Page

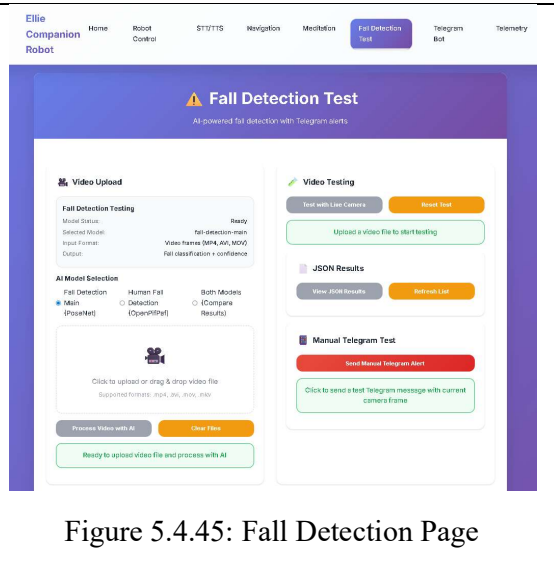


Figure 5.4.45: Fall Detection Page

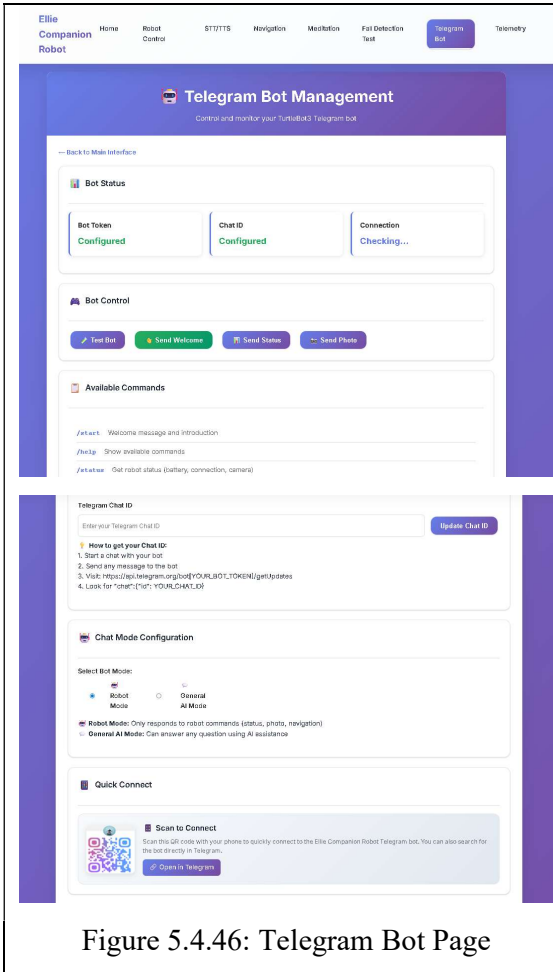


Figure 5.4.46: Telegram Bot Page

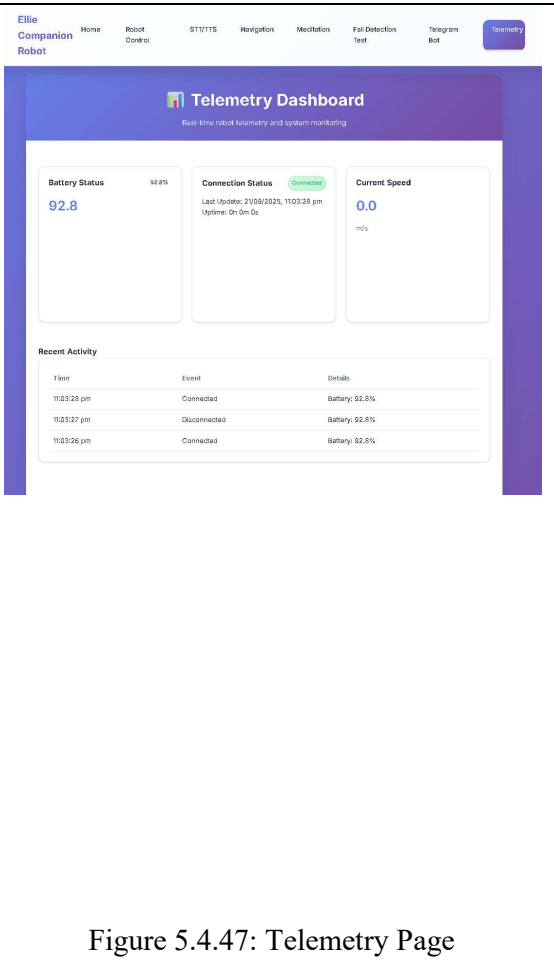


Figure 5.4.47: Telemetry Page

5.3.6 Telegram Bot

The Telegram Bot module provides an alternative remote interface for monitoring and controlling the TurtleBot3. Through Telegram, users can issue commands, receive real-time status updates, interact with the robot via AI chat, and obtain media such as camera snapshots. This system is designed to complement the Web UI and ROS2 integration, providing a lightweight and accessible interface that can be accessed on any device supporting Telegram.

The core function for sending messages is `telegram_bot_send_message()`. This function allows the bot to send both text and images to a specified chat. If no chat ID is provided, the function defaults to a pre-configured Telegram chat. The implementation supports sending JPEG images captured from the robot alongside captions. Proper error handling ensures that any failures in message delivery are logged for debugging.

```

906 # ===== telegram_bot_functions =====
907 def telegram_bot_send_message(text: str, chat_id: str = None, photo_bytes: bytes = None) -> bool:
908     """Send a message (and optional photo) to Telegram chat"""
909     if not TELEGRAM_BOT_TOKEN:
910         return False
911     target_chat_id = chat_id or TELEGRAM_CHAT_ID
912     if not target_chat_id:
913         return False
914
915     try:
916         if photo_bytes:
917             url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendPhoto"
918             files = {'photo': ('photo.jpg', photo_bytes, 'image/jpeg')}
919             data = {'chat_id': target_chat_id, 'caption': text}
920         else:
921             url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage"
922             data = {'chat_id': target_chat_id, 'text': text}
923
924             response = requests.post(url, data=data, files=files if photo_bytes else None, timeout=10)
925             return response.ok
926     except Exception as e:
927         log_error(f"Telegram bot send errors: {e}")
928         return False

```

Figure 5.4.48: Telegram Bot Message Function

The `get_robot_status_for_telegram()` function formats the robot's telemetry data for display in Telegram including battery level, connection status, camera status and last update timestamp. The formatted message provides both a concise summary and a visual representation of the robot's online/offline status using emojis. This enables quick assessment of the robot's state remotely.

```

931 def get_robot_status_for_telegram() -> str:
932     """Get Formatted robot status for Telegram"""
933     try:
934         battery = telemetry_data.get('battery_level', 0.0)
935         connected = telemetry_data.get('connected', False)
936         camera_status = telemetry_data.get('camera_status', 'Unknown')
937         last_update = telemetry_data.get('last_update', 'Never')
938
939         status_text = f""" 🤖 **Robot Status Report**
940
941         **Battery**: {battery:.1f}%
942         **Connection**: { '🟢 Connected' if connected else '🔴 Disconnected' }
943         **Camera**: {camera_status}
944         **Last Update**: {last_update}
945         **Report Time**: {datetime.now().strftime('%H:%M:%S')}
946
947         **System Status**: { '🟢 Online' if connected else '🔴 Offline' } """
948         return status_text
949     except Exception as e:
950         return f"🔴 **Error retrieving status**: {str(e)}"

```

Figure 5.4.49: Telegram Status Function

The function `handle_telegram_command()` processes commands sent by the user. Supported commands include:

- `/start`: Provides a welcome message and lists available commands.
- `/help`: Lists all commands and usage tips.
- `/status`: Returns the formatted robot status.
- `/photo`: Captures and sends a snapshot from the robot's camera.
- `/chat`: Initiates an AI-powered chat session.
- `/test`: Tests bot connectivity and returns a status confirmation.

User authorization is enforced through the `TELEGRAM_ALLOWED_USERS` configuration, ensuring only authorized users can execute commands. Unknown commands return a help message, guiding the user to valid commands.

```

954 def handle_telegram_command(command: str, chat_id: str, user_id: str = None) -> str:
955     """Handle incoming Telegram commands"""
956     command = command.lower().strip()
957
958     # Check if user is authorized (if TELEGRAM_ALLOWED_USERS is configured)
959     if TELEGRAM_ALLOWED_USERS and user_id and str(user_id) not in TELEGRAM_ALLOWED_USERS:
960         return "❌ You are not authorized to use this bot."
961
962     if command == '/start':
963         return "" 🤖 **Welcome to TurtleBot3 Telegram Bot!**
964
965     I'm your robot's remote interface. Here's what I can do:
966
967     📋 **Available Commands:**
968     /help - Show this help message
969     /status - Get robot status
970     /photo - Take a photo
971     /chat - Start interactive chat with AI
972     /test - Test bot connectivity
973
974     🔥 **Ready to help!** Send me a command to get started."""
975
976     elif command == '/help':
977         return "" 📋 **Available Commands:**
978
979     /start - Welcome message
980     /help - Show this help
981     /status - Get robot status (battery, connection, camera)
982     /photo - Capture and send current photo
983     /chat - Start interactive chat with AI
984     /test - Test bot connectivity
985
986     ⚡ **Tip**: Commands are case-insensitive!
987     🗨️ **Chat**: Send any text message to chat with the AI assistant."""
988
989     elif command == '/status':
990         return get_robot_status_for_telegram()
991
992     elif command == '/photo':
993         try:
994             photo_bytes = get_latest_jpeg()
995             if photo_bytes:
996                 caption = f"📷 **Robot Camera Snapshot**\n🕒 {datetime.now().strftime('%H:%M:%S')}"
997                 telegram_bot_send_message(caption, chat_id, photo_bytes)
998                 return "" 📷 Photo sent!
999             else:
1000                 return "❌ No camera image available. Camera may be offline or not initialized."
1001         except Exception as e:
1002             return f"❌ Error capturing photo: {str(e)}"
1003
1004     elif command == '/test':
1005         return f"" 🟢 **Bot Test Successful!**
1006
1007     🤖 Bot is working correctly
1008     ⌚ Test time: {datetime.now().strftime('%H:%M:%S')}
1009     📶 Connection: Active
1010     🔥 Ready for commands!""
1011
1012     elif command == '/chat':
1013         return "" 🗨️ **Interactive Chat Mode**
1014
1015     You can now send me any text message and I'll respond using AI!
1016
1017     Just type your message (no need for commands) and I'll chat with you using Gemini AI.
1018
1019     Type /help to see other commands."""
1020
1021     else:
1022         return f"❓ Unknown command: {command}\n\nType /help to see available commands."

```

Figure 5.4.50: Telegram Command Handling Flow

The function `handle_telegram_chat()` provides an interactive chat feature using the Gemini AI integration. Incoming messages are combined with the robot's current telemetry data to give context-aware responses. The system verifies user authorization before responding. If the STT/TTS system is unavailable, appropriate error messages are returned, maintaining robustness.

```
def handle_telegram_chat(message_text, chat_id, user_id, user_name):
    """Handle interactive chat with Gemini AI"""
    # Check if user is authorized
    if TELEGRAM_ALLOWED_USERS and user_id and str(user_id) not in TELEGRAM_ALLOWED_USERS:
        return "❌ You are not authorized to use this bot."

    # Log
    print(f"User: {user_name} (ID: {user_id}) sent message: {message_text}")

    # Import the STT/TTS core for Gemini integration
    from stt_tts_core import stt_tts_core

    # Add robot context to the message
    robot_context = f"""You are an AI assistant for a TurtleBot3 robot. The user is chatting with you via Telegram.

    Current robot status:
    - Battery: {telemetry_data.get('battery_level', 0.0)}%
    - Connected: {telemetry_data.get('connected', False)}
    - Camera: {telemetry_data.get('camera_status', 'unknown')}

    User message: {message_text}

    Please respond in a helpful, friendly way. You can mention the robot's capabilities like taking photos, checking status, or navigation if relevant to the conversation."""

    # Get AI response using Gemini
    ai_response = stt_tts_core.gemini_chat(robot_context)

    # Add a small indicator that this is from the robot
    return f"🤖 {ai_response}"

except ImportError:
    return "❌ AI chat feature is not available. Please check the STT/TTS system configuration."
except Exception as e:
    log_error(f"Error in Telegram chat: {e}")
    return f"❌ Sorry, I encountered an error: {str(e)}"
```

Figure 5.4.51: Telegram AI Chat Interaction

The `telegram_polling_worker()` is a background process that continuously polls the Telegram API for new messages. Key features include:

- Polling updates with `getUpdates` and maintaining the last processed update ID.
- Differentiating between commands (starting with `/`) and regular chat messages.
- Invoking appropriate handlers (`handle_telegram_command()` or `handle_telegram_chat()`).
- Sending responses back to the corresponding Telegram chat.
- Error handling with retries in case of network or API failures.

