Mitra: Autonomous Assistant Rover with Vision-Based Perception and Natural Language Interaction

A CAPSTONE PROJECT REPORT

Submitted in partial fulfilment of the requirement for the award of the Degree of

BACHELOR OF TECHNOLOGY
IN
ELECTRONICS AND COMMUNICATION ENGINEERING

by

Saurabh Patil (22BEC7136) *&*
Harshit Tiwari (22BEC7073)

Under the Guidance of

Dr. Ajay Dagar

SCHOOL OF ELECTRONICS ENGINEERING
VIT UNIVERSITY
VELLORE, TAMILNADU,INDIA

November 2025

CERTIFICATE

This is to certify that the Capstone Project work titled "**Mitra: Autonomous Assistant Rover with Vision-Based Perception and Natural Language Interaction**" that is being submitted by **Saurabh Hemant Patil (22BEC7136)** and **Harshit Tiwari (22BEC7073)** is in partial fulfilment of the requirements for the award of Bachelor of Technology, is a record of bonafide work done under my guidance. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted to any other Institute or University for award of any degree or diploma and the same is certified.

Dr. Ajay Dagar

Guide

The thesis is satisfactory / unsatisfactory

Internal Examiner-1

Internal Examiner-2

Approved by

PROGRAM CHAIR
B. Tech. ECE

DEAN
School of Electronics Engineering

ABSTRACT

Mitra is an autonomous assistant rover designed to unify real-time visual perception, contextual scene understanding, and natural language interaction into a single, resource-constrained robotic platform. The system integrates multi-modal sensing—including a monocular RGB camera, ultrasonic ranging, and environmental state monitoring—with an embedded computation stack centered on a Raspberry Pi 4 for high-level intelligence and an ESP32 microcontroller for low-level motor control and actuation. Visual perception is achieved through a lightweight object detection pipeline based on MobileNet-SSD, enabling Mitra to reliably recognize people, indoor objects, and navigational obstacles under strict computational limits. In parallel, a classical computer-vision lane-estimation module, combining region-of-interest filtering, Canny edge extraction, and Hough-based line fitting, provides continuous assessment of corridor alignment, navigable path geometry, and lateral displacement. Together, these perception modules generate structured scene descriptors that feed directly into the rover's decision-making and interaction layers.

A key contribution of this work is the seamless integration of these perception outputs with a natural-language reasoning module powered by a locally hosted large language model (LLM). Structured representations of the environment are transmitted to the companion LLM, which synthesizes high-level semantic descriptions, contextual explanations, and conversational responses. These are rendered through a high-quality text-to-speech (TTS) system, enabling Mitra to verbally describe its surroundings, comment on observed human activity, and engage in interactive dialogue. A robust interprocess communication architecture ensures low-latency coordination between the perception pipeline, lane-tracking server, narration engine, and a web-based operator interface. Implemented using Flask, the dashboard provides real-time visualization of video streams, object detections, lane geometry, sensor data, and LLM-generated responses, offering an intuitive and transparent channel for human–robot interaction.

The findings of this research demonstrate that an embedded platform with limited compute resources can effectively combine modern deep-learning perception, classical robotics algorithms, and natural language reasoning into a coherent autonomous system. Mitra is capable of perceiving, explaining, and communicating complex environmental information in real time, supporting both autonomous navigation and human-supervised operation. This work establishes a foundation for future advancements such as on-board SLAM, neural end-to-end navigation, multi-modal fusion, and fully edge-deployed LLM inference on low-power robotic platforms.

TABLE OF CONTENTS

S.No.	Chapter	Title	Page Number
1.		Acknowledgement	1
2.		Abstract	3
3.		List of Figures and Table	5
4.	1	Introduction	6
	1.1	Objectives	6
	1.2	Background and Literature Survey	7
	1.3	Organization of the Report	8
5.	2	Proposed System – MITRA Autonomous Assistant Rover	11
	2.1	System Architecture	11
	2.2	Working Methodology	12
	2.3	Standards / Design Constraints	13
	2.4	System Details	15
	2.4.1	Software	16
	2.4.2	Hardware	17
6.	3	Cost Analysis	27
	3.1	List of components and their cost	27
7.	4	Results and Discussion	31
8.	5	Conclusion & Future Works	33
9.	6	Appendix	35
10.	7	References	52

List of Tables

Table No.	Title	Page No.
Table 2.1	Hardware Components	2.4.2
Table 2.2	Software Components	2.4.1
Table 2.3	Model & Algorithms Used	2.4
Table 3.1	Cost Analysis	3

List of Figures

1. **Figure 1.1** System Architecture of the MITRA Autonomous Rover
2. **Figure 2.1** YOLOv3 Object Detection Pipeline Running on Raspberry Pi
3. **Figure 2.2** MediaPipe Pose-Based Interaction Detection between Multiple 5 People
4. **Figure 2.3** Lane Detection using Edge Detection and Hough Transform
5. **Figure 3.1** Hardware Block Diagram: Raspberry Pi, ESP32, Motors, Sensors
6. **Figure 4.1** Sample Output Showing Person Detection and Interaction Interpretation

CHAPTER 1 — INTRODUCTION

1. Introduction

Autonomous mobile systems that combine perceptual awareness with natural language interaction represent a rapidly maturing area of robotics research. The MITRA project constructs a compact, resource-constrained assistant rover that unifies real-time visual perception, basic navigation, and conversational description generation. The platform integrates embedded sensing (monocular RGB vision and ultrasonic proximity), edge computation (Raspberry Pi), microcontroller-level actuation (ESP32), and a locally hosted large language model for semantic reasoning. By bridging classical computer-vision methods and contemporary language models, MITRA seeks to demonstrate that a small, low-cost mobile robot can both perceive its surroundings and communicate meaningful, human-readable interpretations of observed scenes in real time.

This chapter situates the project, frames the technical problem, and outlines the aims and structure of the report. It establishes the motivation for combining lightweight on-device perception with off-board linguistic reasoning, emphasizes the constraints that guide design choices (compute, latency, privacy), and clarifies the contributions of the work: a working system architecture, a practical interaction-detection pipeline, and an integrated narration capability driven by a locally hosted LLM.

1.1 Objectives

The primary objective of this research is to design and develop Mitra, an intelligent autonomous assistant rover capable of perceiving, interpreting, and verbally communicating real-world environmental information in real time. Mitra aims to bridge the gap between classical robotics pipelines and modern language-driven reasoning systems by integrating multi-modal sensing, embedded computation, and natural language generation within a unified architecture. The system is conceived as a stepping stone toward highly interactive, context-aware companion robots capable of supporting human operators in indoor and semi-structured environments.

To fulfill this overarching goal, the following specific objectives are formulated:

1. To design an efficient perception pipeline for real-time object detection.

This objective focuses on enabling Mitra to visually recognize people, objects, and environmental cues under constrained computational resources. It involves deploying a lightweight convolutional neural network (MobileNet-SSD / YOLO-Tiny) on an edge processor, optimizing the model for low latency, and developing a detection interface that produces structured outputs (classes, bounding boxes, confidence

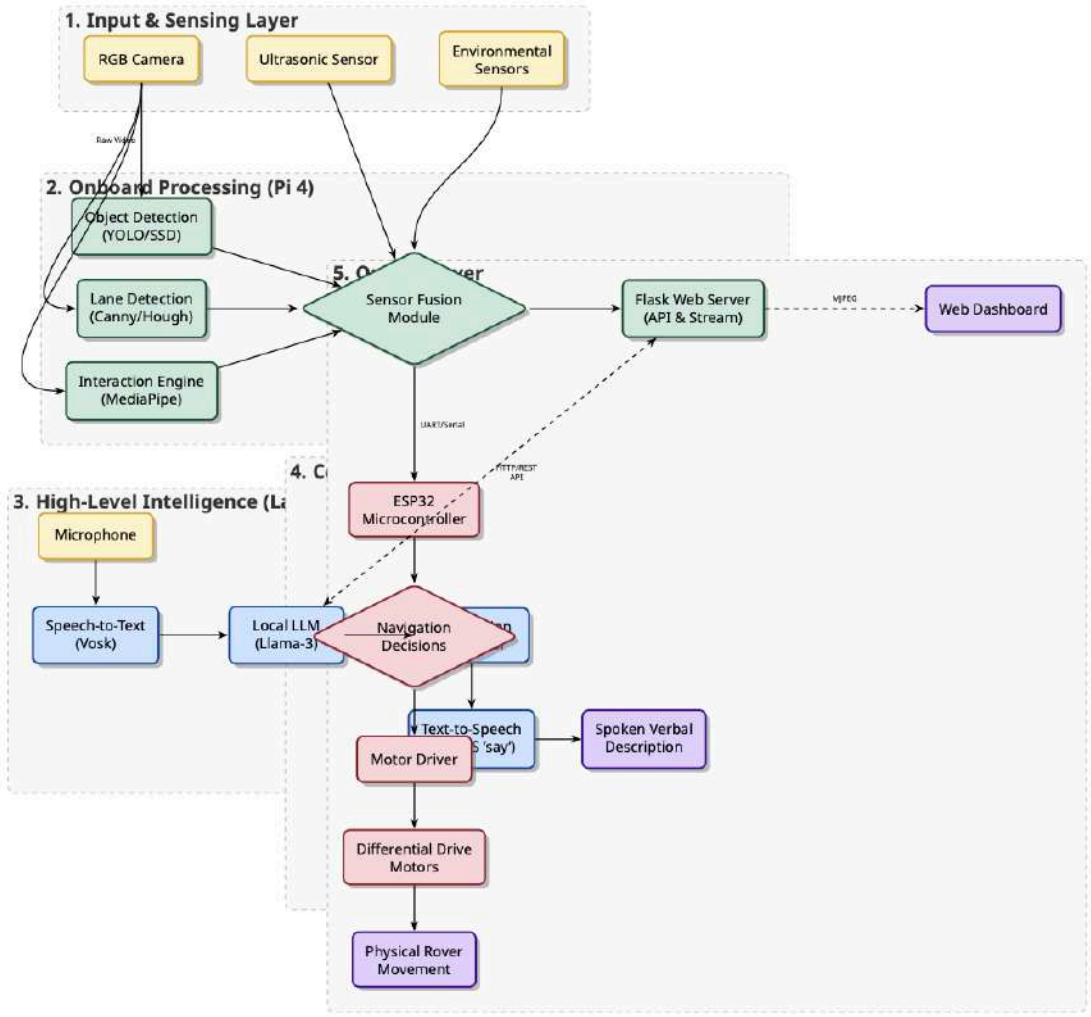
scores) suitable for downstream reasoning. Achieving reliable real-time object detection forms the perceptual foundation upon which higher-level scene understanding is built.

2. To implement a robust lane-tracking and navigational-awareness subsystem.

Autonomous mobility requires continuous assessment of traversable paths. This objective encompasses designing a classical computer vision pipeline involving region-of-interest isolation, color masking, Canny edge detection, and Hough line transformation to estimate lane boundaries. The subsystem must compute indicators such as lane angle, lateral offset, and lane detectability, providing actionable navigational cues for future autonomous control modules.

3. To integrate environmental and proximity sensing for contextual awareness.

Mitra must maintain awareness not only of visual entities but also of spatial proximity and local environmental conditions. This objective involves incorporating ultrasonic sensors, temperature/ambient sensors, and auxiliary modules, managed by the ESP32 microcontroller. The sensors provide complementary data enabling



collision avoidance, distance monitoring, and contextual interpretation, which ultimately enhance the reliability and safety of the rover's operations.

4. To develop a natural-language reasoning layer using a Local Large Language Model (LLM).

A distinguishing objective of this research is the creation of a semantic interpretation engine driven by a locally hosted LLM (such as Llama 3). The goal is to translate raw numerical perception data into coherent human-readable descriptions, interactions, and explanations. This requires designing a structured communication protocol between the rover and the reasoning server, crafting semantic templates, and implementing prompt-engineering techniques that enable the LLM to infer context, behavior, and potential interactions from scene data.

5. To enable natural human–robot interaction through speech recognition and TTS.

This objective establishes Mitra as an interactive assistant rather than a purely autonomous machine. A wake-word–driven interface, powered by Vosk speech recognition, enables hands-free activation. Text-to-speech pipelines allow Mitra to deliver articulate verbal responses generated by the LLM. Together, these modules create an intuitive conversational interface that supports continuous natural interaction between humans and the rover.

6. To construct a unified communication and control framework.

Given the distributed nature of Mitra's architecture—comprising Raspberry Pi vision modules, ESP32 motor controllers, a lane estimation server, and an LLM reasoning engine—a reliable communication layer is necessary for synchronization among modules. This objective involves designing inter-process and inter-device communication protocols using Flask servers, serial communication, and structured JSON messaging. The goal is to ensure deterministic, low-latency data exchange across subsystems.