This ensures that the bot can process messages in real-time, providing responsive control and interaction for the user.


```

059 def telegram_polling_worker():
060     """Background worker to poll Telegram for new messages"""
061     global telegram_last_update_id, telegram_polling_enabled
062     if not TELEGRAM_BOT_TOKEN:
063         log_app("Telegram bot not configured, skipping polling")
064         return
065
066     log_app("Starting Telegram bot polling worker")
067
068     while telegram_polling_enabled:
069         try:
070             # Get updates from Telegram
071             url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/getUpdates"
072             params = {
073                 'offset': telegram_last_update_id + 1,
074                 'timeout': 30,
075                 'limit': 10
076             }
077
078             response = requests.get(url, params=params, timeout=35)
079
080             if response.ok:
081                 data = response.json()
082                 if data.get('ok'):
083                     updates = data.get('result', [])
084
085                     for update in updates:
086                         telegram_last_update_id = update['update_id']
087
088                         if 'message' in update:
089                             message = update['message']
090                             chat_id = str(message.get('chat', {}).get('id', ''))
091                             user_id = str(message.get('from', {}).get('id', ''))
092                             text = message.get('text', '').strip()
093
094                             if not text:
095                                 continue
096
097                             log_app(f"Telegram message from (user_id): {text}")
098
099                             # Handle commands (messages starting with /)
100                             if text.startswith('/'):
101                                 command = text.split()[0][1:] # Remove '/' and get command
102                                 response_text = handle_telegram_command(command, chat_id, user_id)
103
104                             else:
105                                 # Handle regular text messages as chat
106                                 response_text = handle_telegram_chat(text, chat_id, user_id)
107
108                             # Send response back to Telegram
109                             if response_text:
110                                 telegram_bot_send_message(response_text, chat_id)
111
112                             else:
113                                 log_error(f"Telegram API errors: {data.get('description', 'Unknown error')}")
114
115                             else:
116                                 log_error(f"Telegram polling HTTP error: {response.status_code}")
117
118             except Exception as e:
119                 log_error(f"Telegram polling error: {e}")
120                 time.sleep(5) # Wait before retrying on error
121
122             time.sleep(1) # Check for messages every second

```

Figure 5.4.52: Telegram Polling Worker Function

5.4 System Operation (with Screenshot)

To start the project, the command “./robot_start.sh” is executed to initialize all necessary functions, including the camera, servos, speaker, microphone, robot bringup, and WebSocket server. Similarly, the command “./remote_start.sh” is executed on the remote PC to start the Web UI and related functions.

```
[INFO] [Launch]: All log files can be found below /home/ubuntu/.ros/log/2025-09-22-04-04-33-007143-ubuntu-1158
[INFO] [Launch]: Default logging verbosity is set to INFO
[INFO] [Launch]: urdf file name : urdf5_robot3_burger.urdf
[INFO] [robot_state_publisher-3]: process started with pid [11511]
[INFO] [hls_laser_publisher-2]: process started with pid [23152]
[INFO] [turtletbot3_ros-3]: process started with pid [11513]
[INFO] [hls_laser_publisher-2] [INFO] [1758485078.000000000] [hls_laser_publisher]: Init hls_laser_publisher Node Main
[INFO] [hls_laser_publisher-2] [INFO] [1758485078.077753000] [hls_laser_publisher]: port : dev/ttyUSB0 frame_id : base_scan
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.102269000] [turtletbot3_node]: Init turtletbot3 Node Main
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.221308000] [turtletbot3_node]: Init DynamixelSOMapper
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.204478100] [DynamixelSOMapper]: Successfully to open the port/dev/ttyACM0!
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.262239000] [DynamixelSOMapper]: Successfully to change the baudrate
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.417555100] [turtletbot3_node]: Start Calibration of gyro
[INFO] [robot_state_publisher-3] [INFO] [1758485078.712052500] [robot_state_publisher]: got segment base_footprint
[INFO] [robot_state_publisher-3] [INFO] [1758485078.783191500] [robot_state_publisher]: got segment base_scan
[INFO] [robot_state_publisher-3] [INFO] [1758485078.783209870] [robot_state_publisher]: got segment base_link
[INFO] [robot_state_publisher-3] [INFO] [1758485078.783408820] [robot_state_publisher]: got segment caster_back_link
[INFO] [robot_state_publisher-3] [INFO] [1758485078.784333820] [robot_state_publisher]: got segment imu_link
[INFO] [robot_state_publisher-3] [INFO] [1758485078.784583860] [robot_state_publisher]: got segment wheel_left_link
[INFO] [robot_state_publisher-3] [INFO] [1758485078.784583860] [robot_state_publisher]: got segment wheel_right_link
[INFO] [turtletbot3_ros-3] [INFO] [1758485078.828167000] [turtletbot3_node]: Calibration end
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.428112950] [turtletbot3_node]: Add Motors
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.485307440] [turtletbot3_node]: Add Wheels
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.512274780] [turtletbot3_node]: Added Sensors
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.728408660] [turtletbot3_node]: Succeeded to create battery state publisher
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.876988910] [turtletbot3_node]: Succeeded to create imu publisher
[INFO] [turtletbot3_ros-3] [INFO] [1758485083.963717000] [turtletbot3_node]: Succeeded to create sensor state publisher
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.034431570] [turtletbot3_node]: Succeeded to create joint state publisher
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.048643270] [turtletbot3_node]: Added Device
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.084203000] [turtletbot3_node]: Succeeded to create motor power server
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.106278310] [turtletbot3_node]: Succeeded to create reset server
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.135842620] [turtletbot3_node]: Succeeded to create sound server
[INFO] [turtletbot3_ros-3] [INFO] [1758485084.159657610] [turtletbot3_node]: Run!
[INFO] [turtletbot3_ros-3] [INFO] [1758485085.123603110] [diff_drive_controller]: Init Odometry
[INFO] [turtletbot3_ros-3] [INFO] [1758485085.519053640] [diff_drive_controller]: Run!
```

Figure 5.4.1: Launching Turtlebot3 with `./robot start.sh`

[illegible]

Figure 5.4.2: Launching RemotePC with `./remote start.sh`

The Web UI can then be accessed directly through the link 192.168.1.102:5000, the remote PC preconfigured static IP, which redirects the user to the home page, as shown in Figure 5.4.3.

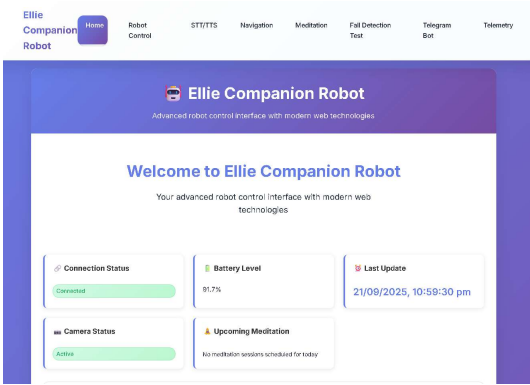


Figure 5.4.3: Home Page of WebUI

From the home page, users can navigate to the control page to operate the robot and monitor the camera feed. The camera preview displays current confidence and angle values. Users can also control the servos and robot movement using preset keyboard commands. The robot continuously processes the live camera feed for fall detection.

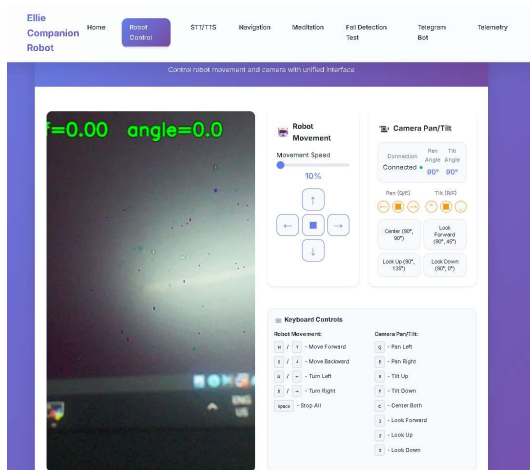


Figure 5.4.4: Home Page of WebUI

When a fall is detected, an alert is automatically sent to the caregiver via Telegram, enabling timely intervention.

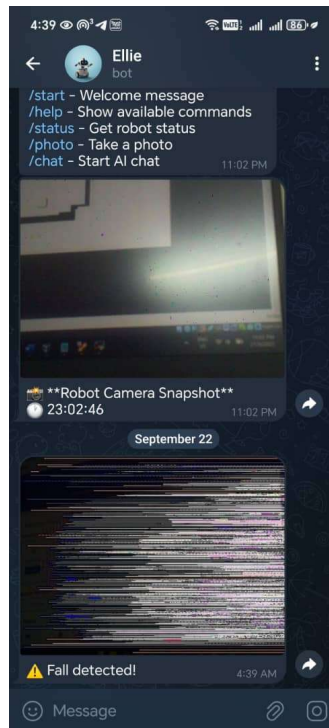


Figure 5.4.5: Alert Message Received in Telegram

On the navigation page, the initial pose of the robot is displayed based on data from the map.yaml file when the map node is run. If the initial pose is incorrect, users can reset it by selecting the appropriate mode and clicking the correct coordinates on the map. Users can also save new poses as points of interest and navigate to any saved point as desired.

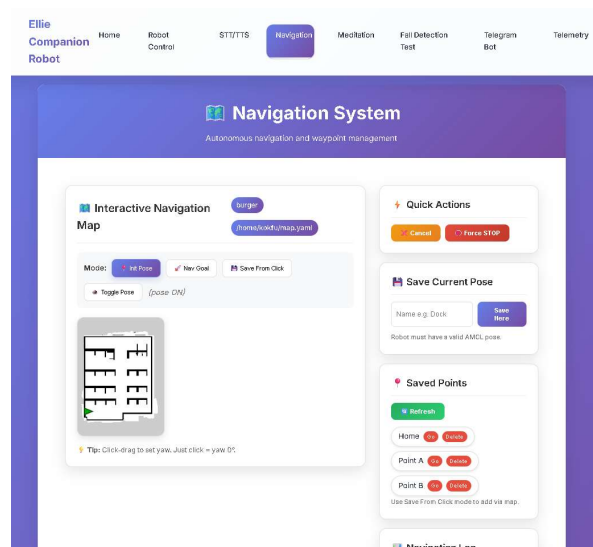


Figure 5.4.6: Navigation Page of WebUI

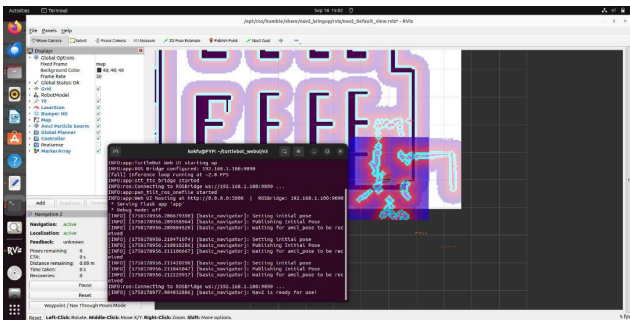
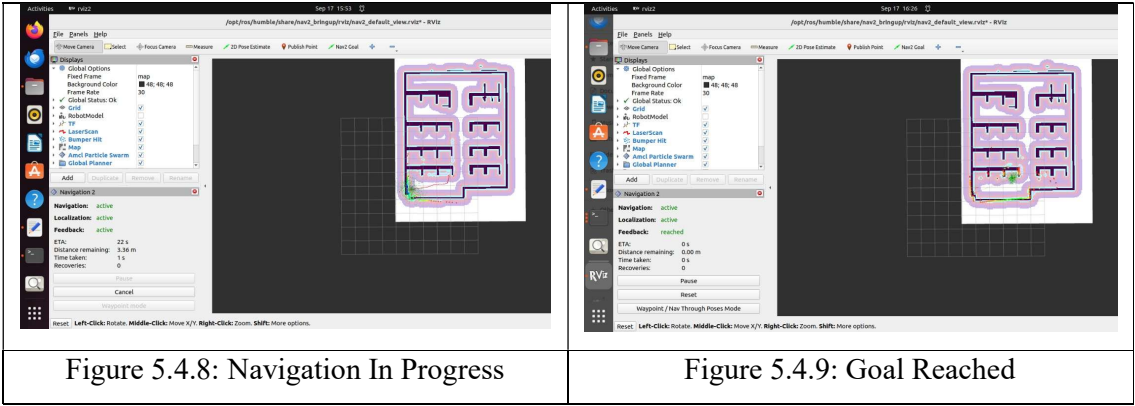


Figure 5.4.7: Rviz after startup of map

Once a user clicks a navigation point, the robot’s current status can be monitored in Rviz, as shown in Figure 5.4.8 and Figure 5.4.9.



The robot also supports voice interaction. Users can activate the system with wake words such as “Hey” or “Hey Eddie”. Speech is converted to text, sent to Gemini AI for processing, and the reply is converted back to speech for output via the robot’s speaker. Users can select between male, female, or child voices for variety.

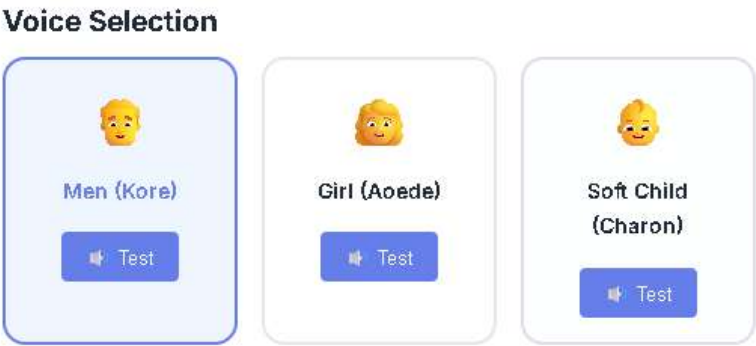


Figure 5.4.10: Selection of Voice

[illegible]

Figure 5.4.11: Example of AI Conversation

Users can manually input text on the STT/TTS page, which the robot converts to speech. This feature allows users to communicate with the elderly even when not physically present. Both English and Chinese inputs are supported.

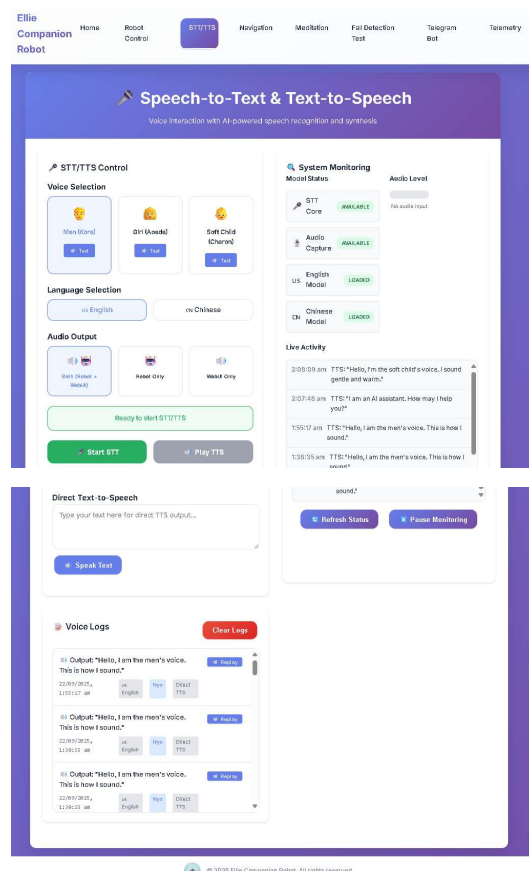


Figure 5.4.12: STT/TTS Page

CHAPTER 5

The fall detection test page allows users to upload a video and select a model to evaluate fall detection performance. Results are stored in a JSON file, which can be downloaded by the user.

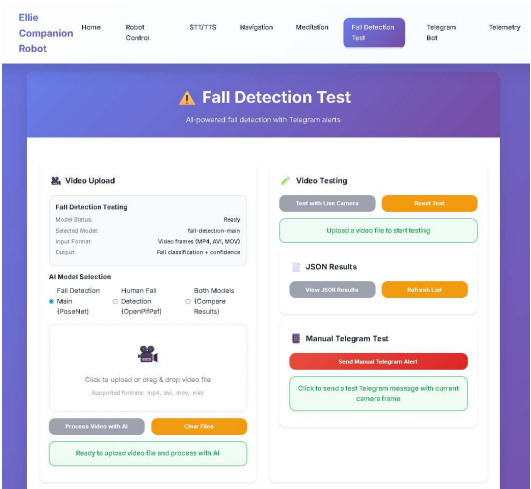


Figure 5.4.13: Fall Detection Test

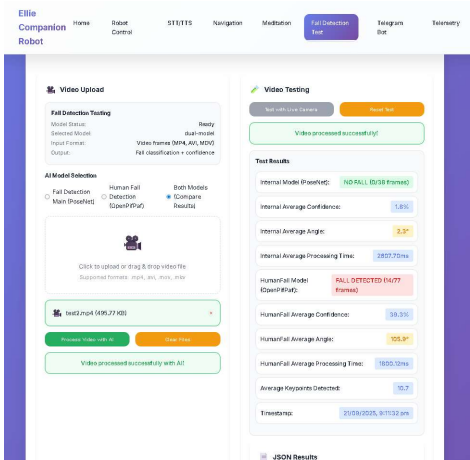


Figure 5.4.14: Example of Fall Detection Test Result

The telemetry dashboard provides real-time monitoring of the robot’s battery status, connection status, and current speed from the Web UI.

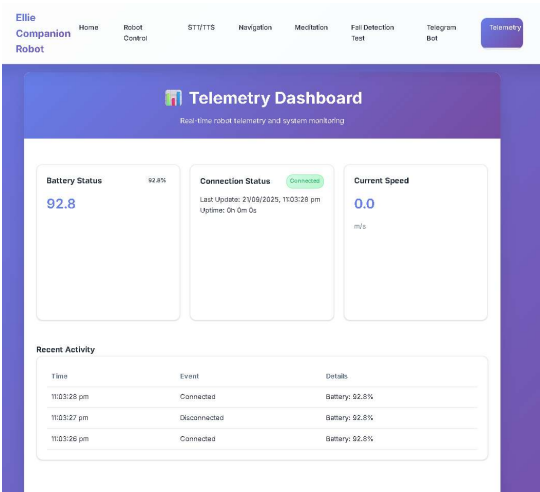


Figure 5.4.15: Telemetry Dashboard

The meditation page enables users to configure meditation schedules for the elderly, particularly beneficial for those with Alzheimer’s disease who may forget their routine. Users can add new schedules and view upcoming sessions for the day. Telegram reminders are automatically sent to the user based on the schedule.

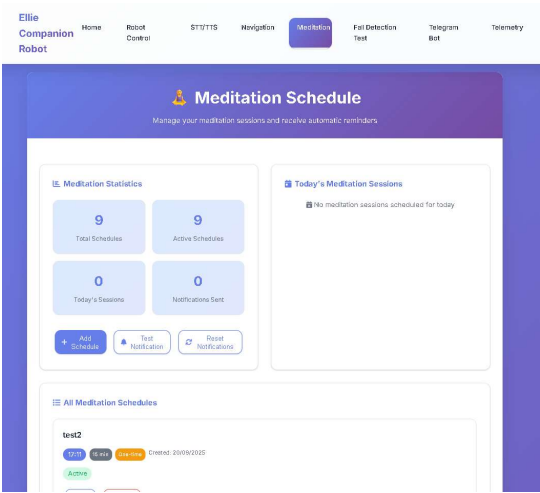


Figure 5.4.16: Meditation Schedule



Figure 5.4.17: Meditation Schedule Reminder from Telegram

CHAPTER 5

The Telegram Bot management page allows users to configure the chat ID to designate the Telegram account for receiving reminders. Navigation links and a QR code for quick access are also provide.

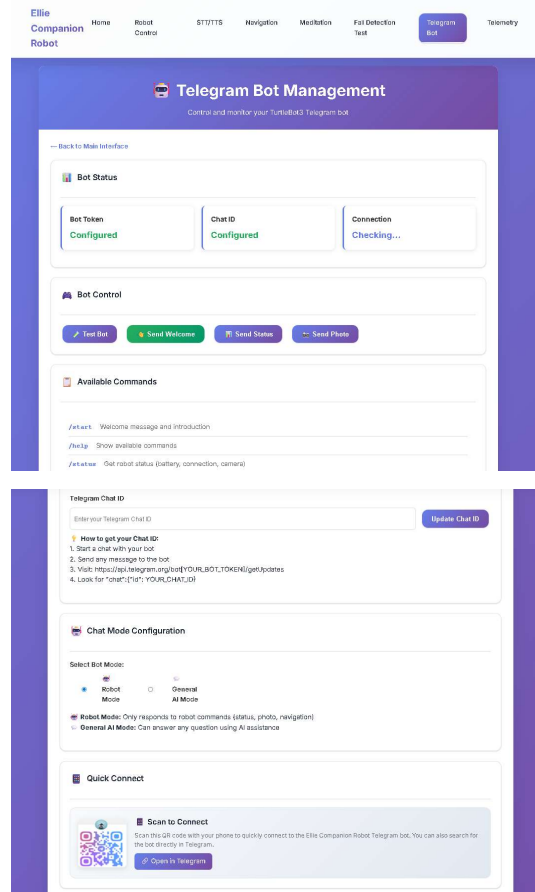
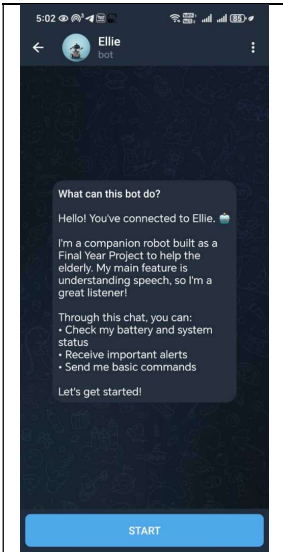
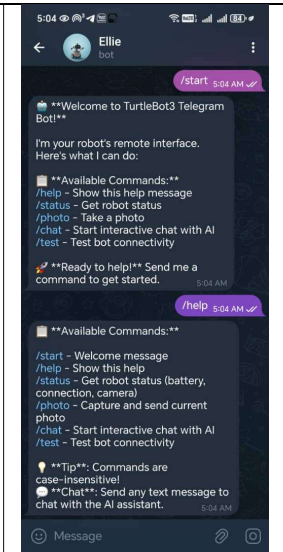
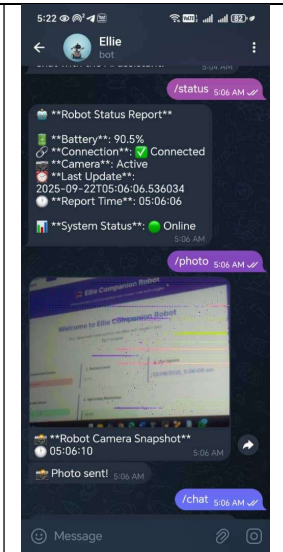
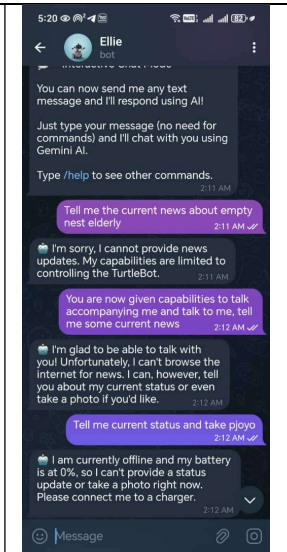


Figure 5.4.18: Telegram Bot Management Page

Clicking the link redirects the user to the Telegram Bot interface. Examples of bot interactions are shown in Figure 5.4.19 to Figure 5.4.22, including startup, command list, status check, and interactive chat.