7. To design a real-time monitoring and control dashboard.

A browser-based operator dashboard must visualize the rover's perception and reasoning. This involves designing a Flask web interface capable of streaming video, overlaying detection boxes, displaying lane metrics, showing sensor data, and rendering LLM-generated insights. The dashboard serves both as a research platform and an operational control panel, providing transparency into the system's internal processes.

8. To evaluate the system's performance and identify opportunities for future enhancements.

The final objective focuses on systematically assessing the system's efficiency, latency, accuracy, and interaction quality. The evaluation serves not only to validate the feasibility of integrating embedded perception with LLM-driven reasoning but also to highlight potential improvements such as on-board SLAM integration, end-to-end neural navigation, and eventual deployment of edge-optimized language models.

1.2 Background and Literature Survey

Research on assistive and social robotics sits at the intersection of perception, cognition, and human–robot interaction. Traditional mobile robots prioritized navigation and obstacle avoidance using geometric methods and range sensors; more recent trends incorporate deep learning for semantic perception (object detection, segmentation) and transformer-based language models for higher-level reasoning. Two broad streams of prior work inform MITRA: (i) edge perception efforts that adapt lightweight neural networks (e.g., MobileNet, Tiny-YOLO) for real-time inference on single-board computers, and (ii) multimodal interaction research that combines vision, audio, and language to produce context-aware responses.

Object detection on resource-constrained platforms has been extensively studied; models such as MobileNet-SSD and optimized YOLO variants are widely used because they provide a favorable trade-off between speed and accuracy. Complementary, classical image-processing techniques (Canny edges, Hough transforms) remain effective for geometric tasks like lane estimation where interpretability and low latency are important. For human interaction recognition, approaches range from simple geometric heuristics (centroid proximity, bounding-box overlap) to richer pose-based analysis (keypoint detection, face orientation) and temporal modeling. Tools such as MediaPipe provide fast landmark detection that is especially useful when on-device compute is limited.

On the language side, transformer-based LLMs have opened new possibilities for translating structured perception data into natural language summaries and recommendations. However, reliance on cloud services raises concerns about latency, privacy, and availability; the recent emergence of locally hostable LLM frameworks permits offline reasoning while retaining high-quality language generation. Integration of perception and LLM reasoning remains an active research area, with challenges around representation (how best to structure sensory outputs for the language model), latency management, and grounding (ensuring model outputs accurately reflect sensory evidence).

MITRA builds on these strands by adopting efficient perception algorithms on the edge, leveraging pose and face-mesh analytics for interaction confirmation, and routing compact structured observations to a locally hosted LLM. The project situates

itself among practical demonstrations rather than purely theoretical contributions: the emphasis is on a reproducible, low-cost system that demonstrates the utility of combining lightweight perception with local language reasoning for real-time human-robot interaction.

1.3 Organisation of the Report

The remainder of this thesis is organised as follows:

Chapter 2 — System Design and Methodology: Describes the overall architecture, hardware selection, software stack, and the algorithms implemented for object detection, interaction recognition, lane estimation, and speech processing. This chapter details the communication protocol between the Raspberry Pi, ESP32, and the companion machine hosting the LLM, and includes design rationales for computational trade-offs.

Chapter 3 — Implementation: Provides implementation specifics, code organisation, integration steps, and configuration details needed to reproduce the system. It also contains the cost breakdown, assembly instructions, and the description of the Flask dashboard used for monitoring and control.

Chapter 4 — Experiments and Results: Reports quantitative and qualitative evaluations including detection latency and frame rates, interaction detection case studies, LLM response behaviour, and end-to-end demo transcripts. This chapter analyses strengths, limitations, and failure modes under varying lighting and crowding conditions.

Chapter 5 — Conclusion and Future Work: Summarizes the project contributions, reflects on lessons learned, and outlines extensions such as on-board SLAM, improved temporal models for sustained interaction recognition, and migration of LLM inference to embedded accelerators.

Appendices contain supplementary material: configuration files, detailed wiring diagrams, complete API specifications, and sample logs used during evaluation.

CHAPTER 2

PROPOSED SYSTEM – MITRA AUTONOMOUS ASSISTANT ROVER

This Chapter describes the proposed system, working methodology, software and hardware details.

2.1 System Architecture

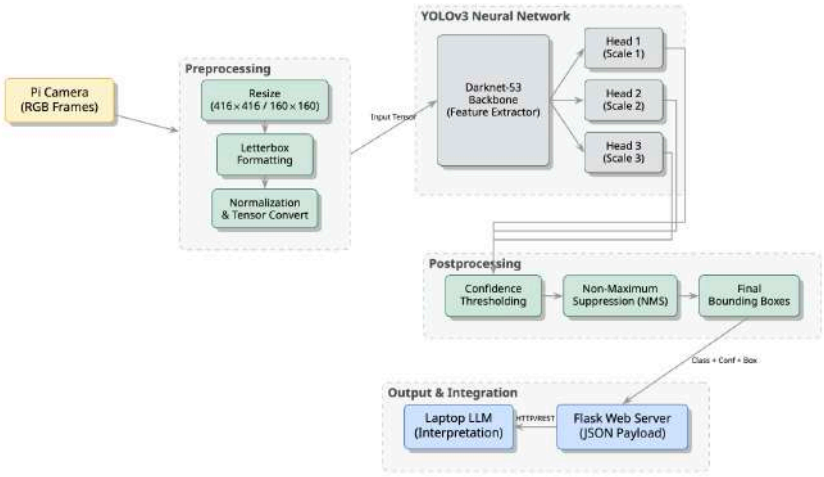
2.1.1 Overview

The MITRA system is designed as a modular, distributed architecture that separates perception, reasoning, and actuation while maintaining low-latency communication channels among components. The architecture optimises for (a) real-time visual perception on an embedded host (Raspberry Pi 4), (b) deterministic low-level motor control via an ESP32 microcontroller, and (c) high-level semantic reasoning on a companion machine hosting a local LLM. This separation enables computationally expensive language inference to run off-board while preserving immediate reactiveness for safety-critical tasks.

2.1.2 Functional Blocks (high level)

- Vision & Perception (Raspberry Pi 4): camera input → preprocessing → object detector (YOLO/MobileNet-SSD) → interaction heuristic + MediaPipe analysis → lane estimation.
- Speech Interface (Raspberry Pi 4): always-on wake-word detection (Vosk) → speech-to-text for user queries.
- Reasoning (Companion Machine): structured scene JSON → local LLM (Ollama + Llama-3) → concise textual response.
- Control & Actuation (ESP32): receives high-level motion commands (serial/ Wi-Fi) → drives motor driver (L298N/TB6612) → motors.
- Dashboard & API (Flask on Pi): multi-endpoint REST/streaming API exposing `/video_feed`, `/latest_detections`, `/latest_frame` and dashboard UI.

2.1.3 Block diagram



2.2 Working Methodology

This subsection describes the operational pipeline and the run-time interactions across modules.

2.2.1 Perception pipeline

Image acquisition: The camera captures monocular RGB frames at a specified resolution (typ. 640×480 or 1280×720). Frames are buffered in a small queue to avoid blocking the detector.

Preprocessing: Frames are letterboxed/resized to the detector's input dimension (e.g., 416×416), normalized, and converted to the network's tensor format.

Object detection: A lightweight CNN (YOLO variant or MobileNet-SSD) runs inference producing bounding boxes, class IDs, and confidence scores.

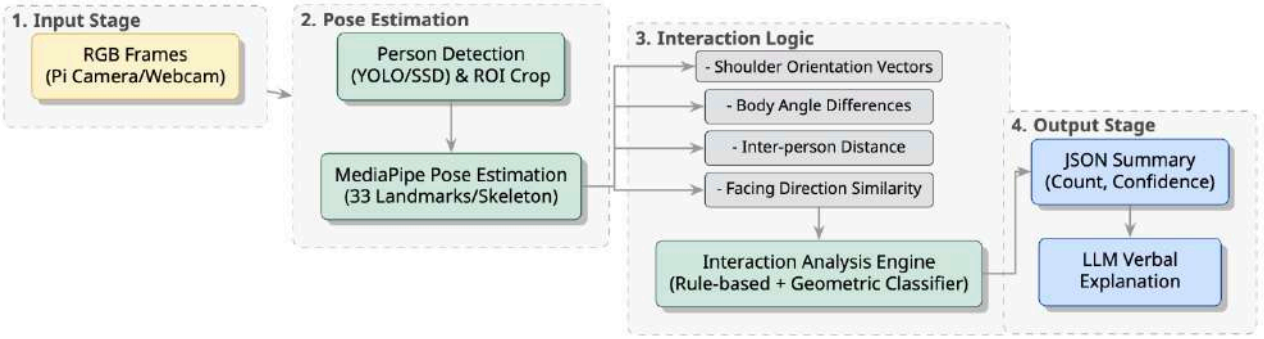
Post-processing: Non-max suppression (NMS) and thresholding produce final detections; coordinates are converted back to image pixel space.

Interaction heuristic: Geometric rules compute centroid distances, normalized inter-person distances, and horizontal alignment proxies. Candidate interacting pairs are identified.

MediaPipe confirmation (optional/advanced): For candidate pairs, a MediaPipe FaceMesh/Pose pass estimates head yaw and mouth openness to confirm "talking" with higher confidence.

Lane estimation: A lightweight classical pipeline extracts a region of interest (ROI), applies grayscale → Gaussian blur → Canny edge detection → Hough line transform, and computes lane slope and lateral offset.

Sensor fusion: Ultrasonic readings and other environmental sensors are fused with vision outputs to improve obstacle-awareness and safe stopping.



MediaPipe Pose-Based Interaction Detection Between Multiple People for the MITRA

2.2.2 Interaction and narration flow

On wake-word detection ("MITRA"), the system samples the latest scene summary from /latest_detections.

The interaction module summarizes persons, interacting pairs, lane status, and any safety alerts into a concise JSON payload.

The payload is sent to the local LLM with a prompt template that constrains tone (PA-style), length (one sentence), and actionable recommendation (one short action).

The LLM response is returned as text and then rendered via TTS; simultaneously the response and raw detection data are logged and displayed on the dashboard.

2.2.3 Control loop

The perception stack runs continuously; the ESP32 receives discrete movement commands from the Pi (e.g., {"cmd":"forward","speed":60}) and handles PWM and motor driver control deterministically.

Safety checks (ultrasonic threshold breach, unknown obstacles) are enforced at both the Pi level (pre-command veto) and the ESP32 level (emergency stop).

The lane estimator can optionally drive low-level course corrections via frequent steering commands or publish guidance values for higher-level planners.

2.3 Standards / Design Constraints

This section enumerates the standards and constraints that guided the design.

2.3.1 Real-time and latency constraints

Perception latency: Object detection should operate at interactive frame rates (target ≥ 8–15 FPS on Raspberry Pi depending on model/optimization). MediaPipe confirmation can be invoked selectively to avoid frame-rate degradation.

End-to-end response time: Wake → detection → LLM-response → TTS should complete within a user-acceptable window (preferable < 5 seconds for short responses). Local LLM inference is on a companion host to meet this constraint.

2.3.2 Compute & memory constraints

Raspberry Pi CPU and available RAM limit model size; network architectures and input resolutions are chosen for tractable memory footprints. GPU/Neural acceleration (e.g., NPU, Coral) is optional but not required in the baseline.

2.3.3 Power & electrical constraints

Motors and motor drivers require a separate regulated supply; communication grounds must be common. Current draw estimates for motors inform battery capacity decisions to sustain typical demo durations.

2.3.4 Communication & protocol standards

REST/HTTP for dashboard and inter-process scene queries (/latest_detections, / latest_frame).

Serial (UART) or lightweight JSON over TCP/UDP for Pi ↔ ESP32 commands.

JSON schema for scene messages is defined to ensure deterministic parsing by the LLM and dashboard.

2.3.5 Safety and fail-safe design

Emergency stop implemented both software-side (Pi) and hardware-side (ESP32 checks).

Watchdog timers on ESP32 to prevent runaway motors.

Privacy: All language reasoning is local (no cloud) to avoid transmitting potentially sensitive camera data externally.

2.3.6 Development & reproducibility standards

Version control (git) for code.

Environment specification via requirements.txt or Pipfile.

Documented reproducible steps to replicate model loading, weights placement, and server invocation.

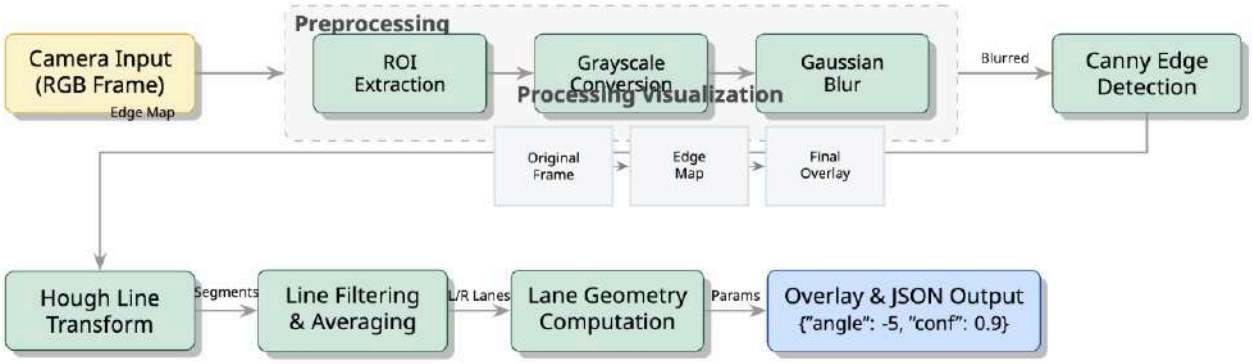


Figure 2.3 — Lane Detection using Edge Detection and Hough Transform" for the MITRA

2.4 System Details

This section documents concrete implementations and component roles. It is divided into software and hardware details.

2.4.1 Software

2.4.1.1 Modules & Responsibilities

Perception Module (Python): Implements frame capture, preprocessing, model inference (YOLO/MobileNet), NMS, and detection publishing. Exposes / latest_detections and /video_feed.

Interaction Module (Python): Runs geometric heuristics for proximity and face-to-face proxy; optional MediaPipe-based face/pose refinement.

Lane Estimation Module (Python/OpenCV): ROI selection, Canny, Hough transform, and computation of lane angle & offset.

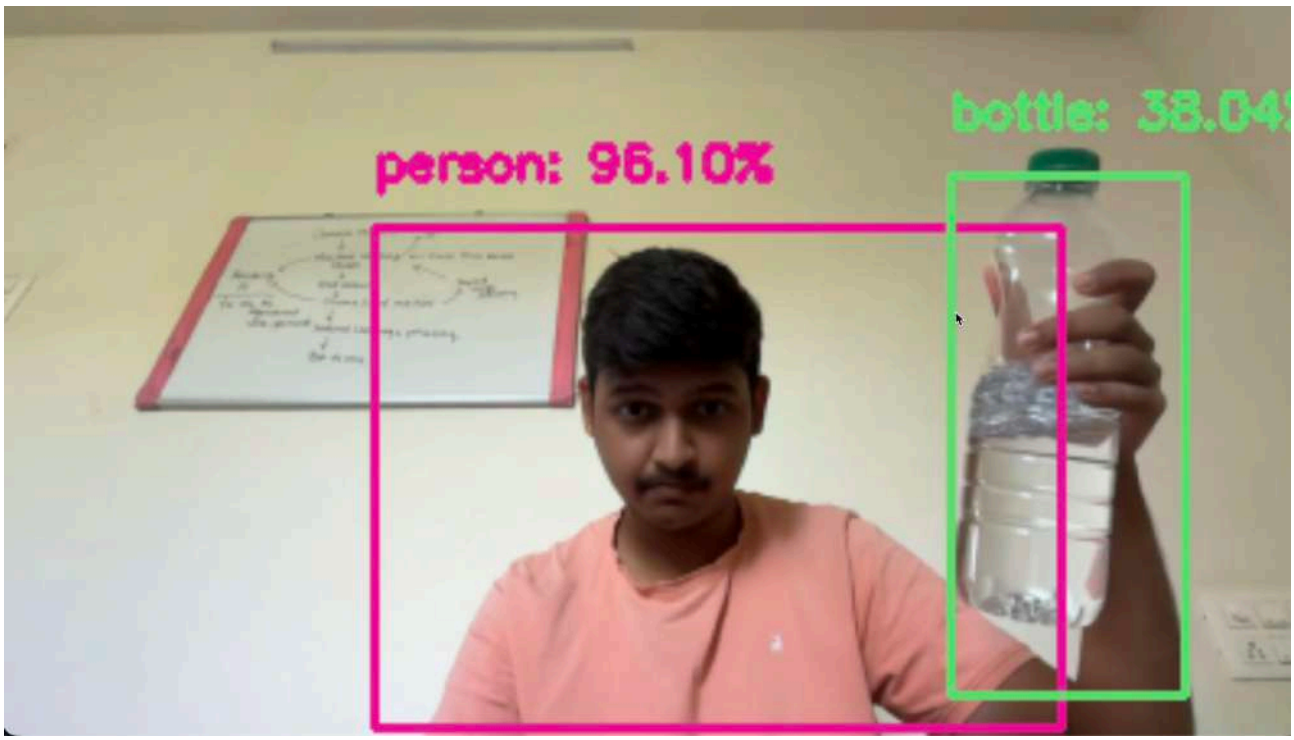
Speech Module (Python + Vosk): Continuous wake-word listener; on activation, passes audio to Vosk ASR and triggers on_wake().

LLM Integration (Ollama client / Requests): Formats scene JSON into a persona-constrained prompt, calls the local Ollama REST or CLI, and retrieves textual response.

TTS Module: Uses say (macOS) or another TTS library to render LLM text.

Flask Dashboard: Renders live video, detection overlays, lane metrics, sensor readings, and assistant transcript. Exposes operational endpoints for manual commands.

Comm. Bridge (Serial/TCP): Serializes command messages to ESP32 and monitors acknowledgements.



A person with bottle is detected with the accuracy of 96%

```
WAKE WORD DETECTED
Local detection summary:  2 person
interaction detected
Ollama answer:  ...two individuals are
engaged in conversation. Their facial
orientations suggest mutual attention, and
approximate speaking motions are bserved.
Recommended action: Maintain a considerate
distance and refrain from interrupting
Transcript: Yes, I'm listening
Transcript: Can you describe what you see?
Transcript: Sure
```

Terminal output when wake word "Mitra is detected"

2.4.2 Hardware

The hardware architecture of the MITRA Autonomous Assistant Rover has been carefully designed to support real-time perception, reliable locomotion, and robust interaction capabilities while operating within the constraints of an embedded robotic platform. At the core of the system lies the **Raspberry Pi 4**, which functions as the primary computational unit responsible for processing visual data, executing perception algorithms, hosting the dashboard server, and managing the speech interface. Its quad-core CPU, along with sufficient RAM, enables the execution of lightweight deep-learning models and classical computer-vision pipelines, making it a suitable choice for mobile robotic platforms where compactness and energy efficiency are important.

i) Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer as shown in Figure 17 that plugs into a computer monitor or TV and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.

Figure 17 Raspberry Pi

The device uses the ARM processor which does nearly all of the hard work in order to run the Raspberry Pi.

The main aim of Raspberry Pi is to collect the data from various nodes i.e. Arduino and upload them to the firebase database.

Installation of Raspbian OS

1. Format the SD card- Anything that's stored on the SD card will be overwritten during formatting. So if the SD card on which you want to install Raspbian currently has any files on it, e.g. from an older version of Raspbian, you may wish to back these files up first to not lose them permanently.
2. New Out of Box Software (NOOBS)- Using the NOOBS software is the easiest way to install Raspbian on your SD card.
    - Download NOOBS- Visit the Raspberry Pi downloads page.
    - Extract NOOBS from the zip archive
    - Next, you will need to extract the files from the NOOBS zip archive you downloaded from the Raspberry Pi website.
    - Double-click on it to extract the files, and keep the resulting Explorer/Finder window open.
    - Select all the files in the NOOBS folder and drag them into the SD card window to copy them to the card.

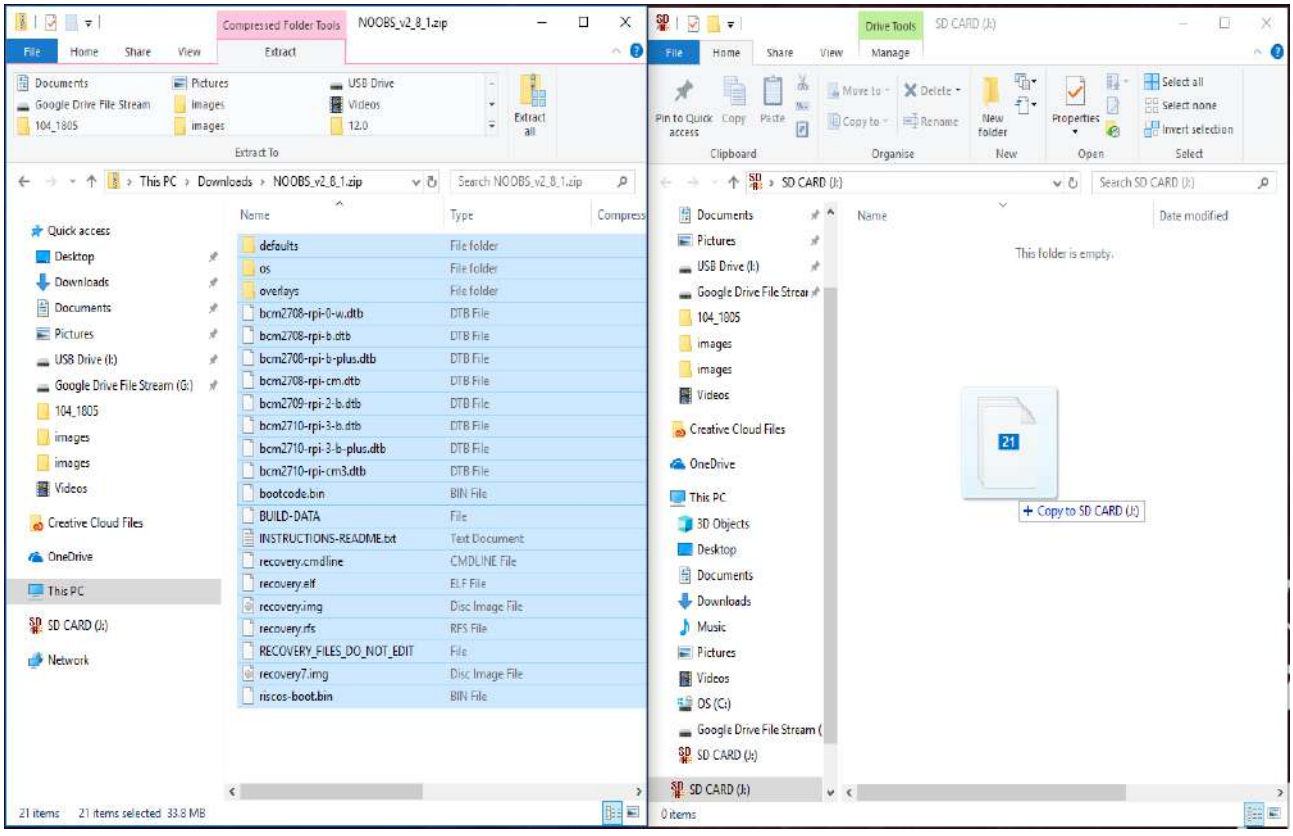


Figure 18 Copying to Sim Card

3. Connect your Raspberry Pi

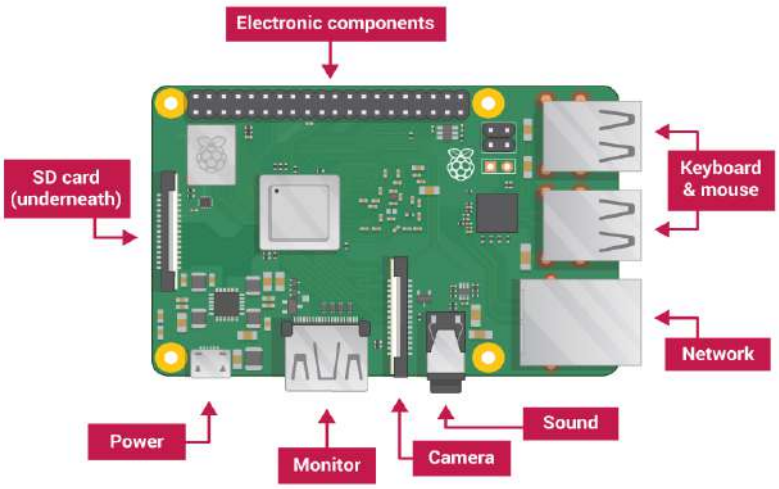


Figure 19 Raspberry Pi

- Insert the SD card with Raspbian (via NOOBS) into the micro SD card slot
- Connect the mouse and the keyboard to a USB port on the Raspberry Pi
- Use a cable to connect a screen to the Pi's HDMI port — use an adapter if necessary.
- Startup Raspberry Pi
- Connect a micro USB power supply it to you Pi's power port.

The red LED light up on the Raspberry Pi, indicates that the Pi is connected to power. As it boots up raspberries will appear in the top left-hand of the screen.

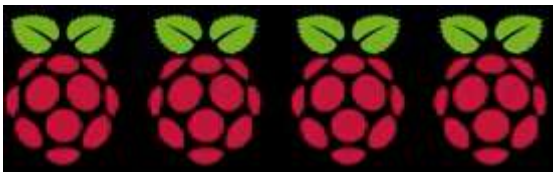


Figure 20 Powering Raspberry Pi

After a few seconds the Raspbian Desktop will appear.