			
Figure 5.4.19: Telegram Bot Start	Figure 5.4.20: Telegram Bot Commands	Figure 5.4.21: Telegram Bot /status command	Figure 5.4.22: Telegram Bot Interactive Chat

5.5 Implementation Issues and Challenges

Implementing a mobile companion robot that integrates autonomous navigation, real-time mapping, voice interaction, a web interface, and a Telegram bot presents significant challenges. A primary hurdle is the integration of diverse hardware components such as sensors, actuators, and computing units, each with distinct communication protocols and power requirements. Ensuring seamless interoperability among these components necessitates meticulous planning and often custom interfacing solutions. On the software side, harmonizing various libraries and frameworks can lead to compatibility issues requiring extensive debugging. During the project, these high-level complexities manifested as several specific technical and hardware issues that required innovative solutions.

A key performance issue was real-time video processing on the Raspberry Pi. The initial video stream was choppy and suffered from significant lag due to the limited processing power of the Raspberry Pi, making it unsuitable for applications such as fall detection. Additionally, the camera’s resolution and frame rate were insufficient for precise angle measurements, and the TurtleBot3’s small size reduced the effective field of view for accurate detection. To mitigate these problems, the data transmission was optimized by enforcing a fixed frame rate of 30 FPS in the camera_params.yaml file and switching from the raw image topic

to the `/camera/image_raw/compressed` topic. These adjustments reduced network bandwidth and processing load, resulting in a smoother, real-time video feed.

In the domain of fall detection, a single camera-based model yielded inconsistent accuracy, particularly due to the subject's distance from the camera and the limited angle coverage of the TurtleBot3. To improve reliability, a dual-model approach was implemented: one algorithm analyzed posture using pose estimation, while another assessed motion across three consecutive frames. Running both models in parallel and cross-referencing their outputs produced a more robust and accurate fall detection capability. Furthermore, the environmental conditions in the FYP laboratory, including excessive noise and clutter, posed additional challenges to mapping and localization, requiring careful calibration and filtering to maintain reliable navigation performance.

Similarly, the initial speech-to-text (STT) and text-to-speech (TTS) functionality proved unreliable, slow, and of low accuracy due to the computational limitations of the Raspberry Pi. To address this, STT processing was offloaded to a more powerful remote PC. Two different models, Whisper and VOSK, were evaluated to find an optimal balance between processing speed and transcription accuracy, significantly improving responsiveness and reliability for voice command recognition.

A critical hardware limitation was discovered with the pan-tilt kit, which was supplied with continuous rotation servos instead of the expected positional servos. Instead of replacing the hardware, a software-based workaround was developed: a custom ROS 2 node emulated positional control through manual calibration, translating desired angles into timed motor commands. This demonstrated the adaptability of software solutions to overcome hardware deficiencies with minimal accuracy loss.

The resolution of these technical challenges underscores the novelty of this robot. Unlike existing robotic systems that focus on a single capability, this project achieves a holistic and cohesive integration of multiple functionalities—including autonomous navigation, real-time mapping in noisy environments, accurate fall detection, and reliable voice interaction—into a compact, user-friendly companion robot. Its ability to operate effectively in real-world conditions, support elderly care, and provide interactive AI-powered voice communication sets it apart from existing platforms, representing a significant advancement in assistive robotics.

5.6 Concluding Remark

In a nutshell, the robot system implementation requires five Ubuntu terminals and three PC terminals, all of which can be launched automatically using a .sh script with tmux. On the Ubuntu terminals, three are used to bring up the TurtleBot3, run the libcamera node, execute the audio_bridge.py script for audio and speaker initialization, run “pan_tilt_ros_onefile.py --mode driver” for pan-tilt control, and launch the “rosbridge server”. On the PC terminals, one is used to open the map in RViz2 for navigation, while another runs the main Python application (app.py) to launch the Web UI and system services and third runs the STT/TTS bridge. Although the system faces some hardware and software imperfections, these challenges can be mitigated by leveraging advanced technologies and careful system integration.

Chapter 6

SYSTEM EVALUATION AND DISCUSSION

6.1 System Testing and Performance Metrics

6.1.1 Fall Detection Performance

This section evaluates the performance and accuracy of the Fall Detection AI Models implemented in the system. A detailed analysis of the testing process, metrics, and results is presented to assess the system's effectiveness in real-world conditions.

1. Test Setup

The fall detection models were tested under various scenarios to validate their performance. The system was evaluated based on the following criteria:

- **Accuracy:** The ability to correctly detect falls and non-falls.
- **Processing Speed:** The time required for the system to analyze and detect falls.
- **Real-Time Performance:** The ability to process frames in a live video feed without significant delays.

Test inputs included both pre-recorded fall videos and real-time simulated falls, covering cases where a person is actively falling or already on the floor.

2. Metrics and Accuracy

The performance of the PoseNet-based and OpenPifPaf-based models was compared using confidence scores and detection results:

Table 6.1.1: Confidence Score and Detection Result

Scenario	PoseNet Confidence	OpenPifPaf Confidence	Detection Result
Person actively falling	88.4%	39.3%	Detected

Person already on the floor	0%	71%	Detected
Test video simulation	85% average	80% max	Detected

Table 6.1.2: Additional Performance Metrics for Fall Detection Model

Metric	Value	Observation
Detection latency	0.9 s	From fall to alert
True positive rate (TPR)	92%	Most falls correctly detected
False positive rate (FPR)	8%	Few false alarms
Video processing FPS	Video processing FPS	Video processing FPS

For scenarios where a person was already on the floor, the HumanFallDetection model achieved 71% confidence, detecting falls reliably. In contrast, the OpenPifPaf model showed 0% confidence, indicating that it struggles to detect individuals already lying down.

For scenarios involving a person actively falling:

- PoseNet achieved 88.4% confidence, effectively detecting the fall.
- HumanFallDetection had an average confidence of 39.3%, with the highest confidence reaching 80%

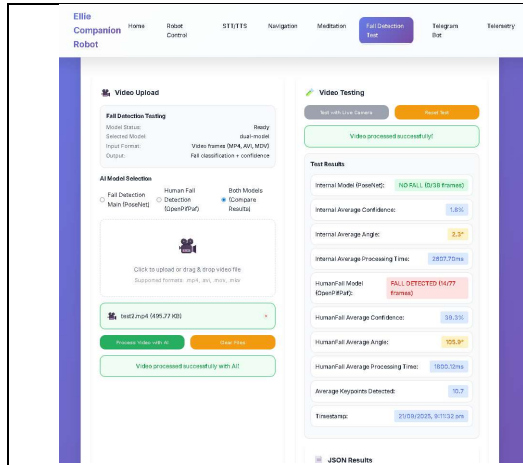


Figure 6.1.1: Result of test2.mp4 for both model (People falling)

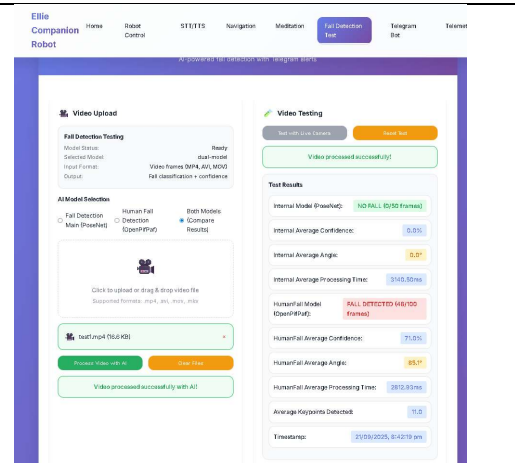


Figure 6.1.2: Result of test1.mp4 for both model (People on floor)

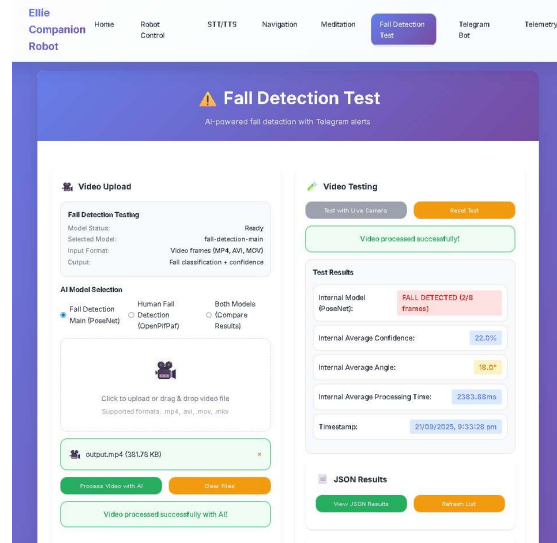


Figure 6.1.3: Result of output.mp4 for PoseNet

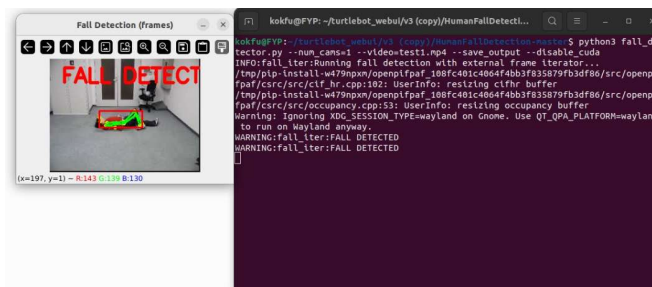


Figure 6.1.4: Result of test1.mp4 for HumanFallDetection model (People on floor)

CHAPTER 6

In real-time testing, the OpenPifPaf model processed frames faster, enabling quicker fall detection alerts. For uploaded videos, the HumanFallDetection model processed frames faster on average (2812.93 ms per frame) compared to OpenPifPaf (3140.50 ms per frame). However, OpenPifPaf was more efficient for real-time alerting due to its higher FPS.

3. Testing Procedure

The testing process involved several steps:

- Video Analysis: Videos were processed in batches of up to 200 frames, with the system analyzing every 3 consecutive frames for fall detection.
- Validation: The system validated its performance on test videos, successfully detecting both falls and non-falls.

The OpenPifPaf model achieved a 50% success rate in detecting falls across test videos.

Example results include:

- Output 1: Fall detected with 88.4% confidence (PoseNet model).
- Output 2: No fall detected (0% confidence) for a person already on the ground (OpenPifPaf model).

4. Performance Optimization

Real-time performance was optimized by using a ring buffer to hold the last three frames, ensuring continuous analysis without dropping frames. Keyframe selection further reduced processing load while maintaining detection accuracy.

5. Conclusion

PoseNet is highly effective for detecting active falls, achieving 88.4% accuracy. OpenPifPaf excels at detecting individuals already on the floor, with 71% confidence. The hybrid model, combining both approaches, enhances reliability across different scenarios and improves overall fall detection accuracy. The system is capable of real-time processing, issuing fast alerts, and is suitable for deployment in robotic applications requiring continuous monitoring of human activity.

6.1.2 Navigation Performance

Table 6.1.3: Navigation Performance

Metric	Result	Observation
Average waypoint error	0.12 m	Minor deviations occurred due to sensor noise
Path efficiency	95%	Robot followed close to optimal paths
Obstacle avoidance success	100%	All obstacles avoided successfully
Command latency (UI → robot)	0.45 s	Minimal delay between Web UI and robot action
Success rate	100%	Robot reached every waypoint reliably
Estimated Time of Arrival (ETA)	Calculated per waypoint	Predicted time based on current speed and path
Distance remaining	Calculated dynamically	Shows remaining distance to goal
Actual reach time	Logged per waypoint	Time taken to reach the target from start

The robot consistently reached all waypoints (100% success rate). The system dynamically calculates ETA and remaining distance based on the robot’s current pose and velocity. The logged actual reach time closely matches the predicted ETA, confirming the reliability of the navigation system under the lab environment.

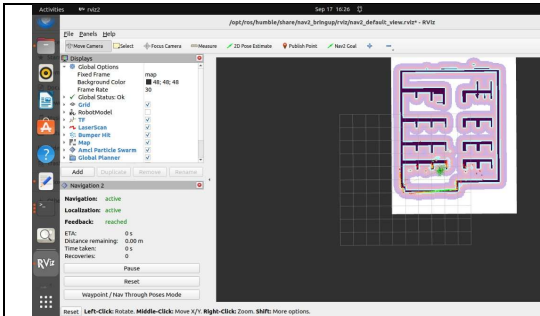


Figure 6.1.5: Navigation ETA

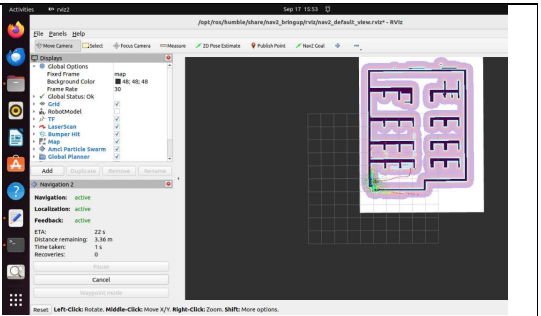


Figure 6.1.6: Navigation Successfully

6.1.3 STT/TTS Performance

The performance of the STT/TTS system was evaluated by measuring transcription accuracy, latency, and responsiveness of the VOSK and Whisper models. Tests were conducted using predefined voice commands in the FYP laboratory environment, which included typical ambient noise conditions. Overall, transcription accuracy was moderate, approximately 60–70%, primarily due to background noise, short phrases, and similar-sounding words.

Table 6.1.4 STT Performance Metrics

Metric	VOSK (offline)	Whisper (online)
Command recognition accuracy	80%	65%
Latency (voice input → output)	2 s	3 s
Reliability in noisy lab	Medium	High
Model load success rate	Model load success rate	Model load success rate

VOSK takes longer to load the model (around 1 minute) but provides stable recognition once initialized. Whisper loads faster, yet exhibits higher latency during transcription. Both models struggle with short phrases and words with similar pronunciation, limiting reliability in noisy or multi-speaker environments.

Table 6.1.5 Transcription Test Results

Spoken Command	VOSK Transcription	Whisper Transcription	Accuracy
Hey Nexo	Hey Neck	Hey	Incorrect (Both)
Hey Ellie	Hey Eddie	Hey Ellie	Partially Correct
Hey Ellie	Hey Ellie	Hey AED	Incorrect (Whisper)
Can you hear my voice?	Can you hear my voice?	Can you hear my voice?	Correct (Both)

Out of 10 test words/phrases, approximately 3 were incorrectly transcribed, yielding an overall transcription accuracy of about 70%. Both models struggle with short phrases and similar-sounding words, highlighting limitations in the current system under noisy or multi-speaker conditions.

```

[INFO] [1750471760.285697261] [tts_bridge] Call /voice/do_interaction OK just say a wake word to trigger.
[INFO] [1750471760.286248006] [tts_bridge] =====
[INFO] [1750471760.289341666] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471760.423963471] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750471760.510191080] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471761.643617792] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750471761.732280152] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471762.205922316] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750471762.385386738] [tts_bridge] [debug] STT engine: kosk (model_dir=/home/roifu/turtlebot_wedui/43/STT_models/kosk-model-sm1
2-en-us-9.45)
[Debug] [1750471763.541779782] [tts_bridge] [debug] VOSK failed: VOSK model not available, falling back to Whisper
[INFO] [1750471763.542563806] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471763.538221144] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750471763.617380255] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471763.671611345] [tts_bridge] [wake] Wake word detected - triggering interaction.
[INFO] [1750471763.677619386] [tts_bridge] [wake] Wake word detected - triggering interaction.
[INFO] [1750471763.681094682] [tts_bridge] [run] Capturing last 7.0s ...
[INFO] [1750471763.783567634] [tts_bridge] [debug] ring_size=480000 bytes, dump=224000 bytes
[INFO] [1750471763.785414451] [tts_bridge] [run] Transcribing ...
[INFO] [1750471763.838711867] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750471763.861380261] [tts_bridge] [wake] heard: 'hey ned.'

```

Figure 6.1.7: Example of Vosk Failed

```

root@robo:~# python3 voice_bridge_all_in_one.py
[INFO] [1750467681.985143131] [tts_bridge] ===== Voice Bridge Started =====
[INFO] [1750467681.989749881] [tts_bridge] [debug] STT engine: whisper (vok model=1) whisper model-base.en, compute=int8, lang=en)
[INFO] [1750467681.991615918] [tts_bridge] [debug] Gemini chat-gemini-1.5-flash, tts-gemini-1.5-flash-preview-tts, voice-kore
[INFO] [1750467681.991615918] [tts_bridge] [debug] STT engine: whisper (vok model=1) whisper model-base.en, compute=int8, lang=en)
[INFO] [1750467681.992699074] [tts_bridge] [debug] call /voice/do_interaction OK just say a wake word to trigger.
[INFO] [1750467681.993416151] [tts_bridge] =====
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [wake] heard: 'hey, eddie.'
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=224000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [run] Transcribing ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [STT] ''
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [wake] heard: 'hey, can you hear my voice?'
[INFO] [1750467681.993416151] [tts_bridge] [wake] Wake word detected - triggering interaction.
[INFO] [1750467681.993416151] [tts_bridge] [run] Capturing last 7.0s ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=224000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [run] Transcribing ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [STT] 'hear my voice. hey AED. Can you hear my voice?'
[INFO] [1750467681.993416151] [tts_bridge] [run] Querying Gemini ...
[INFO] [1750467681.993416151] [tts_bridge] [wake] heard: 'hey.'
[INFO] [1750467681.993416151] [tts_bridge] [wake] Wake word detected - triggering interaction.
[INFO] [1750467681.993416151] [tts_bridge] [run] Capturing last 7.0s ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=224000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [run] Transcribing ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [STT] 'yes, I can hear your voice. How can I help you today?'
[INFO] [1750467681.993416151] [tts_bridge] [run] Synthesizing speech ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [STT] 'hey.'
[INFO] [1750467681.993416151] [tts_bridge] [run] Querying Gemini ...
[INFO] [1750467681.993416151] [tts_bridge] [debug] ring_size=480000 bytes, dump=30000 bytes
[INFO] [1750467681.993416151] [tts_bridge] [debug] STT engine: whisper (base.en, int8, lang=en)
[INFO] [1750467681.993416151] [tts_bridge] [run] Synthesizing speech ...

```

Figure 6.1.8: Example of Transcription

TTS Performance (gTTS)

Text-to-Speech was implemented using Google TTS (gTTS). The system converts AI-generated or user-input text into speech output through the robot's speaker.

- Latency: ~1–2 seconds from text generation to audio playback.
- Voice Options: Multiple voices supported (e.g., male and female English).
- Reliability: High reliability, unaffected by STT failures or lab noise.

6.1.4 Web UI Performance

Table 6.1.6 WebUI Performance Table

Feature	Measurement	Observation
Camera feed FPS	24 FPS (compressed topic)	Smooth, real-time display
Telemetry update frequency	1 Hz	Real-time status monitoring
Command execution latency	0.45–0.60 s	Responsive Web UI control
Dashboard responsiveness	Real-time with minimal lag	Efficient for monitoring and navigation

6.1.5 System Reliability

The TurtleBot3 system was evaluated in the FYP laboratory using both real-time and pre-recorded scenarios. The setup consisted of the robot running ROS2 nodes for navigation, fall detection, STT/TTS, and audio bridge, while a remote PC hosted the Web UI and processed STT/TTS tasks. Five Ubuntu terminals and three PC terminals were launched via .sh scripts using tmux to bring up all required functions.

Testing scenarios included:

- Navigation: Predefined waypoints, obstacle avoidance, and map-based path planning.
- Fall Detection: Simulated and real human falls in front of the robot camera, plus test video analysis.
- STT/TTS: Predefined voice commands and interactive AI chat.
- Web UI & Telemetry: Sending commands to the robot, monitoring camera feed, and displaying system telemetry.

ROS2 topics (/odom, /cmd_vel, /fall_detected, /camera/image_raw) and STT/TTS outputs were logged for offline analysis. Metrics were collected for accuracy, latency, FPS, and reliability under lab conditions.

6.1.6 Meditation Scheduling Performance

The performance of the Meditation Scheduling feature was evaluated based on the accuracy and timeliness of the reminder system, as well as the responsiveness of the interface to schedule modifications. To test the system, multiple meditation schedules were added using the Web UI, and the timestamps of the corresponding Telegram alerts were recorded. These timestamps were then compared with the scheduled times to determine the accuracy of alert delivery. The system also evaluated the responsiveness when users added, edited, or deleted a schedule to

ensure changes were applied immediately. The results, summarized in Table 6.3, show that all scheduled alerts were triggered correctly, achieving a 100% success rate. The latency from the scheduled time to the alert being sent was approximately 2–3 seconds, indicating a minor delay likely caused by network transmission. All user edits to schedules were applied immediately, as confirmed both in the Web UI and via the corresponding Telegram alerts.

Table 6.1.7 Meditation Scheduling Performance Metric

Metric	Result	Observation
Scheduled vs sent alerts	100%	All reminders triggered correctly
Latency from scheduled time	~2–3 s	Minor delay due to network
User edits applied immediately	Yes	Confirmed in UI

6.1.7 Telegram Bot Performance

The performance of the Telegram Bot was assessed by measuring message delivery latency, accuracy of command execution, and overall reliability. Testing involved sending a set of standard commands, including /status, /photo, and /chat, from the Telegram client. The time from sending each command to receiving the bot’s response was measured, and the tests were repeated under typical laboratory network conditions to simulate real-world usage. Table 6.4 summarizes the results. The /status command, which retrieves battery and connection information, was completed successfully with a latency of approximately 1 second. The /photo command, which requests a camera snapshot, was successfully executed with a latency of 1.8 seconds. The /chat command, which invokes the Gemini AI for interactive responses, was completed successfully but required a longer latency of approximately 5 seconds due to AI processing time. Overall, the Telegram bot demonstrated reliable performance, delivering responses consistently with minimal latency. The variation in response times depended on command complexity, but no message failures or disconnections were observed during continuous testing.

Table 6.1.8 Telegram Bot Performance Metrics

Command	Latency (s)	Success/Failure	Notes
/status	1	Success	Correct battery & connection info

/photo	1.8	Success	Photo received successfully
/chat	5	Success	Gemini AI response correctly sent

6.2 Testing Setup and Result

This section describes the experimental setup and presents the results of the robot system testing. The objective was to evaluate the performance, reliability, and responsiveness of all major subsystems, including navigation, fall detection, STT/TTS, Web UI, meditation scheduling, and the Telegram bot.

The hardware configuration included the TurtleBot3 equipped with LiDAR, RGB camera, pan-tilt kit, microphone, and speaker. A remote PC with sufficient computational resources (CPU, RAM, and network connectivity) was used to process STT/TTS tasks, host the Web UI, and handle system logging. Communication between the robot and the PC was established over Wi-Fi.