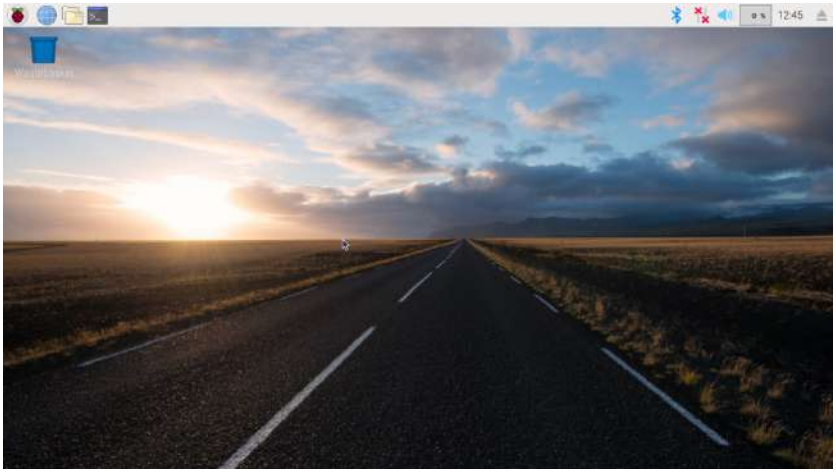


Figure 21 Raspbian Desktop

4. Finish the setup

When you start your Raspberry Pi for the first time, the Welcome to Raspberry Pi application will pop up and guide you through the initial setup.

Figure 22 Raspberry Pi Welcome Screen

- Click Next to start the setup.
- Click Done or Reboot to finish the setup.

Sometimes it is not convenient to work directly on the Raspberry Pi as sometimes we need to work on it from another device by remote control .VNC is a graphical desktop sharing system

that allows us to remotely control the desktop interface of one computer (running VNC Server) from another computer or mobile device (running VNC Viewer). VNC Viewer transmits the keyboard and either mouse or touch events to VNC Server and receives updates to the screen in return.

VNC viewer and VNC server

VNC is a graphical desktop sharing system that allows one to remotely control the desktop interface of one computer (running VNC Server) from another computer or mobile device (running VNC Viewer). VNC Viewer transmits the keyboard and either mouse or touch events to VNC Server and receives updates to the screen in return.

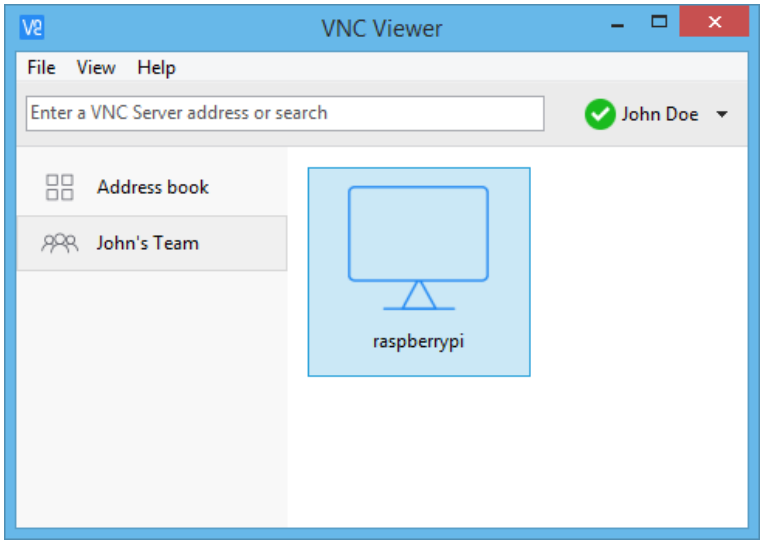


Figure 23 VNC Viewer

VNC Connect from Real-VNC is included with Raspbian. It consists of both VNC Server, which allows us to control your Raspberry Pi remotely, and VNC Viewer, which allows us to control desktop computers remotely from our Raspberry Pi.

1. Enabling VNC Server

On the Raspberry Pi, run the following commands to make sure the latest version of VNC is installed. Follow the following steps:

In the Terminal enter these commands

- sudo apt-get update
- sudo apt-get install realvnc-vnc-server realvnc-vnc-viewer

Now enable VNC Server. You can do this graphically or at the command line.

- Enabling VNC Server graphically
  - On your Raspberry Pi, boot into the graphical desktop.
  - Select **Menu > Preferences > Raspberry Pi Configuration > Interfaces**.
  - Ensure **VNC** is **Enabled**.
- Enabling VNC Server at the command line
  - You can enable VNC Server at the command line
  - sudo raspi-config
  - Now, enable VNC Server by doing the following:
    - Navigate to **Interfacing Options**.
    - Scroll down and select **VNC > Yes**.

2. Connecting to your Raspberry Pi with VNC Viewer

There are two ways to connect to your Raspberry Pi.

- Establishing a direct connection-Direct connection is quick and simple providing you're joined to the same private local network as your Raspberry Pi. For example, this might be a wired or wireless network at home, at school, or in the office).
  - On your Raspberry Pi (using a terminal window or via SSH) use these instructions or run ifconfig to discover your private IP address.
  - On the device you'll use to take control, download VNC Viewer. For best results, use the compatible app from RealVNC.
  - Enter your Raspberry Pi's private IP address into VNC Viewer:

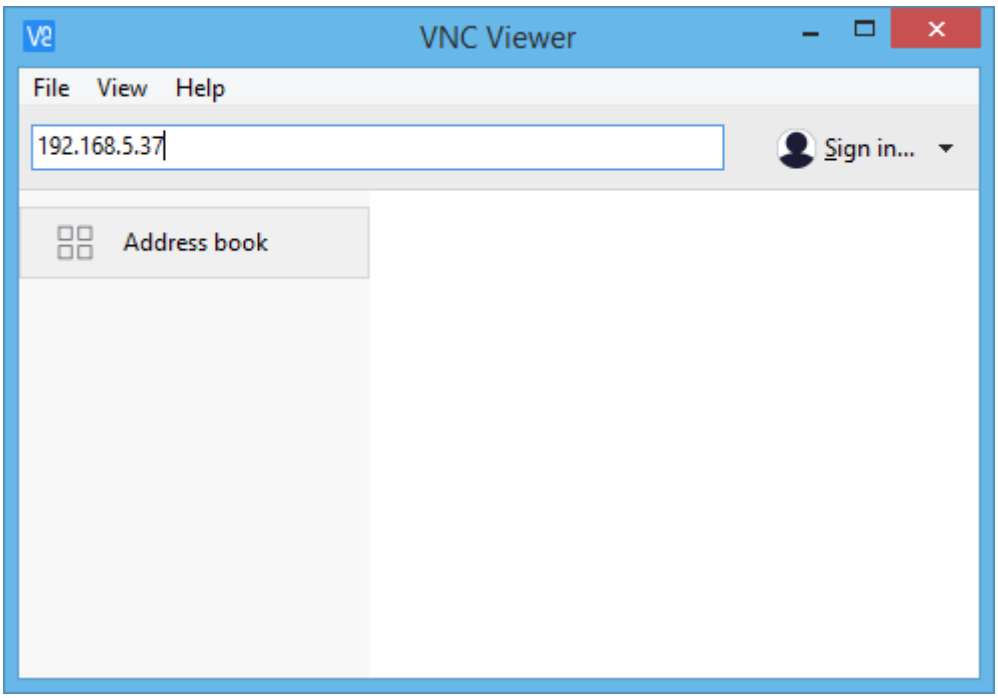


Figure 24 VNC Viewer Address

- Establishing a cloud connection- Cloud connections are convenient and encrypted end-to-end. They are highly recommended for connecting to your Raspberry Pi over the internet.
  - On your Raspberry Pi, sign in to VNC Server using your new RealVNC account credentials:

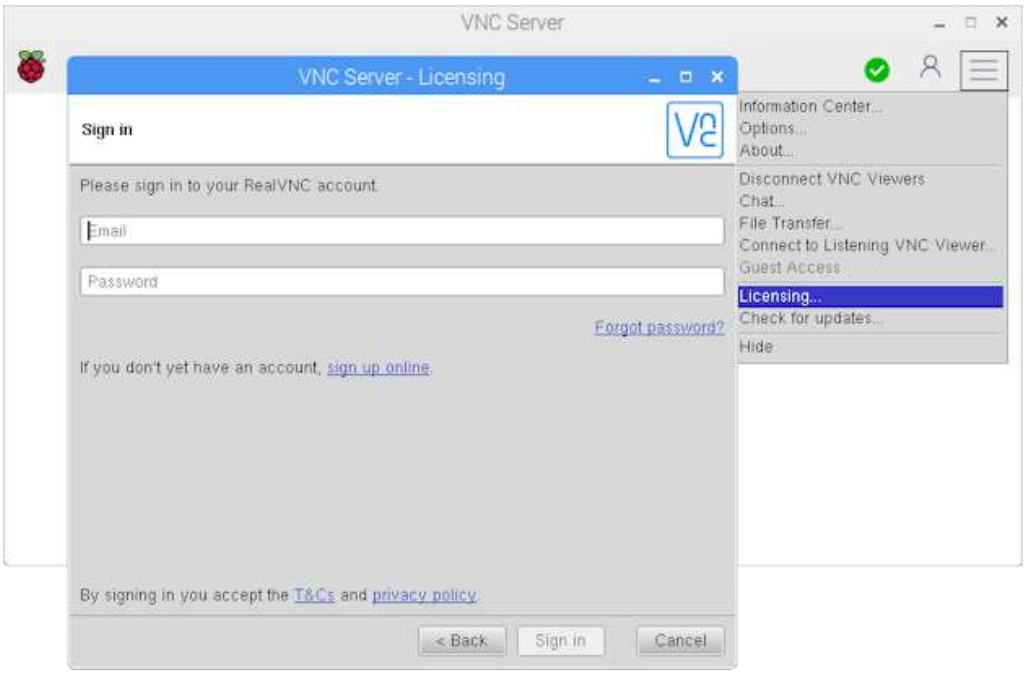


Figure 25 VNC Server

  - On the device you'll use to take control, download VNC Viewer.

- Sign in to VNC Viewer using the same RealVNC account credentials, and then connect to your Raspberry Pi or set up a new connection:

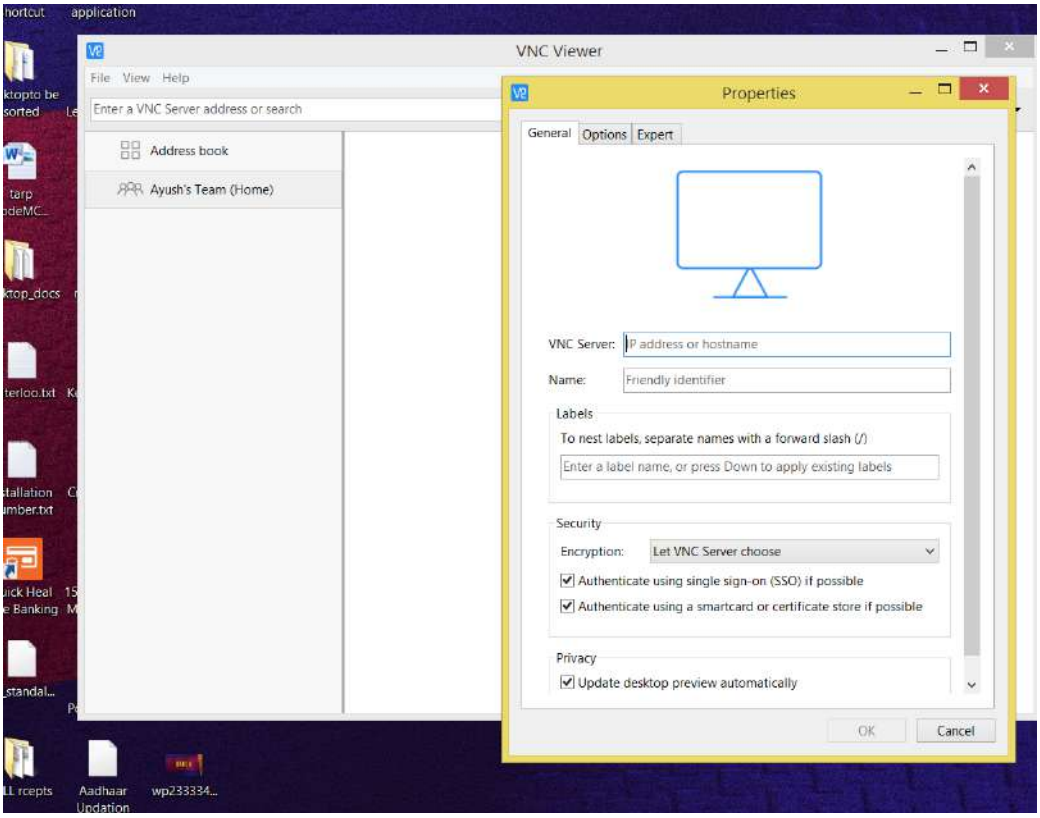


Figure 26 VNC Properties Window

- NOTE: IP address here is Internet IP address which is different from Ethernet IP (LAN) address as both Ethernet port and built in WIFI modules have different MAC addresses

To complement the Raspberry Pi, the rover incorporates an **ESP32 microcontroller**, which handles all low-level control and actuation tasks. Delegating motor-control responsibilities to the ESP32 ensures deterministic timing for PWM generation, direction switching, and speed regulation—tasks that are time-critical and unsuitable for a multitasking Linux system. The ESP32 also interfaces directly with onboard proximity sensors, such as the **HC-SR04 ultrasonic sensor**, which continuously monitors the environment for obstacles and enables microsecond-level response for emergency stops. The division of responsibilities between the Pi and the ESP32 creates a layered control hierarchy in which high-level perception-driven decisions are translated into fast, stable motor commands.