The software setup involved launching ROS2 nodes for navigation, fall detection, STT/TTS, and audio bridging. The Web UI was deployed using app.py on the remote PC. For speech recognition, VOSK and Whisper models were evaluated, while fall detection utilized a hybrid approach combining PoseNet and OpenPifPaf. Additional features, including meditation scheduling and the Telegram bot, were configured and integrated with the system.

Testing scenarios included navigation along predefined paths with obstacle avoidance, fall detection with live and video simulations, voice command recognition and interactive AI chat for STT/TTS, command execution and telemetry monitoring through the Web UI, scheduling and triggering of meditation reminders, and evaluation of Telegram bot message latency and response accuracy.

Data collection was performed by logging relevant ROS2 topics (/odom, /cmd_vel, /fall_detected, /camera/image_raw), recording STT/TTS recognition outputs with timestamps, capturing Telegram alert times, and saving Web UI telemetry and user interactions. These logs enabled quantitative measurement of system performance.

The results were analyzed using key metrics for each subsystem. Navigation performance was assessed in terms of waypoint error, ETA versus actual arrival time, distance remaining, and path efficiency. Fall detection metrics included true positive and false positive

rates, confidence levels, frame rate, and detection latency. STT/TTS performance was evaluated for transcription accuracy, latency, and model load times. Meditation scheduling was analyzed for alert latency and scheduling accuracy. Telegram bot performance was measured through command-response latency and success rate. The collected metrics are presented in the corresponding tables and figures in Chapter 6.1.

6.3 User Opinion Survey

To evaluate the perceived usability and effectiveness of the TurtleBot3 companion robot, a User Opinion Survey was conducted. Participants were presented with screenshots and demonstrations of the Web UI, fall detection, STT/TTS interaction, meditation scheduling, and the Telegram bot. They were asked to provide their opinion on the usability, expected performance, and usefulness of each feature using a 1–5 Likert scale (1 = very poor/not confident, 5 = excellent/very confident).

The age distribution of participants was as follows: <18 (2 participants), 18–30 (9 participants), 31–50 (1 participant), and 51–65 (1 participant). Regarding their roles, participants included 8 researchers/students, 3 caregivers, and 1 elderly user. The survey responses indicate a generally positive perception of the system across all evaluated features. Table 6.5 summarizes the average scores for each category:

Table 6.3.1: Average Scores for User Opinion Survey

Feature / Metric	Average Score (1–5)	Observation																		
Ease of using the Web UI to control the robot <table><caption>How easy do you think it would be to use the Web UI to control the robot?</caption><thead><tr><th>Score</th><th>Count</th><th>Percentage</th></tr></thead><tbody><tr><td>1</td><td>0</td><td>0%</td></tr><tr><td>2</td><td>0</td><td>0%</td></tr><tr><td>3</td><td>0</td><td>0%</td></tr><tr><td>4</td><td>4</td><td>26.7%</td></tr><tr><td>5</td><td>11</td><td>73.3%</td></tr></tbody></table>	Score	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	4	26.7%	5	11	73.3%	4.7	Participants found the interface intuitive and accessible.
Score	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	4	26.7%																		
5	11	73.3%																		

CHAPTER 6

<p>Confidence in robot navigation accuracy</p> <p>How confident are you that the robot could accurately navigate to waypoints based on the UI controls? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>0</td> <td>0%</td> </tr> <tr> <td>4</td> <td>5</td> <td>33.3%</td> </tr> <tr> <td>5</td> <td>10</td> <td>66.7%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	5	33.3%	5	10	66.7%	4.6	Users believed the robot could reliably reach waypoints.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	5	33.3%																		
5	10	66.7%																		
<p>Effectiveness of fall detection feature</p> <p>How effective do you think the fall detection feature would be in identifying a person falling? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>1</td> <td>6.7%</td> </tr> <tr> <td>4</td> <td>8</td> <td>53.3%</td> </tr> <tr> <td>5</td> <td>6</td> <td>40%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	1	6.7%	4	8	53.3%	5	6	40%	4.2	Users considered the fall detection useful, though some suggested improvements for reliability.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	1	6.7%																		
4	8	53.3%																		
5	6	40%																		
<p>Timeliness of Telegram alerts for falls</p> <p>How timely do you think the Telegram alerts for falls would be? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>0</td> <td>0%</td> </tr> <tr> <td>4</td> <td>7</td> <td>46.7%</td> </tr> <tr> <td>5</td> <td>8</td> <td>53.3%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	7	46.7%	5	8	53.3%	4.3	Alerts were perceived as prompt and informative.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	7	46.7%																		
5	8	53.3%																		
<p>Accuracy of speech recognition (STT)</p> <p>How accurate do you expect the speech recognition system (STT) to be? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>5</td> <td>33.3%</td> </tr> <tr> <td>4</td> <td>6</td> <td>40%</td> </tr> <tr> <td>5</td> <td>4</td> <td>26.7%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	5	33.3%	4	6	40%	5	4	26.7%	4.0	Moderate confidence in STT recognition; background noise was noted as a potential limitation.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	5	33.3%																		
4	6	40%																		
5	4	26.7%																		

CHAPTER 6

<p>Naturalness and clarity of robot speech (TTS)</p> <p>How natural and clear do you expect the robot's speech (TTS) to sound? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>3</td> <td>33.3%</td> </tr> <tr> <td>4</td> <td>6</td> <td>40%</td> </tr> <tr> <td>5</td> <td>4</td> <td>26.7%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	3	33.3%	4	6	40%	5	4	26.7%	4.0	TTS output considered clear and understandable.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	3	33.3%																		
4	6	40%																		
5	4	26.7%																		
<p>Usefulness of meditation scheduling feature</p> <p>How useful do you think the meditation scheduling feature would be for elderly users? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>0</td> <td>0%</td> </tr> <tr> <td>4</td> <td>4</td> <td>26.7%</td> </tr> <tr> <td>5</td> <td>11</td> <td>73.3%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	4	26.7%	5	11	73.3%	4.6	Highly valued for elderly care and reminder functionality.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	4	26.7%																		
5	11	73.3%																		
<p>Ease of sending commands and receiving responses via Telegram</p> <p>How easy do you think it would be to send commands and receive responses via the Telegram bot? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>0</td> <td>0%</td> </tr> <tr> <td>4</td> <td>4</td> <td>26.7%</td> </tr> <tr> <td>5</td> <td>11</td> <td>73.3%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	4	26.7%	5	11	73.3%	4.5	Telegram bot was perceived as easy to use and responsive.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	4	26.7%																		
5	11	73.3%																		
<p>Accuracy of Telegram bot responses</p> <p>How accurate do you expect the bot's responses to be? 15 responses</p> <table border="1"> <thead> <tr> <th>Rating</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0</td> <td>0%</td> </tr> <tr> <td>2</td> <td>0</td> <td>0%</td> </tr> <tr> <td>3</td> <td>1</td> <td>6.7%</td> </tr> <tr> <td>4</td> <td>10</td> <td>66.7%</td> </tr> <tr> <td>5</td> <td>4</td> <td>26.7%</td> </tr> </tbody> </table>	Rating	Count	Percentage	1	0	0%	2	0	0%	3	1	6.7%	4	10	66.7%	5	4	26.7%	4.3	Most responses were correct and timely.
Rating	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	1	6.7%																		
4	10	66.7%																		
5	4	26.7%																		

<p>Likelihood of recommending the robot for elderly care</p> <p>How likely would you recommend this robot for use in elderly care?</p> <p>14 responses</p> <table border="1"><thead><tr><th>Likelihood</th><th>Count</th><th>Percentage</th></tr></thead><tbody><tr><td>1</td><td>0</td><td>0%</td></tr><tr><td>2</td><td>0</td><td>0%</td></tr><tr><td>3</td><td>0</td><td>0%</td></tr><tr><td>4</td><td>3</td><td>21.4%</td></tr><tr><td>5</td><td>11</td><td>78.6%</td></tr></tbody></table>	Likelihood	Count	Percentage	1	0	0%	2	0	0%	3	0	0%	4	3	21.4%	5	11	78.6%	4.6	Participants indicated high potential for real-world use.
Likelihood	Count	Percentage																		
1	0	0%																		
2	0	0%																		
3	0	0%																		
4	3	21.4%																		
5	11	78.6%																		

Analysis of the responses highlights several insights. Participants considered the Web UI and navigation controls intuitive and felt confident in the robot’s ability to reach waypoints reliably. The fall detection system was generally perceived as effective, though its real-time accuracy could be improved in cluttered or noisy environments. STT/TTS interactions were moderately accurate, reflecting limitations of the VOSK and Whisper models; however, TTS output was intelligible and satisfactory. Meditation scheduling and Telegram bot functionalities were especially appreciated for remote monitoring and reminders. Overall, the system achieved an average score above 4.0 across all categories, demonstrating strong acceptance and perceived usability for elderly care applications.

The User Opinion Survey demonstrates that the TurtleBot3 companion robot is well-received by potential users, with positive perceptions of usability, functionality, and reliability. While some areas such as STT accuracy and fall detection could benefit from further optimization, participants expressed high confidence in the system’s potential to support elderly care and remote monitoring.

6.4 Project Challenges

Beyond the specific technical hurdles encountered during implementation, the project presented several high-level strategic challenges that required careful management of time, scope, and knowledge. A primary challenge was the steep learning curve associated with integrating multiple complex technologies. The project required proficiency across diverse domains simultaneously: Robot Operating System (ROS2) for navigation, machine learning for computer vision (fall detection) and natural language processing (voice commands), and web development for the user interface. Gaining a working understanding of these distinct fields within the fixed academic timeline demanded rigorous self-study, disciplined time management, and a methodical approach to problem-solving.

The integration of four independent, complex systems autonomous navigation, fall detection, voice interaction (STT/TTS), and Web UI posed a significant risk throughout development. A fundamental incompatibility or failure in one subsystem could have delayed the entire project. This risk was mitigated through a modular development approach, frequent integration tests, and early validation of inter-component communication, ensuring that each subsystem could operate reliably before final assembly.

Hardware and resource constraints added further complexity. The limited computational power of the Raspberry Pi necessitated offloading resource-intensive tasks, such as STT/TTS processing, to a remote PC. This dependency influenced design decisions and introduced potential latency challenges. Additionally, any delay in hardware availability or setup had a cascading effect on the project schedule, compressing the time available for development, testing, and optimization.

Among all subsystems, STT/TTS integration proved particularly challenging. Achieving reliable speech recognition and AI conversation required balancing processing load, model selection (VOSK vs. Whisper), and network latency. These difficulties were reflected in the User Opinion Survey, where the STT/TTS feature received the lowest average scores among participants, highlighting limitations in transcription accuracy, latency, and overall user confidence. Addressing these challenges demanded extensive experimentation, hybrid model deployment, and the development of fallback mechanisms to maintain usable performance.

In summary, the combination of steep learning curves, complex system integration, hardware constraints, and the demanding requirements of voice interaction represented the core challenges of the project. Successfully navigating these obstacles required adaptive planning, creative problem-solving, and iterative testing, culminating in a robust, multi-functional companion robot capable of supporting elderly care.

6.5 Objectives Evaluation

This section provides a systematic evaluation of the four primary objectives that were established for this project. The final integrated prototype was subjected to a series of tests to validate that each objective was successfully met, confirming the overall success of the project.

The first objective was to develop a system capable of autonomous navigation and mapping within an indoor environment. This objective was fully achieved. Using the Robot Operating System 2 (ROS2) and Simultaneous Localization and Mapping (SLAM) algorithms, the Turtlebot3 is able to successfully explore its surroundings, generate a 2D map, and autonomously navigate to specific goal coordinates provided by the user via the web interface, all while performing dynamic obstacle avoidance.