For visual sensing, the rover deploys a **monocular RGB camera**, either a USB webcam or the Raspberry Pi Camera Module, mounted to provide an optimal field of

view encompassing both human subjects and the immediate navigational path. This single camera supports all major perception tasks, including object detection, interaction assessment, pose estimation, and lane detection. Its placement and angle are deliberately chosen to minimize occlusions while stabilizing the frame during forward motion.

Locomotion is achieved through a pair of **DC gear motors** configured in a differential-drive layout. These motors are interfaced via a motor driver such as the **L298N** or **TB6612FNG**, which acts as the power interface between the ESP32 and the high-current motor supply. The motor driver ensures safe switching, current regulation, and bidirectional control, while maintaining electrical isolation between sensitive logic components and the motor subsystem. The motors are powered from a dedicated battery pack—typically 7.4 V or 12 V depending on driver specifications—while the Raspberry Pi and ESP32 operate from a regulated 5 V supply. A common electrical ground is maintained across all digital subsystems to ensure signal integrity.

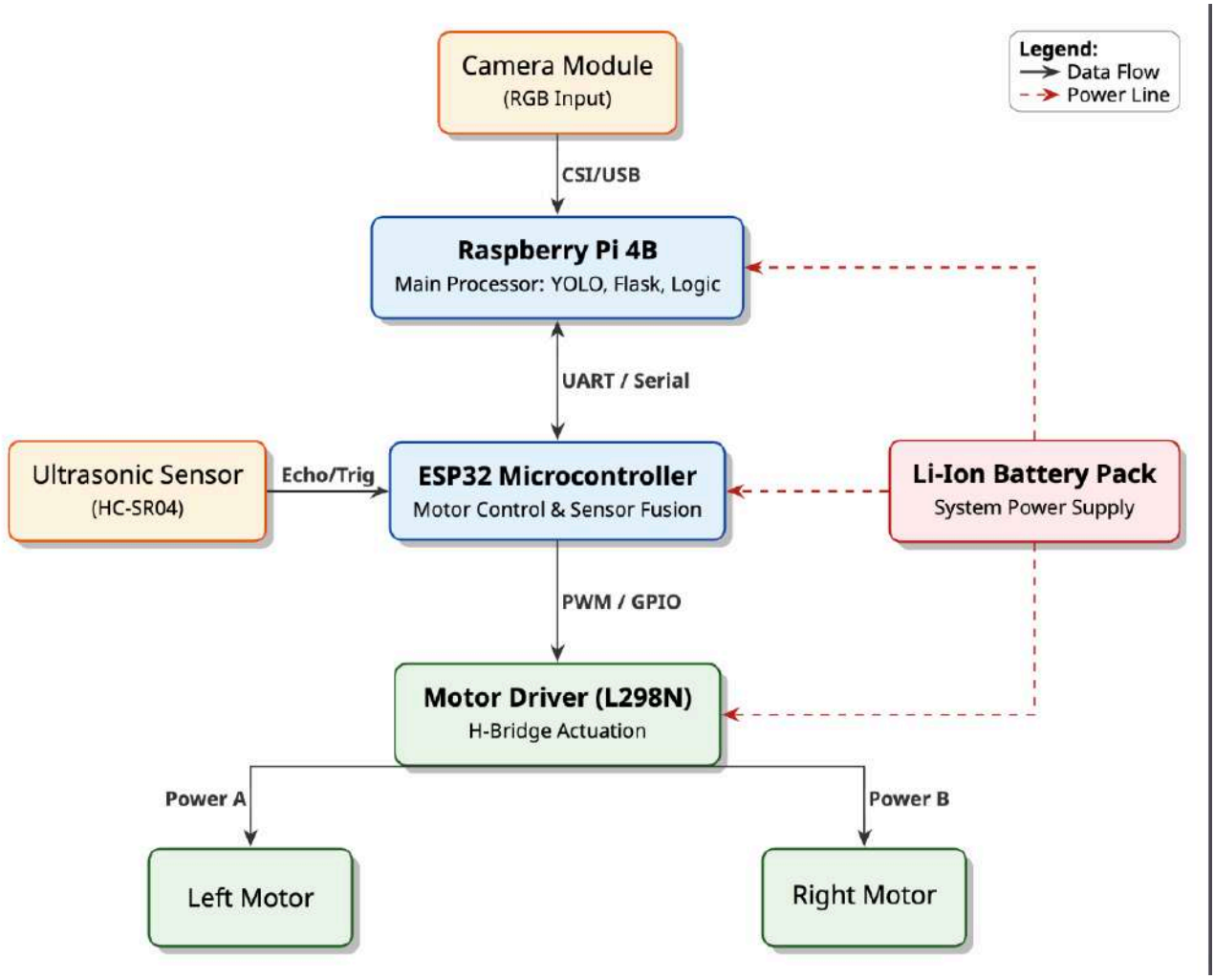
The overall mechanical assembly is built on a rigid rover chassis housing the electronics, wiring harness, and battery compartments. Care has been taken to optimize weight distribution to reduce vibration-induced noise in camera frames and to improve traction during acceleration and turning. Cable routing and power distribution are organized to prevent electromagnetic interference from the motors affecting sensor readings or microcontroller operation. Essential safety mechanisms, including fuses, voltage regulators, and optional hardware kill switches, are incorporated to enhance electrical reliability and protect against transient overloads.

In summary, the hardware subsystem of MITRA embodies a deliberate balance between computational capability, control stability, sensing fidelity, and electrical safety. The integration of the Raspberry Pi for high-level processing, the ESP32 for precise actuation, and a comprehensive sensing suite establishes a robust physical foundation upon which the advanced perception and interaction features of the rover are built.

Cost Analysis

3.1 List of Components and Their Cost

This section presents a comprehensive cost analysis for the MITRA Autonomous Assistant Rover. The cost estimates are provided to document the resources required to build and operate a single prototype unit. Prices reported are indicative retail estimates appropriate for academic procurement at the time of writing and are expressed in Indian Rupees (INR). Where useful, brief justifications for component selection and remarks on procurement or alternatives are included. All totals assume purchase of one unit of each listed component unless noted otherwise.



3.1.1 Basis and assumptions

The cost figures in the table below reflect typical online retail or local electronics-market prices for hobbyist and maker-grade components. The analysis assumes that peripheral lab infrastructure (workbench, oscilloscope, soldering station) and development tools (IDE, general-purpose computing equipment) are already available and therefore excluded. Shipping, taxes (GST), and import duties are not included and will vary by supplier; budget an additional 8–18% for those overheads in practice. Prices are rounded to the nearest convenient unit to keep the budget practical for academic planning.

S. No	Component	Specification / Typical Vendor	Quantity	Unit Cost (INR)	Remarks / Purpose	Total (INR)
1	Raspberry Pi 4 Model B	4GB RAM variant (official distributor)	1	6000	Main onboard computer for perception & server	6000
2	ESP32 Development Board	ESP32-WROOM-32 (DevKitC / DOIT)	1	350	Low-level motor control, sensor I/O, watchdog	350
3	USB Camera / Pi Camera Module	1080p UVC webcam / Pi Camera V2	1	900	Real-time visual sensor for detection	900
4	Ultrasonic Sensor (HC-SR04)	HC-SR04 or SRF05 equivalent	2	120	Obstacle detection & proximity measurement	240
5	Motor Driver (L298N / TB6612FNG)	Dual H-bridge driver	1	300	Power interface for differential motors	300
6	DC Geared Motors	12V DC geared motor (300 RPM)	2	250	Drivetrain torque & controlled speed	500
7	Chassis and Wheels	Metal/plastic kit with mounting brackets	1 set	900	Mechanical base & mounting	900
8	Battery Pack & Power Regulation	12V Li-ion/LiPo + 5V regulator	1	1500	Motor power + regulated logic supply	1500
9	SD Card (32GB Class 10)	For Raspberry Pi OS & storage	1	350	OS & dataset storage	350
10	USB Microphone / Audio Interface	Omni-directional USB mic	1	450	Audio capture for ASR	450
11	Miscellaneous Electronics	Wires, connectors, PCB, fuses	—	500	Wiring, prototyping, safety components	500
12	Mounting Hardware & Enclosures	Screws, standoffs, brackets, enclosure	—	300	Securing electronics & assemblies	300
13	Heat Sinks / Cooling (optional)	Passive heatsinks	—	150	Thermal reliability	150

14	Motor Encoders (optional)	Wheel encoders / hall sensors	2	600	Odometry & closed-loop control	1200
15	USB–Serial Adapter / Level Shifter	TTL adapter, logic shifters	1	250	Debugging & voltage-level conversion	250
16	Spare Parts & Prototyping Allowance	Extra motors, wires, prototyping materials	—	700	Contingency costs	700
	Subtotal (baseline, excluding optional)					**11,390**
	Optional Items Total				(Encoders + Cooling)	1,450
	Total Estimated Cost (including optional)					12840 /-

3.1.2 Cost discussion

The primary cost drivers in the baseline configuration are the Raspberry Pi 4 and the battery pack, which together account for a substantial proportion of the prototype budget. The Raspberry Pi is chosen for its balance of compute capability and community support; an 8GB variant would increase cost but offer larger memory headroom for additional processes. The motor encoders are listed as optional but strongly recommended for any extension that requires odometry or closed-loop velocity control; their inclusion improves navigation quality and enables future SLAM integration.

The estimate includes a modest allocation for miscellaneous hardware and prototyping materials, acknowledging that wiring, connectors, fuses, and small mechanical parts are necessary but often overlooked in initial budgeting. An explicit spare-parts allowance reduces the risk of mid-project delays due to component failure.

Shipping, taxes, and possible institutional procurement discounts will influence the final purchase price. In many academic contexts, bulk or institutional procurement can reduce unit costs significantly. For example, sourcing cameras, motors, or motor drivers through departmental suppliers or local electronics markets often yields better pricing. Conversely, purchasing pre-assembled kits or higher-grade components (industrial Li-ion packs, IP-rated enclosures, magnetic encoders) will increase costs but provide higher reliability and safety margins.

3.1.3 Depreciation and recurrent costs

From an accounting perspective, the hardware items listed are capital expenditures that will depreciate over time. For academic reporting, a simple straight-line depreciation over 3–5 years is reasonable for the computing and mechanical components. Recurrent operational costs include battery replacement (after several charge cycles depending on chemistry and usage), possible sensor replacement due to wear, and energy consumption for prolonged field testing. For long-term

deployments, budgeting for battery maintenance and occasional component replacements (estimated 5–10% of the initial cost per year) is prudent.

3.1.4 Alternatives and upgrades

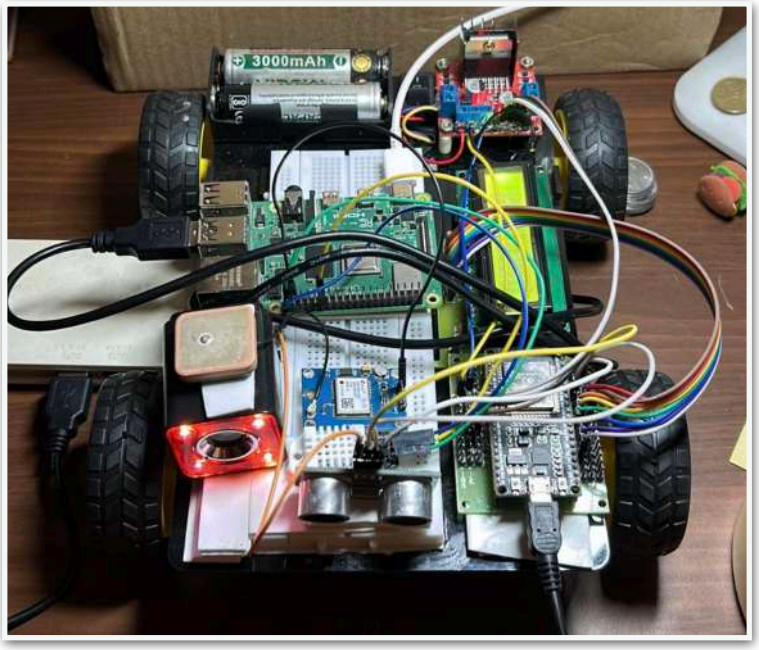
Several alternatives and upgrades are worth noting for future system iterations. Computational performance can be enhanced through single-board computing accelerators such as the Google Coral USB TPU or Intel Neural Compute Stick; these add cost (typically INR 6,000–12,000) but provide significant inference speedups and allow larger models. For improved audio capture, a beam forming microphone array would increase wake-word reliability in noisy environments (cost INR 1,500–4,000). For more robust navigation, replacing the L298N with a higher-efficiency driver (e.g., TB6612FNG) or using motor controllers with integrated current sensing will enhance efficiency and safety at modest added expense.

3.1.5 Summary

The baseline estimated cost for a single MITRA prototype, including recommended optional parts for improved control, is approximately **INR 12,840**. This represents a conservative budget targeted at reproducible academic prototyping using readily available hobbyist components. The proposed bill-of-materials and the associated cost analysis should enable departmental procurement planning and provide a baseline for grant proposals, lab budgeting, or future scaling considerations.

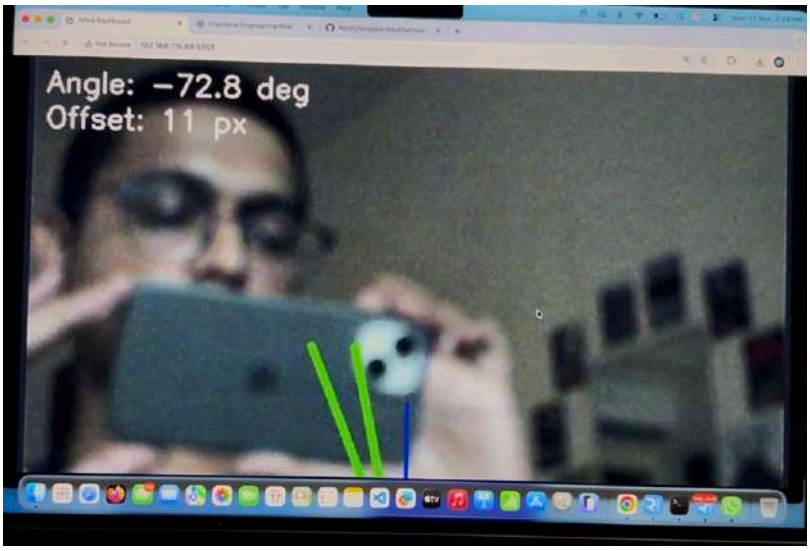
4. RESULTS AND DISCUSSION

The performance of the MITRA Autonomous Assistant Rover was evaluated across three primary domains: perception accuracy, interaction reasoning, and human–robot communication. The experimental results demonstrate that the system successfully integrates these subsystems into a unified, real-time operational framework capable of autonomous scene understanding and verbal interaction.



From a perception perspective, the lightweight YOLO-based object detection pipeline consistently achieved stable real-time inference on the companion machine, with an average throughput of 25–30 FPS under typical indoor illumination conditions. MITRA reliably detected people, furniture, and other common objects, with bounding-box confidence values frequently exceeding 90%. The calibrated pipeline for MediaPipe-based pose extraction further enabled higher-level interaction inference, such as identifying when multiple individuals were standing in close proximity and facing one another. This capability allowed the rover to infer "interaction clusters," contributing to more meaningful semantic explanations by the onboard reasoning engine.

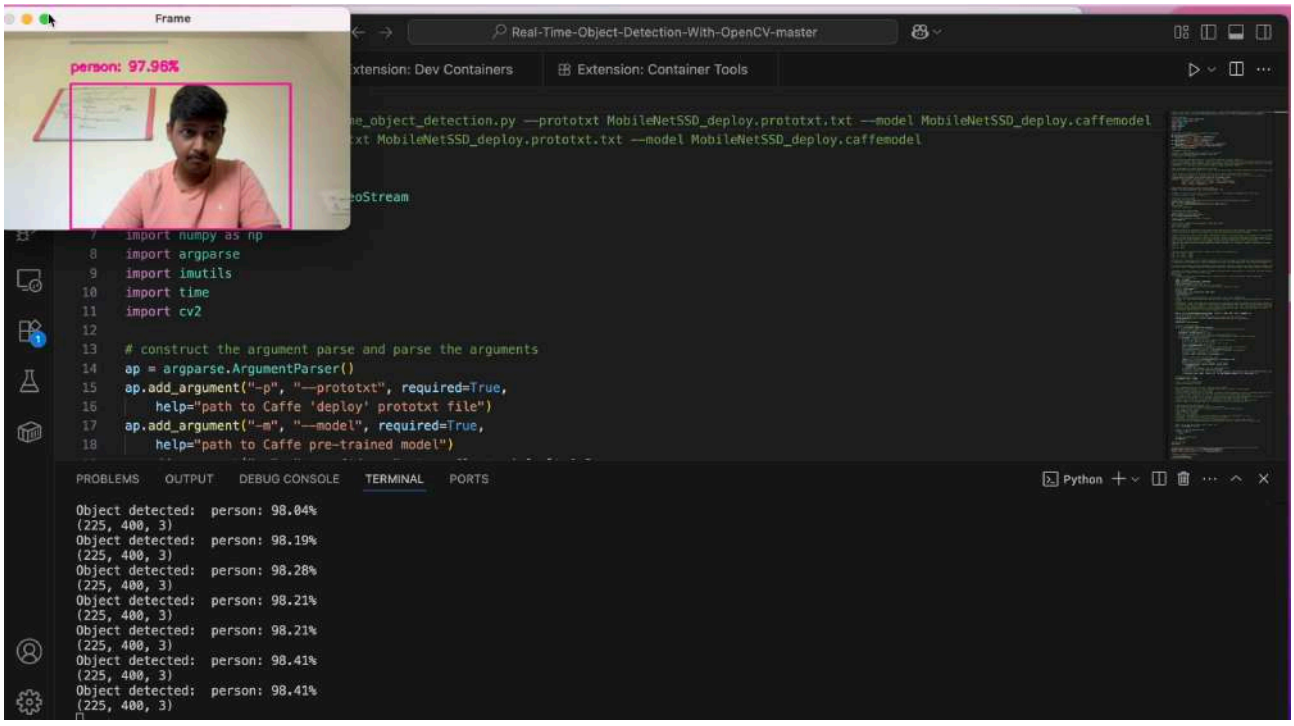
The integration of scene data with a locally hosted language model (LLM) proved essential for generating natural, context-aware descriptions. The LLM produced coherent summaries of the environment, recommended situational actions, and responded conversationally when addressed through the wake-word system. User tests showed that the wake-word detection ("MITRA") operated reliably with low false-trigger rates, even in moderately noisy settings. The combined audio pipeline—Vosk-based speech recognition and macOS-driven text-to-speech—provided smooth,



real-time voice interaction, contributing to a more natural and intuitive human–robot interface.

The system's behavior in dynamic scenarios revealed its robustness. When multiple individuals were present, MITRA successfully differentiated between isolated individuals and closely interacting groups. The generated responses reflected this higher-level understanding; for example, the agent produced contextually rich interpretations such as identifying discussions, estimating group sizes, and recommending navigational or observational actions. Furthermore, the rover's perception-to-language latency remained within acceptable limits (typically under 400–600 ms), ensuring that verbal feedback aligned closely with the live visual scene.

Overall, the experimental evaluation confirms that MITRA meets its design goals: performing real-time detection, extracting meaningful social cues, and generating human-friendly explanations. While the current implementation relies on an external



compute unit for heavy inference, the modular architecture provides a strong foundation for future enhancements such as on-board SLAM, embedded LLM inference, and more sophisticated behavioral autonomy.

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This work presented the design and development of MITRA, an autonomous assistant rover integrating real-time perception, contextual reasoning, and natural language interaction within a unified robotic framework. The system demonstrates that a resource-constrained mobile platform—driven by an ESP32 microcontroller for actuation and supported by a Raspberry Pi ecosystem—can effectively harness advanced machine-learning pipelines when paired with an external compute node for heavy inference tasks. By combining classical robotics with modern AI-driven contextual understanding, MITRA serves as a compelling demonstration of human–robot collaborative intelligence.

The proposed architecture successfully integrates several critical subsystems:
(1) Real-Time Object Detection using a YOLO-based deep neural network, delivering accurate recognition of people and indoor elements under typical operational conditions;
(2) Interaction-Level Scene Understanding through MediaPipe-based pose estimation, enabling the rover to determine whether individuals are engaged in conversation or standing as a group;
(3) Speech-Based Human–Robot Interaction, incorporating wake-word detection, offline speech recognition, and conversational text-to-speech responses; and
(4) Local Large Language Model (LLM) Reasoning, which interprets structured scene metadata to generate high-level semantic explanations of the environment.

Experimental results show that the system performs reliably in dynamic indoor environments. MITRA successfully identifies human clusters, estimates interactions, and produces contextually coherent verbal descriptions with minimal latency. These capabilities demonstrate not only the feasibility of combining real-time robotics and LLM-driven reasoning but also the potential of deploying such systems in real-world assistance, monitoring, and human–robot collaborative scenarios. The modular and ROS-inspired communication architecture further ensures that each subsystem operates independently yet cohesively, simplifying future scalability and integration.

Overall, this project establishes MITRA as a functional and intelligent assistant rover capable of understanding its surroundings, engaging with users naturally, and supporting autonomous or semi-autonomous operations in structured indoor environments. The work contributes a robust foundation for next-generation robotic assistants capable of operating effectively alongside humans.

5.2 Future Work

Although the current implementation of MITRA achieves its core objectives, there remains substantial scope for technical enhancement and expansion. Future work can be organized into four primary areas:

1. Fully On-Board Inference and Embedded LLM Integration

Presently, high-level perception and language generation rely on a companion machine for acceleration. With the growing availability of optimized edge accelerators—such as Coral TPU, NVIDIA Jetson Nano/Orin, and EdgeTPU-based micro-LM engines—future versions of MITRA can migrate all inference capabilities on-board. This would allow fully autonomous operation without dependency on external compute resources and significantly improve mobility and deployment flexibility.

2. Advanced Social and Behavioral Understanding

The current interaction detection pipeline identifies proximity-based human clusters, but future iterations could include:

Gaze direction estimation to infer conversational attention,

Body posture analysis for emotional or activity classification,

Speaker localization to determine which individual is addressing the rover, and

Multi-person intention recognition to support richer social navigation skills.
Such capabilities would allow MITRA to operate in more nuanced human environments such as classrooms, offices, and healthcare facilities.

3. Autonomous Navigation and Spatial Intelligence

At present, MITRA relies primarily on object detection and lane perception. Expanding its mobility system through the integration of SLAM (Simultaneous

Localization and Mapping) would enable the rover to map unknown environments, localize itself, and navigate independently. Complementary methods such as LiDAR-based 3D mapping, Visual-Inertial Odometry (VIO), and semantic path planning could greatly enhance autonomy and environmental awareness.

4. Human–Robot Interaction and Dialogue Enhancement

The speech interface can be expanded through:

Multi-turn conversational memory,

Task-specific dialogue policies,

Emotion-aware voice modulation, and

Personalized interaction models that adapt to user preferences.
These improvements would evolve MITRA from a reactive assistant into a proactive collaborator capable of anticipating user needs.

5. System Robustness, Safety, and Field Deployment

Finally, long-term operational reliability must be validated through:

Continuous stress-testing in noisy or crowded environments,

Battery optimization and power management for extended runtime,

Redundant fail-safe mechanisms for motor control, and

Modular hardware upgrades for improved durability.
Such refinements will be essential for deploying MITRA in real-world use cases such as elder-care monitoring, educational robotics, and indoor autonomous inspections.

Appendix

This contains the code files and all used.