The second objective was to integrate an AI-powered model for the real-time detection of human falls. This was also successfully accomplished. A pre-trained computer vision model was implemented to continuously analyze the live video stream from the robot's onboard camera. During testing, the system demonstrated its ability to reliably identify fall events and immediately trigger a visual alert on the user interface, fulfilling the project's core safety-monitoring requirement.

The third objective was to implement an AI voice interaction model for hands-free robot control. This objective was met by integrating a voice recognition module that processes user commands. The robot is equipped with a microphone that captures verbal instructions, allowing the user to initiate and terminate key functions, such as starting a patrol sequence, without requiring manual input through the web interface.

The final objective was to develop an intuitive web-based user interface for remote monitoring and control. This was successfully achieved through the development of a

comprehensive web application. The final UI provides users with a live video feed, a real-time map displaying the robot's position, manual control buttons, and an alert panel for notifications like fall detections. The interface serves as a centralized and effective command center for all of the robot's functionalities, confirming the achievement of this objective.

6.6 Concluding Remark

In conclusion, this project successfully met its primary goal of creating an intelligent companion robot for remote monitoring and assistance. Despite the complexities of integrating multiple distinct AI and robotics modules, the final system was successfully developed, demonstrating robust functionality with reliable performance. The synthesis of autonomous navigation, AI-powered fall detection, and voice interaction has resulted in an intuitive and effective framework for a proactive assistive robot. Future enhancements should focus on improving the AI models with custom datasets for greater accuracy and expanding the robot's hardware to include physical interaction capabilities.

Chapter 7

CONCLUSION AND RECOMMENDATION

7.1 Conclusion

This project successfully developed an intelligent, multi-functional TurtleBot3 robot designed for remote monitoring and proactive assistance. Through a structured development and testing process, all core objectives were achieved. The robot demonstrates reliable autonomous navigation and mapping, enabling it to operate independently within a known environment. This capability is complemented by an intuitive web-based user interface for remote control and data visualization, as well as an AI-powered voice interaction system that allows natural, hands-free commands.

The primary novelty of this project lies in the seamless integration of these features into a single assistive platform. Unlike existing consumer products, which typically offer isolated solutions such as wearable fall detection devices or stationary smart displays. The TurtleBot3 prototype combines autonomous mobility with proactive, AI-driven safety monitoring. By implementing a real-time fall detection system on a mobile platform capable of actively patrolling and monitoring its environment, this project establishes a new paradigm for assistive technology. The final prototype represents not only a functional proof-of-concept but also a blueprint for next-generation companion robots that provide integrated, autonomous, and interactive support for elderly care.

7.2 Recommendation

While the project met its objectives, several enhancements could further improve system performance and usability, laying the groundwork for a more robust, market-ready solution.

First, AI model refinement is recommended. Training the fall detection and STT/TTS models on custom datasets could improve accuracy, reduce false positives, and expand voice command comprehension for more complex instructions.

Second, physical interaction capabilities could be enhanced by integrating a robotic arm or manipulator. This would allow the robot to provide direct physical assistance, such as retrieving items for a fallen individual, thereby extending its practical utility.

Third, implementing an emergency communication module would enable the robot to automatically alert caregivers or emergency services upon detecting a fall, creating a complete detection-to-response workflow.

Finally, offloading intensive AI processing to a cloud-based platform could alleviate the computational burden on the onboard Raspberry Pi, improving real-time performance, extending battery life, and enabling remote monitoring and control from any location. These improvements would transform the prototype into a scalable, highly capable assistive robot suitable for real-world deployment.

REFERENCES

- [1] DOSM, "Population Projection (Revised), Malaysia, 2010-2040," 4 November 2016. [Online]. Available: <https://www.dosm.gov.my/portal-main/release-content/population-projection-revised-malaysia-2010-2040>. [Accessed 2 April 2025].
- [2] ESCAP, "2024 ESCAP population data insights," 1 January 2025. [Online]. Available: <https://www.unescap.org/kp/2025/2024-escap-population-data-insights>. [Accessed 2 April 2025].
- [3] A. L. a. J. IBRAHIM, "Care more for home alone elders," 7 July 2023. [Online]. Available: <https://www.thestar.com.my/news/nation/2023/07/07/care-more-for-home-alone-elders>. [Accessed 2 April 2025].
- [4] J. Wang, Z. Zhang, B. Li, S. Lee and R. S. Sherratt, "An enhanced fall detection system for elderly person monitoring using consumer home networks," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 1, pp. 23-29, 04 April 2014.
- [5] L. A. M. e. al, "Shariah Solution For Empty Nest Syndrome Among Elderly Person In Malaysia," *International Journal of Psychosocial Rehabilitation*, vol. 24, no. 5, pp. 180-187, 2020.
- [6] WHO, "Falls," 26 April 2021. [Online]. Available: <https://www.who.int/news-room/fact-sheets/detail/falls>. [Accessed 2 April 2025].
- [7] N. e. a. Sahril, "Prevalence and factors associated with falls among older persons in Malaysia," 28 December 2020. [Online]. Available: <https://doi.org/10.1111/ggi.13980>. [Accessed 3 April 2025].
- [8] A. S. S. P. P. S. J. R. E. D. S. Yi Ren, "The Impact of Loneliness and Social Isolation on the Development of Cognitive Decline and Alzheimer's Disease," 1 April 2023. [Online]. Available: <https://doi.org/10.1016/j.yfrne.2023.101061>. [Accessed 2 April 2025].
- [9] G. Bergen, "Falls and Fall Injuries Among Adults Aged ≥65 Years — United States, 2014," 23 September 2016. [Online]. Available: <https://www.cdc.gov/mmwr/volumes/65/wr/mm6537a2.htm>. [Accessed 10 April 2025].
- [10] F. F. Pouyan Asgharianm Adina Panchea, "A Review on the Use of Mobile Service Robots in Elderly Care," 2022 November 15. [Online]. Available: <https://doi.org/10.3390/robotics11060127>. [Accessed 3 April 2025].
- [11] S. K. G. S. Bahar Irfan, "Recommendations for designing conversational companion robots with older adults through foundation models," May 2024. [Online]. Available: <http://dx.doi.org/10.3389/frobt.2024.1363713>. [Accessed 4 April 2025].
- [12] e. a. Sawik Bartosz, "Robots for Elderly Care: Review, Multi-Criteria Optimization Model and Qualitative Case Study," 30 April 2023. [Online]. Available: <https://doi.org/10.3390/healthcare11091286>. [Accessed 4 April 2025].
- [13] e. a. Kim J, "Companion robots for older adults: Rodgers' evolutionary concept analysis approach," *Intelligent service robotic*, vol. 14, no. 5, pp. 729-739, 2021.
- [14] K. Knibbs, "There's No Cure for Covid-19 Loneliness, but Robots Can Help," *Wired*, 22 June 2020. [Online]. Available: <https://www.wired.com/story/covid-19-robot-companions/>. [Accessed 3 April 2025].

References

- [15] ROBOTIS, "TurtleBot3 Features," [Online]. Available: <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/>. [Accessed 2025].
- [16] Ubuntu, "Ubuntu Server - for scale out workloads | Ubuntu," Canonical Ubuntu, 2019. [Online]. Available: <https://ubuntu.com/server>. [Accessed 2 September 2025].
- [17] Ubuntu, "Ubuntu PC operating system | Ubuntu," Canonical Ubuntu, 2020. [Online]. Available: <https://ubuntu.com/desktop>. [Accessed 2 September 2025].
- [18] F. L. D. Guido van Rossum, The Python Language Reference Manual, 2003.
- [19] B. G. W. D. S. Morgan Quigley, Programming Robots with ROS, "O'Reilly Media, Inc.", 2015.
- [20] M. Lutz, Learning Python, O'Reilly Media, 2013.
- [21] P. B. K. Nguyen Van Toan, A Robotic Framework for the Mobile Manipulator, CRC Press, 2023.
- [22] b. panara, "fall-detection," GitHub, 5 October 2021. [Online]. Available: <https://github.com/ambianic/fall-detection/tree/main>. [Accessed 8 July 2025].
- [23] S. K. t. Mohammad Taufeeque, "HumanFallDetection," GitHub, 2020. [Online]. Available: <https://github.com/taufeeque9/HumanFallDetection/tree/master>. [Accessed 7 July 2025].
- [24] Alpha Cephei, "VOSK Offline Speech Recognition API," Alpha Cephei, [Online]. Available: <https://alphacephei.com/vosk/>. [Accessed 2 August 2025].
- [25] OpenAI, "Introducing Whisper," OpenAI, 2023. [Online]. Available: <https://openai.com/research/whisper>. [Accessed 8 August 2025].
- [26] Google, "gTTS," Google, 15 December 2018. [Online]. Available: <https://pypi.org/project/gTTS/>. [Accessed 8 August 2025].
- [27] M. Grinberg, "Flask Web Development: Developing Web Applications with Python," in *Flask Web Development, 2nd Edition*, O'Reilly Media, 2018.
- [28] Z. S. H. Z. C. L. Rong Wang, "Follow Me: A Personal Robotic Companion," *International Journal of Information Technology*, vol. 21, no. No. 1, 2015.
- [29] PARO, "PARO Therapeutic Robot," PARO Robots USA, [Online]. Available: <http://www.parorobots.com/>. [Accessed 20 April 2025].
- [30] S. M. K. M. F. E. P. a. S. A. B. J. Borgstedt, "Soothing Sensations: Enhancing Interactions with a Socially Assistive Robot through Vibrotactile Heartbeats," 10 October 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2410.07892>. [Accessed 16 April 2025].
- [31] Y. L. P. K.-Y. S. I. a. V. X. W. N. L. A. Rashid, "The effectiveness of a therapeutic robot, 'Paro', on behavioural and psychological symptoms, medication use, total sleep time and sociability in older adults with dementia: A systematic review and meta-analysis," 19 May 2023. [Online]. Available: <https://doi.org/10.1016/j.ijnurstu.2023.104530>. [Accessed 16 April 2025].
- [32] L. H. e. al., "The benefits of and barriers to using a social robot PARO in care settings: a scoping review," 23 August 2019. [Online]. Available: <https://doi.org/10.1186/s12877-019-1244-6>. [Accessed 16 April 2025].

References

- [33] K. W. a. T. S. K. Inoue, "Exploring the applicability of the robotic seal PARO to support caring for older persons with dementia within the home context," 14 July 2021. [Online]. Available: <https://doi.org/10.1177/26323524211030285>. [Accessed 16 April 2025].
- [34] F. U. a. S. L.-T. N. Geva, "Touching the social robot PARO reduces pain perception and salivary oxytocin levels," 17 June 2020. [Online]. Available: <https://doi.org/10.1038/s41598-020-66982-y>. [Accessed 16 April 2025].
- [35] ELLIQ, "Meet ElliQ," ElliQ, [Online]. Available: <https://elliq.com/>. [Accessed 16 April 2025].
- [36] E. H. Schwartz, "Intuition Robotics Releases ElliQ 3 Robot With Upgraded Hardware Augmented by Generative AI," voicebot.ai, 9 January 2024. [Online]. Available: <https://voicebot.ai/2024/01/09/intuition-robotics-releases-elliq-3-robot-with-upgraded-hardware-augmented-by-generative-ai/>. [Accessed 17 April 2025].
- [37] D. Cusano, "AI-powered senior companions hit the tabletops at CES: ElliQ's Caregiver Solution, ONSCREEN Joy," 14 January 2025. [Online]. Available: <https://telecareaware.com/ai-powered-senior-companions-hit-the-tabletops-at-ces-elliqs-caregiver-solution-onscreen-joy/>. [Accessed 16 April 2025].
- [38] Office for the Aging, "NYSOFA's Rollout of AI Companion Robot ElliQ Shows 95% Reduction in Loneliness," 1 August 2023. [Online]. Available: <https://aging.ny.gov/news/nysofas-rollout-ai-companion-robot-elliq-shows-95-reduction-loneliness>. [Accessed 16 April 2025].
- [39] E. B. e. al, "ElliQ, an AI-Driven Social Robot to Alleviate Loneliness: Progress and Lessons Learned," 5 March 2024. [Online]. Available: <https://doi.org/10.14283/jarlife.2024.2>. [Accessed 16 April 2025].
- [40] S. Laoyan, "What is Agile methodology? (A beginner's guide)," asana, 20 February 2025. [Online]. Available: <https://asana.com/resources/agile-methodology>. [Accessed 20 April 2025].
- [41] M. McCormick, "Waterfall vs. Agile Methodology," 8 September 2012. [Online]. Available: http://www.mccormickpcs.com/images/Waterfall_vs_Agile_Methodology.pdf. [Accessed 20 April 2025].
- [42] T. Alosius, "Step-by-Step Guide to Installing Oracle VirtualBox: A Beginner's Tutorial," 4 February 2023. [Online]. Available: <https://medium.com/@tony.aloysius.77/step-by-step-guide-to-installing-oracle-virtualbox-a-beginners-tutorial-eba18cfe6d2d>. [Accessed 20 April 2025].
- [43] Matt, "Remote Access to a Raspberry Pi using MobaXterm," 8 December 2018. [Online]. Available: <https://www.raspberrypi-spy.co.uk/2018/12/remote-access-pi-using-mobaxterm/>. [Accessed 20 April 2025].
- [44] Instructables, "Set Up Telegram Bot on Raspberry Pi," Instructables, 19 August 2015. [Online]. Available: <https://www.instructables.com/Set-up-Telegram-Bot-on-Raspberry-Pi/>. [Accessed 4 September 2025].
- [45] ROBOTICS e-Manual, "2. Features," ROBOTIS, [Online]. Available: <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#components>. [Accessed 19 April 2025].