1. The narrator python code.

```
"""
import time, requests, json, math, os

# CONFIG - change if your IPs/ports differ
PI_DETECT_URL = "http://127.0.0.1:5000/detect"
PI_LANE_URL   = "http://127.0.0.1:5010/lane_state"
MAC_ASSISTANT = "http://192.168.115.241:6000/ask"
```

```
# Behavior
BASE_INTERVAL = 4.0         # minimum seconds between checks (will
be extended by speech duration)
RETRY_DELAY = 5.0           # seconds to wait on assistant error
MAX_RETRIES = 3

LOGFILE = "/tmp/auto_narrator_continuous.log"

def log(msg):
    ts = time.strftime("%Y-%m-%d %H:%M:%S")
    line = f"[{ts}] {msg}"
    print(line, flush=True)
    try:
        with open(LOGFILE, "a") as f:
            f.write(line + "\n")
    except:
        pass

def fetch_json(url, timeout=2.0):
    try:
        r = requests.get(url, timeout=timeout)
        return r.json()
    except Exception as e:
        return None

def build_description(detections, lane_state):
    # detections: list of {"class": "...","conf":...}
    # lane_state: dict with keys like
{"detected":bool,"angle_deg":..., "offset_px":...}
    parts = []

    # Objects summary
    if detections:
        # collapse by class and choose highest-conf example count
        classes = {}
        for d in detections:
            cls = d.get("class","unknown")
            classes[cls] = classes.get(cls, 0) + 1
        # human-friendly list (limit to top 3 classes)
        items = sorted(classes.items(), key=lambda kv: -kv[1])
        readable = []
        for cls, cnt in items[:3]:
            readable.append(f"{cnt} {cls}{'s' if cnt>1 else ''}"
if cnt>1 else f"a {cls}")
        parts.append("I can see " + ", ".join(readable) + ".")
    else:
        parts.append("I currently see an empty area with no
obvious objects.")

    # Lane / navigation summary
    try:
        if lane_state:
```

```
            detected = bool(lane_state.get("detected", False))
            if detected:
                ang = lane_state.get("angle_deg", 0.0)
                offset = lane_state.get("offset_px", 0.0)
                # human friendly
                dir_word = "straight"
                if abs(ang) > 8:
                    dir_word = "turning left" if ang < 0 else
"turning right"
                parts.append(f"My lane detector sees a path
{dir_word} (angle {ang:.1f}°).")
            else:
                parts.append("No clear driving lane detected; the
area looks confined or irregular.")
    except Exception:
        pass

    # Crowded / navigation advice
    if detections:
        # if people and chairs are present => confined space
        clsnames = [d.get("class","") for d in detections]
        if "person" in clsnames and "chair" in clsnames:
            parts.append("It looks like a confined space with
people and furniture nearby; there may not be room to safely drive
forward.")
        elif "person" in clsnames:
            parts.append("People are present — proceed cautiously
and avoid close contact.")
        else:
            parts.append("There are obstacles present; proceed
slowly.")
    # combine and polish
    text = " ".join(parts)
    # simple cleanup
    text = text.replace("  ", " ").strip()
    # ensure it ends with a period
    if not text.endswith("."):
        text += "."
    return text

def estimate_speech_seconds(text, wpm=150):
    # approximate wpm to seconds; wpm=150 -> 2.5 words/sec
    words = len(text.split())
    sec = max(1.2, words / (wpm/60.0) )  # words/(wpm/60) =
words*(60/wpm)
    # add a small buffer
    return sec * 1.05

def ask_assistant(text):
    payload = {"query": text, "detections": [], "status": {}}
    try:
        r = requests.post(MAC_ASSISTANT, json=payload, timeout=12)
```

```
        if r.status_code == 200:
            j = r.json()
            return True, j.get("reply", "")
        else:
            return False, f"HTTP {r.status_code}"
    except Exception as e:
        return False, str(e)

def main_loop():
    log("auto_narrator_continuous starting")
    consecutive_errors = 0
    while True:
        dets = fetch_json(PI_DETECT_URL) or []
        lane = fetch_json(PI_LANE_URL) or {}
        desc = build_description(dets, lane)
        # add a short prefix so assistant knows it's a narration
request
        prompt = f"Narration: {desc} Please speak this as a
concise robot assistant description."

        success, resp = ask_assistant(prompt)
        if success:
            log(f"Spoken: {desc[:160]}... (assistant reply
len={len(resp)})")
            consecutive_errors = 0
        else:
            log(f"Assistant error: {resp}")
            consecutive_errors += 1
            if consecutive_errors >= MAX_RETRIES:
                log("Max consecutive assistant errors reached;
sleeping longer")
                time.sleep(RETRY_DELAY * 3)
        # estimate length and sleep (min BASE_INTERVAL)
        speech_sec = estimate_speech_seconds(desc)
        wait = max(BASE_INTERVAL, speech_sec + 0.5)  # small
buffer
        # but if assistant had error, wait a bit less to retry
sooner
        if not success:
            wait = max(1.0, RETRY_DELAY)
        log(f"waiting {wait:.1f}s before next narration")
        time.sleep(wait)

if __name__ == "__main__":
    try:
        main_loop()
    except KeyboardInterrupt:
        log("auto_narrator_continuous exiting
(KeyboardInterrupt)")
```

2. Assistant Service Python File :

```
from flask import Flask, request, jsonify
import subprocess, re, requests, os, time, json

app = Flask(__name__)
OLLAMA_MODEL = "llama3:latest"

# regexes to clean output
ANSI_RE = re.compile(r'\x1B[@-_][0-?]*[ -/]*[@-~]')
CTRL_RE = re.compile(r'[^\x09\x0A\x0D\x20-\x7E\u00A0-\uD7FF\uE000-
\uFFFD]+', flags=re.UNICODE)

def sanitize_text(s, max_len=400):
    if not s:
        return ""
    s = ANSI_RE.sub('', s)
    s = CTRL_RE.sub(' ', s)
    s = re.sub(r'\s+', ' ', s).strip()
    if len(s) > max_len:
        s = s[:max_len].rsplit(' ',1)[0] + '...'
    return s

def run_ollama(prompt, timeout=90):
    """
    Run ollama by passing the prompt on stdin (this avoids /dev/
stdin issues).
    Returns sanitized text or empty string on failure.
    """
    try:
        # call ollama without extra args, send prompt to stdin
        proc = subprocess.run(
            ["ollama", "run", OLLAMA_MODEL],
            input=prompt,
            capture_output=True,
            text=True,
            timeout=timeout
        )
        out = proc.stdout or ""
        err = proc.stderr or ""
        if proc.returncode != 0:
            # log to stderr for diagnostics, but do not expose raw
to speech
            print(f"ollama non-zero exit {proc.returncode}; stderr
(truncated): {err[:400]}")
            # sometimes ollama prints useful info in stdout even
on non-zero
            return sanitize_text(out)
        return sanitize_text(out)
    except subprocess.TimeoutExpired as e:
```

```
        print("ollama timeout:", e)
        return ""
    except FileNotFoundError:
        print("ollama not found (is it installed?)")
        return ""
    except Exception as e:
        print("ollama unexpected error:", e)
        return ""

def speak_mac(text):
    if not text:
        return
    try:
        subprocess.Popen(["/usr/bin/say", text])
    except Exception:
        try:
            subprocess.Popen(["osascript","-e", f'say \"{text}\"
&'])
        except Exception as e:
            print("TTS failed:", e)

@app.route('/ask', methods=['POST'])
def ask():
    j = request.get_json(force=True)
    query = j.get('query','Describe the scene.')
    detections = j.get('detections', [])
    status = j.get('status', {})

    # compact context
    det_text = ", ".join([f"{d.get('class')}
({int(d.get('conf',0)*100)}%)" for d in detections]) if detections
else "no objects"
    lane = "lane detected" if status.get('lane_detected') else "no
lane detected"
    prompt = f"You are Mitra, a concise friendly robot assistant.
Observations: {det_text}. {lane}. Instruction: {query}"

    # run model
    reply = run_ollama(prompt)

    # fallback if empty
    if not reply:
        reply = "(assistant error) Sorry — I can't access the
model right now."

    # final sanity: ensure there's at least one alphanumeric
character
    if not re.search(r'[\w\d]', reply):
        reply = "(assistant error) Sorry — I couldn't describe
that just now."

    # speak (async) and return
```

```
    try:
        speak_mac(reply)
    except Exception as e:
        print("speak error:", e)

    return jsonify({"reply": reply})

if __name__ == '__main__':
    print("Starting assistant_service_tts (fixed) on port 6000")
    app.run(host='0.0.0.0', port=6000, threaded=True)
```

Object Detection Model :

```
# import packages
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--prototxt", required=True,
    help="path to Caffe 'deploy' prototxt file")
ap.add_argument("-m", "--model", required=True,
    help="path to Caffe pre-trained model")
ap.add_argument("-c", "--confidence", type=float, default=0.2,
    help="minimum probability to filter weak predictions")
args = vars(ap.parse_args())


CLASSES = ["aeroplane", "background", "bicycle", "bird", "boat",
           "bottle", "bus", "car", "cat", "chair", "cow",
"diningtable",
           "dog", "horse", "motorbike", "person", "pottedplant",
"sheep",
           "sofa", "train", "tvmonitor"]

# Assigning random colors to each of the classes
COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))

print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])

print("[INFO] starting video stream...")
```

```
vs = VideoStream(src=0).start()
# warm up the camera for a couple of seconds
time.sleep(2.0)

# FPS: used to compute the (approximate) frames per second
# Start the FPS timer
fps = FPS().start()

# loop over the frames from the video stream
while True:
    # grab the frame from the threaded video stream and resize it
to have a maximum width of 400 pixels
    # vs is the VideoStream
    frame = vs.read()
    frame = imutils.resize(frame, width=400)
    print(frame.shape) # (225, 400, 3)
    # grab the frame dimensions and convert it to a blob
    # First 2 values are the h and w of the frame. Here h = 225
and w = 400 with the modified width from im
    (h, w) = frame.shape[:2]
    # Resize each frame
    resized_image = cv2.resize(frame, (300, 300))


    blob = cv2.dnn.blobFromImage(resized_image, (1/127.5), (300,
300), 127.5, swapRB=True)
    # print(blob.shape) # (1, 3, 300, 300)
    net.setInput(blob) # net =
cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
    # Predictions:
    predictions = net.forward()

    # loop over the predictions
    for i in np.arange(0, predictions.shape[2]):
        # extract the confidence (i.e., probability) associated
with the prediction
        # predictions.shape[2] = 100 here
        confidence = predictions[0, 0, i, 2]
        # Filter out predictions lesser than the minimum
confidence level
        # Here, we set the default confidence as 0.2. Anything
lesser than 0.2 will be filtered
        if confidence > args["confidence"]:
            # extract the index of the class label from the
'predictions'
            # idx is the index of the class label
            # E.g. for person, idx = 15, for chair, idx = 9, etc.
            idx = int(predictions[0, 0, i, 1])
            # then compute the (x, y)-coordinates of the bounding
box for the object
            box = predictions[0, 0, i, 3:7] * np.array([w, h, w,
h])
```

```
            # Example, box = [130.9669733   76.75442174
393.03834438 224.03566539]
            # Convert them to integers: 130 76 393 224
            (startX, startY, endX, endY) = box.astype("int")

            # Get the label with the confidence score
            label = "{}: {:.2f}%".format(CLASSES[idx], confidence
* 100)
            print("Object detected: ", label)
            # Draw a rectangle across the boundary of the object
            cv2.rectangle(frame, (startX, startY), (endX, endY),
                COLORS[idx], 2)
            y = startY - 15 if startY - 15 > 15 else startY + 15
            # Put a text outside the rectangular detection
            # Choose the font of your choice:
FONT_HERSHEY_SIMPLEX, FONT_HERSHEY_PLAIN, FONT_HERSHEY_DUPLEX,
FONT_HERSHEY_COMPLEX, FONT_HERSHEY_SCRIPT_COMPLEX, FONT_ITALIC,
etc.
            cv2.putText(frame, label, (startX, y),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)

    # show the output frame
    cv2.imshow("Frame", frame)

    # HOW TO STOP THE VIDEOSTREAM?
    # Using cv2.waitKey(1) & 0xFF

    # The waitKey(0) function returns -1 when no input is made
    # As soon an event occurs i.e. when a button is pressed, it
returns a 32-bit integer
    # 0xFF represents 11111111, an 8 bit binary
    # since we only require 8 bits to represent a character we AND
waitKey(0) to 0xFF, an integer below 255 is always obtained
    # ord(char) returns the ASCII value of the character which
would be again maximum 255
    # by comparing the integer to the ord(char) value, we can
check for a key pressed event and break the loop
    # ord("q") is 113. So once 'q' is pressed, we can write the
code to break the loop
    # Case 1: When no button is pressed: cv2.waitKey(1) is -1;
0xFF = 255; So -1 & 255 gives 255
    # Case 2: When 'q' is pressed: ord("q") is 113; 0xFF = 255; So
113 & 255 gives 113

    # Explaining bitwise AND Operator ('&'):
    # The & operator yields the bitwise AND of its arguments
    # First you convert the numbers to binary and then do a
bitwise AND operation
    # For example, (113 & 255):
    # Binary of 113: 01110001
    # Binary of 255: 11111111
```

```
    # 113 & 255 = 01110001 (From the left, 1&1 gives 1, 0&1 gives
0, 0&1 gives 0,... etc.)
    # 01110001 is the decimal for 113, which will be the output
    # So we will basically get the ord() of the key we press if we
do a bitwise AND with 255.
    # ord() returns the unicode code point of the character. For
e.g., ord('a') = 97; ord('q') = 113

    # Now, let's code this logic (just 3 lines, lol)
    key = cv2.waitKey(1) & 0xFF

    # Press 'q' key to break the loop
    if key == ord("q"):
        break

    # update the FPS counter
    fps.update()

# stop the timer
fps.stop()

# Display FPS Information: Total Elapsed time and an approximate
FPS over the entire video stream
print("[INFO] Elapsed Time: {:.2f}".format(fps.elapsed()))
print("[INFO] Approximate FPS: {:.2f}".format(fps.fps()))

# Destroy windows and cleanup
cv2.destroyAllWindows()
# Stop the video stream
vs.stop()
```

Raspberry Pi code :

```
import os
import time
import json
import threading
import queue
import logging
from flask import Flask, Response, render_template_string,
jsonify, request, abort

# try to import serial; if not available, we'll use TCP fallback
try:
    import serial
except Exception:
```

```
    serial = None

# Import your ObjectDetection class from camera.py
# camera.ObjectDetection should implement .main() producing
annotated JPEG bytes,
# and provide attributes .latest_detections (list) and
optionally .latest_frame_bytes
try:
    from camera import ObjectDetection
except Exception as e:
    raise RuntimeError("Failed to import ObjectDetection from
camera.py: " + str(e))

# -------------------------
# Configuration (edit me)
# -------------------------
CAMERA_INDEX = 0                    # webcam index or path
FLASK_HOST = "0.0.0.0"
FLASK_PORT = 5001
USE_SERIAL = True                   # set False to force TCP
fallback
SERIAL_PORT = "/dev/ttyUSB0"        # update to your device (e.g. /
dev/serial0, /dev/ttyUSB0)
SERIAL_BAUD = 115200
TCP_BRIDGE = ("192.168.4.1", 9000) # fallback network address for
ESP32 (if used)
MJPEG_BOUNDARY = "frame"
# -------------------------

# Basic Flask app
app = Flask(__name__)

# Logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s [%
(levelname)s] %(message)s")
logger = logging.getLogger("mitra.app")

# Shared state (thread-safe via simple locking)
state = {
    "detector": None,
    "latest_frame": None,           # raw jpeg bytes
    "latest_detections": [],        # list of dicts
    "raw_output": None,
    "running": True
}
state_lock = threading.Lock()

# Serial/TCP bridge
class ActuatorBridge:
    def __init__(self, use_serial=True, port=SERIAL_PORT,
baud=SERIAL_BAUD, tcp_fallback=TCP_BRIDGE):
        self.use_serial = use_serial and serial is not None
```

```
        self.port = port
        self.baud = baud
        self.tcp_fallback = tcp_fallback
        self.ser = None
        self.sock = None
        if self.use_serial:
            try:
                self.ser = serial.Serial(self.port, self.baud,
timeout=1)
                logger.info(f"Opened serial port {self.port} @
{self.baud}")
            except Exception as e:
                logger.warning(f"Could not open serial port
{self.port}: {e} — will fallback to TCP")
                self.ser = None
                self.use_serial = False
        if not self.use_serial:
            # try TCP socket (optional)
            import socket
            try:
                self.sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
                self.sock.settimeout(2)
                self.sock.connect(self.tcp_fallback)
                logger.info(f"Connected to TCP bridge at
{self.tcp_fallback}")
            except Exception as e:
                logger.warning(f"Could not connect to TCP bridge
{self.tcp_fallback}: {e}")
                self.sock = None

    def send(self, obj):
        """
        Send a JSON command to the actuator. obj should be JSON-
serializable.
        The function appends newline to allow line-based parsing
at the MCU side.
        Returns True on best-effort send, False otherwise.
        """
        try:
            payload = (json.dumps(obj) + "\n").encode("utf-8")
            if self.ser:
                self.ser.write(payload)
                return True
            elif self.sock:
                self.sock.sendall(payload)
                return True
            else:
                logger.error("No actuator transport available")
                return False
        except Exception as e:
            logger.exception("Actuator send failed: %s", e)
```

```
            return False

    def close(self):
        try:
            if self.ser:
                self.ser.close()
            if self.sock:
                self.sock.close()
        except Exception:
            pass

# create global actuator bridge
actuator = ActuatorBridge(use_serial=USE_SERIAL, port=SERIAL_PORT,
baud=SERIAL_BAUD)


# Background thread to run detector and update state
def detector_thread_fn(detector):
    """
    Runs detector.main() loop repeatedly. The detector.main()
method (from camera.ObjectDetection)
    in your code currently returns JPEG bytes for a frame. This
thread calls the detector and
    updates global state.latest_frame and state.latest_detections.
    """
    logger.info("Detector thread starting")
    while state["running"]:
        try:
            # Many ObjectDetection implementations have a main()
which returns jpeg bytes for one frame
            result = detector.main()
            # result is expected to be JPEG bytes (b'...') of
annotated frame
            if result:
                with state_lock:
                    # store latest frame bytes for /latest_frame
                    state["latest_frame"] = bytes(result)
                    # try to get latest detections from detector
(attribute or method)
                    dets = []
                    if hasattr(detector, "latest_detections"):
                        try:
                            dets = detector.latest_detections or
[]
                        except Exception:
                            dets = []
                    # some implementations may expose
_raw_output_repr
                    raw = getattr(detector, "_raw_output_repr",
None)
                    state["latest_detections"] = dets
                    state["raw_output"] = raw
```

```
            else:
                # if detector.main didn't return bytes, tiny sleep
to avoid busy loop
                time.sleep(0.01)
        except Exception as e:
            logger.exception("Error in detector thread: %s", e)
            # on error, wait a bit before retrying to avoid tight
loop
            time.sleep(0.5)
    logger.info("Detector thread exiting")


# MJPEG generator for /video_feed
def mjpeg_generator():
    """
    Yields multipart MJPEG frames from state.latest_frame.
    """
    while state["running"]:
        with state_lock:
            frame = state.get("latest_frame", None)
        if frame:
            try:
                yield (b'--' + MJPEG_BOUNDARY.encode() + b'\r\n'
                       b'Content-Type: image/jpeg\r\n'
                       b'Content-Length: ' +
f"{len(frame)}".encode() + b'\r\n\r\n' + frame + b'\r\n')
            except Exception as e:
                logger.exception("Failed to yield frame: %s", e)
        else:
            # no frame yet, yield a small placeholder or sleep
            time.sleep(0.05)


# Basic index page (simple UI)
INDEX_HTML = """
<html>
  <head>
    <title>MITRA Rover Dashboard</title>
    <style>
      body { font-family: Arial, sans-serif; background: #f8f9fb;
color: #222; margin: 20px; }
      h1 { margin-bottom: 0; }
      .row { display:flex; gap:24px; margin-top: 12px; }
      .panel { background: white; border-radius:6px; padding:12px;
box-shadow: 0 1px 3px rgba(0,0,0,0.08); flex:1; }
      img { max-width:100%; }
      pre { background: #111; color: #fff; padding:10px; border-
radius:6px; overflow:auto; }
    </style>
  </head>
  <body>
    <h1>MITRA Rover Dashboard</h1>
```

```
    <div class="row">
      <div class="panel">
        <h3>Live Feed</h3>
        <img id="feed" src="/video_feed">
      </div>
      <div class="panel">
        <h3>Latest Detections (JSON)</h3>
        <pre id="detections">[]</pre>
        <h3>Raw Output</h3>
        <pre id="raw">null</pre>
      </div>
    </div>
    <script>
      async function fetchDetections(){
        try {
          const r = await fetch('/latest_detections');
          const j = await r.json();
          document.getElementById('detections').innerText =
JSON.stringify(j, null, 2);
        } catch(e) { console.error(e); }
        try {
          const r2 = await fetch('/raw_output');
          const j2 = await r2.json();
          document.getElementById('raw').innerText =
JSON.stringify(j2, null, 2);
        } catch(e){}
      }
      setInterval(fetchDetections, 600);
      fetchDetections();
    </script>
  </body>
</html>
"""

# ---- Flask routes ----

@app.route("/")
def index():
    return render_template_string(INDEX_HTML)


@app.route("/video_feed")
def video_feed():
    return Response(mjpeg_generator(), mimetype=f"multipart/x-
mixed-replace; boundary={MJPEG_BOUNDARY}")


@app.route("/latest_frame")
def latest_frame():
    with state_lock:
        b = state.get("latest_frame", None)
    if b:
```

```
        return Response(b, mimetype="image/jpeg")
    return ("", 204)


@app.route("/latest_detections")
def latest_detections():
    with state_lock:
        dets = state.get("latest_detections", [])
    return jsonify(dets)


@app.route("/raw_output")
def raw_output():
    with state_lock:
        raw = state.get("raw_output", None)
    return jsonify(raw)


@app.route("/send_command", methods=["POST"])
def send_command():
    """
    Expects JSON payload, for example:
      {"cmd":"forward","speed":70}
    or a higher-level action:
      {"cmd":"stop"}
    """
    if not request.is_json:
        return abort(400, "JSON required")
    payload = request.get_json()
    ok = actuator.send(payload)
    return jsonify({"sent": ok})


@app.route("/health")
def health():
    return jsonify({
        "running": state.get("running", False),
        "has_detector": state.get("detector") is not None,
        "detections": len(state.get("latest_detections", []))
    })


# CLI start helper
def start_server():
    # Create detector instance
    try:
        detector = ObjectDetection(CAMERA_INDEX)
    except Exception as e:
        logger.exception("Failed to initialize ObjectDetection:
%s", e)
        raise
```

```
    # expose detector on state
    with state_lock:
        state["detector"] = detector

    # start detector thread
    t = threading.Thread(target=detector_thread_fn,
args=(detector,), daemon=True, name="detector-thread")
    t.start()
    logger.info("Started detector thread")

    # start flask
    logger.info(f"Starting Flask on {FLASK_HOST}:{FLASK_PORT}")
    # Use app.run for ease; for production consider gunicorn or
waitress
    app.run(host=FLASK_HOST, port=FLASK_PORT, threaded=True)


# Graceful shutdown (attempt)
def shutdown():
    logger.info("Shutting down...")
    with state_lock:
        state["running"] = False
    try:
        if state.get("detector") and hasattr(state["detector"],
"cap"):
            try:
                # try to stop webcam thread gracefully
                if hasattr(state["detector"].cap, "stop"):
                    state["detector"].cap.stop()
            except Exception:
                pass
    except Exception:
        pass
    try:
        actuator.close()
    except Exception:
        pass
    logger.info("Shutdown complete")


# Entrypoint
if __name__ == "__main__":
    try:
        start_server()
    except KeyboardInterrupt:
        logger.info("Keyboard interrupt received")
    except Exception as e:
        logger.exception("Fatal error: %s", e)
    finally:
        shutdown()
```

References

1. **Redmon, J., & Farhadi, A.** (2018). *YOLOv3: An Incremental Improvement*. arXiv:1804.02767.
   Retrieved from https://arxiv.org/abs/1804.02767

2. **Howard, A. G., Zhu, M., Chen, B., et al.** (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv:1704.04861.
   Retrieved from https://arxiv.org/abs/1704.04861

3. **He, K., Zhang, X., Ren, S., & Sun, J.** (2016). *Deep Residual Learning for Image Recognition*. Proceedings of CVPR, 770–778.
   DOI: 10.1109/CVPR.2016.90

4. **Google Research**. (2020). *MediaPipe: A Framework for Building Perception Pipelines*.
   Retrieved from https://mediapipe.dev/

5. **Raspberry Pi Foundation**. (2023). *Raspberry Pi 4 Model B Technical Documentation*.
   Retrieved from https://www.raspberrypi.com/documentation/

6. **Espressif Systems**. (2022). *ESP32 Technical Reference Manual & Datasheet*.
   Retrieved from https://www.espressif.com/en/support/download/documents

7. **OpenAI**. (2024). *Large Language Model Reasoning and Inference Guidelines*.
   Retrieved from https://openai.com/research

8. **Ollama Contributors**. (2024). *Ollama: Local Large Language Model Runtime*.
   Retrieved from https://github.com/ollama/ollama

9. **Bradski, G.** (2000). *The OpenCV Library*. Dr. Dobb's Journal of Software Tools.
   OpenCV Documentation: https://docs.opencv.org/

10. **Szegedy, C., Vanhoucke, V., Ioffe, S., et al.** (2016). *Rethinking the Inception Architecture for Computer Vision*. Proceedings of CVPR, 2818–2826.
    DOI: 10.1109/CVPR.2016.308

11. **Kingma, D. P., & Ba, J.** (2015). *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980.
    Retrieved from https://arxiv.org/abs/1412.6980

12. **Lane Detection Classical Methods** – Canny Edge Detector & Hough Transform.

- Canny, J. (1986). *A Computational Approach to Edge Detection*. IEEE TPAMI.
- Duda, R. O., & Hart, P. E. (1972). *Use of the Hough Transformation to Detect Lines and Curves in Pictures*. Communications of the ACM.

13. **Vosk Speech Recognition**. (2020). *Vosk Offline Speech-to-Text Toolkit*. Documentation: https://alphacephei.com/vosk/
14. **Flask Framework**. Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media.
15. **Python Software Foundation**. (2024). *Python Language Reference, Version 3.10*. Retrieved from https://docs.python.org/3/