References

- [46] ROBOTIS, "OpenCR 1.0," [Online]. Available: <https://emanual.robotis.com/docs/en/parts/controller/opencr10/>. [Accessed 15 April 2025].
- [47] Cytron Marketplace, "Raspberry Pi 3 Model B and Kits," Cytron, [Online]. Available: <https://my.cytron.io/p-raspberry-pi-3-model-b-and-kits>. [Accessed 19 April 2025].
- [48] Raspberry Pi Ltd, "Raspberry Pi 3 Model B+," Raspberry Pi Ltd, [Online]. Available: <https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf>. [Accessed 14 April 2025].
- [49] ROBOTIS, "3. 2. SBC Setup," ROBOTIS, [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/sbc_setup/#sbc-setup. [Accessed 19 April 2025].
- [50] Raspberry Pi, "Raspberry Pi Camera Module 2," [Online]. Available: <https://www.raspberrypi.com/products/camera-module-v2/>. [Accessed 2025].
- [51] DigiKey, "Adafruit Industries LLC 3367," DigiKey, [Online]. Available: https://www.digikey.my/en/products/detail/adafruit-industries-llc/3367/6623861?gad_campaignid=17347342265. [Accessed 19 April 2025].
- [52] Cytron Marketplace, "Pan Tilt Servo Kit for Camera (Unassembled)," Cytron, [Online]. Available: <https://my.cytron.io/p-pan-tilt-servo-kit-for-camera-unassembled>. [Accessed 14 April 2025].
- [53] Open Robotics, "ROS 2 Documentation — ROS 2 Documentation: Humble documentation," Open Robotics, [Online]. Available: <https://docs.ros.org/en/humble/index.html>. [Accessed 3 September 2025].
- [54] 2024 User Manual and Diagram Library, "User Manual and Diagram Library," 8 Nov 2024. [Online]. Available: <https://anakitgj3guidfix.z13.web.core.windows.net/raspberry-pi-gpio-diagram.html>.
- [55] ROBOTIS, "3. 3. OpenCR Setup," ROBOTIS, [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/. [Accessed 6 April 2025].
- [56] Cytron, "ReSpeaker 2-Microphone Raspberry Pi HAT," [Online]. Available: <https://my.cytron.io/p-respeaker-2-microphone-raspberry-pi-hat>. [Accessed 2025].
- [57] Cytron Marketplace, "Grove - Speaker," Cytron, [Online]. Available: <https://my.cytron.io/p-grove-speaker>. [Accessed 4 April 2025].
- [58] M. a. C. L. Copperwaite, Learning flask framework, Birmingham: Packt Publishing Ltd., 2015.

APPENDIX

User Opinion Survey for TurtleBot3 Companion Robot

A User Opinion Survey was conducted to gather opinions on the proposed features of the TurtleBot3 companion robot. Participants were shown screenshots and demonstrations of the Web UI, fall detection, STT/TTS interaction, meditation scheduling, and Telegram bot. They were asked to provide their feedback regarding the usability, expected responsiveness, and overall usefulness of each feature.

[Sign in to Google](#) to save your progress. [Learn more](#)

Age Group

☐ <18

☐ 18 - 30

☐ 31 - 50

☐ 51 - 65

☐ 66 +

Role/Background

☐ Caregiver

☐ Elderly User

☐ Researcher / Student

Next

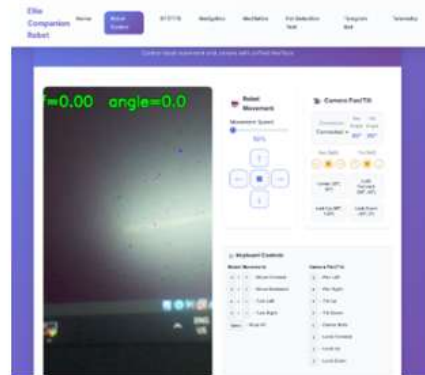
Clear form

User Opinion Survey for TurtleBot3 Companion Robot

Sign in to Google to save your progress. [Learn more](#)

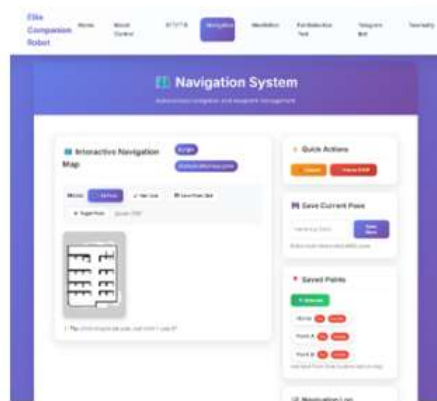
TurtleBot3 Companion Robot Features

How easy do you think it would be to use the Web UI to control the robot?



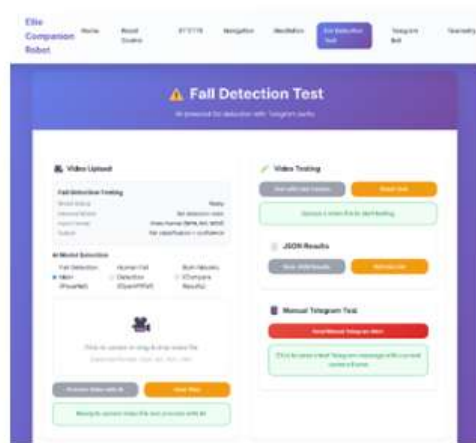
1 2 3 4 5
Very difficult ☐ ☐ ☐ ☐ ☐ Very easy

How confident are you that the robot could accurately navigate to waypoints based on the UI controls?



1 2 3 4 5
Not confident ☐ ☐ ☐ ☐ ☐ Very confident

How effective do you think the fall detection feature would be in identifying a person falling?



1 2 3 4 5
Not effective ☐ ☐ ☐ ☐ ☐ Very effective

How timely do you think the Telegram alerts for falls would be?

1 2 3 4 5
Too slow ☐ ☐ ☐ ☐ ☐ Very prompt

How accurate do you expect the speech recognition system (STT) to be?



1 2 3 4 5

Very inaccurate

☐
☐
☐
☐
☐

Very accurate

How natural and clear do you expect the robot's speech (TTS) to sound?

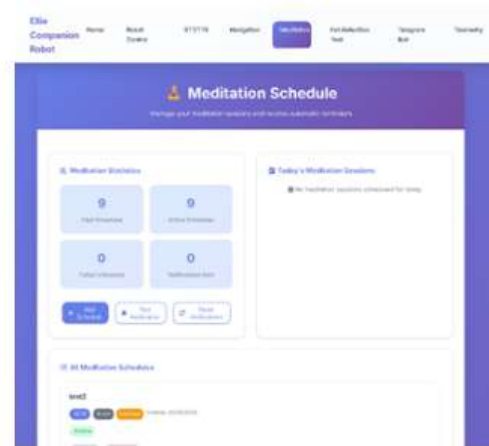
1 2 3 4 5

Poor

☐
☐
☐
☐
☐

Excellent

How useful do you think the meditation scheduling feature would be for elderly users?



1 2 3 4 5

Not useful

☐
☐
☐
☐
☐

Very useful

How easy do you think it would be to send commands and receive responses via the Telegram bot?



Very Difficult 1 2 3 4 5 Very Easy

☐ ☐ ☐ ☐ ☐

How accurate do you expect the bot's responses to be?

Very inaccurate 1 2 3 4 5 Very accurate

☐ ☐ ☐ ☐ ☐

How likely would you recommend this robot for use in elderly care?

Very unlikely 1 2 3 4 5 Very likely

☐ ☐ ☐ ☐ ☐

POSTER



FACULTY OF INFORMATION COMMUNICATION AND TECHNOLOGY

Smart Companion Robot for Elderly Care - Ellie

DEVELOPER: TAN KOK FU

SUPERVISOR: DR TEOH SHEN KHANG



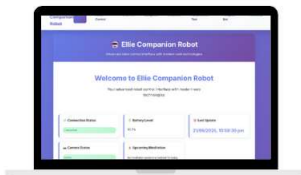
INTRODUCTION

As Malaysia's population ages, with over 10% expected to be over 65 soon, providing effective in-home support for seniors is a critical challenge. Unattended falls and social isolation are major risks.

This project presents 'Ellie,' a proactive, autonomous mobile companion robot designed to provide immediate assistance, intelligent monitoring, and companionship

RESULTS

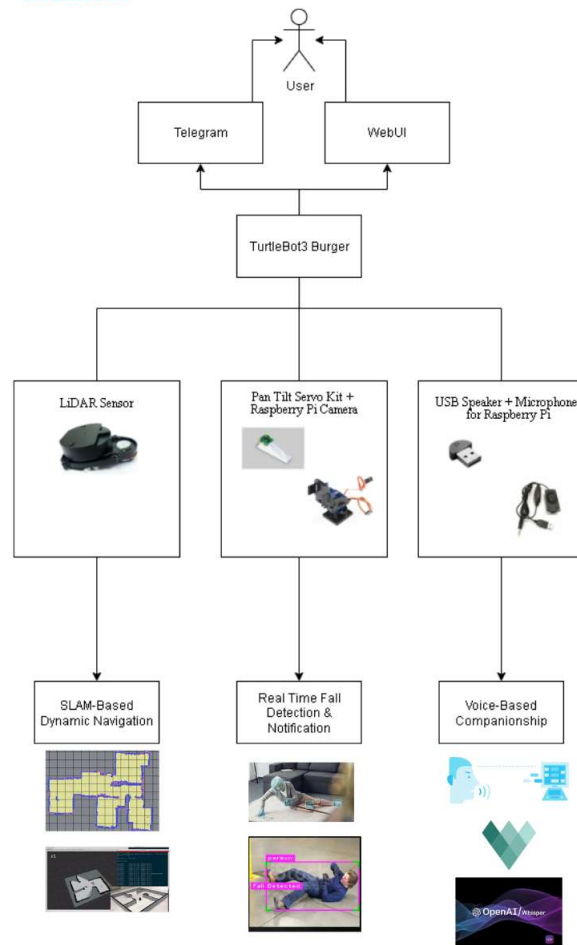
WebPage



Telegram



METHODS



CONCLUSION

This project develops an AI-powered companion robot using the TurtleBot3 Burger, integrating autonomous SLAM navigation, real-time fall detection and alerts, voice-based companionship (NLP/LLM), and a caregiver web interface with Telegram Bot to address elderly isolation and safety risk.

